The Science of Functional Programming

A tutorial, with examples in Scala

Sergei Winitzki

The Science of Functional Programming

A tutorial, with examples in Scala

by Sergei Winitzki, Ph.D.

Draft version of August 18, 2022

Published by **lulu.com** in 2022

Copyright © 2018-2022 by Sergei Winitzki

Printed copies may be ordered at http://www.lulu.com/content/paperback-book/24915714

ISBN: 978-0-359-76877-6

Source hash (sha256): 8b60d00b9deb6e7b5828f5df6c45814f2923f5e1c7c5a78b3809ddeb4760d267

Git commit: 394176c98038d51c778b593d5411973f3a2fc2a0

PDF file built by pdfTeX 3.14159265-2.6-1.40.20 (TeX Live 2019/Debian) on Thu, 18 Aug 2022 19:31:52 +0000 by Linux 5.15.0-46-generic

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the appendix entitled "GNU Free Documentation License" (Appendix G).

A *Transparent* copy of the source code for the book is available at https://github.com/winitzki/sofp and includes LyX, LaTeX, graphics source files, and build scripts. A full-color hyperlinked PDF file is available at https://github.com/winitzki/sofp/releases under "Assets" as `sofp.pdf` or `sofp-draft.pdf`. The source code may be also included as a "file attachment" named `sofp-src.tar.bz2` within a PDF file. To extract, run the command '`pdftk sofp.pdf unpack_files output .`' and then '`tar jxvf sofp-src.tar.bz2`'. See the file `README.md` for compilation instructions.

This book is a pedagogical in-depth tutorial and reference on the theory of functional programming (FP) as practiced in the early 21st century. Starting from issues found in practical coding, the book builds up the theoretical intuition, knowledge, and techniques that programmers need for rigorous reasoning about types and code. Examples are given in Scala, but most of the material applies equally to other FP languages.

The book's topics include working with FP-style collections; reasoning about recursive functions and types; the Curry-Howard correspondence; laws, structural analysis, and code for functors, monads, and other typeclasses built upon exponential-polynomial data types; techniques of symbolic derivation and proof; free typeclass constructions; and parametricity theorems.

Long and difficult, yet boring explanations are logically developed in excruciating detail through 1804 Scala code snippets, 172 statements with step-by-step derivations, 99 diagrams, 214 solved examples with tested Scala code, and 267 exercises. Discussions further build upon each chapter's material.

Beginners in FP will find tutorials about the `map/reduce` programming style, type parameters, disjunctive types, and higher-order functions. For more advanced readers, the book shows the practical uses of the Curry-Howard correspondence and the parametricity theorems without unnecessary jargon; proves that all the standard monads (e.g., `List` or `State`) satisfy the monad laws; derives lawful instances of `Functor` and other typeclasses from types; shows that monad transformers need 18 laws; and explains the use of Yoneda identities for reasoning about the Church encoding and the free typeclasses.

Readers should have a working knowledge of programming; e.g., be able to write code that prints the number of distinct words in a sentence. The difficulty of this book's mathematical derivations is at the level of undergraduate calculus, similar to that of multiplying matrices or simplifying the expressions:

$$\frac{1}{x-2}-\frac{1}{x+2} \quad \text{and} \quad \frac{d}{dx}\left((x+1)f(x)e^{-x}\right) \quad .$$

Sergei Winitzki received a Ph.D. in theoretical physics. After a career in academic research, he currently works as a software engineer.

Contents

Preface 1
Formatting conventions used in this book . . . 2

I Beginner level 3

1 Mathematical formulas as code. I. Nameless functions 4
1.1 Translating mathematics into code . . . 4
1.1.1 First examples . . . 4
1.1.2 Nameless functions . . . 5
1.1.3 Nameless functions and bound variables . . . 7
1.2 Aggregating data from sequences . . . 8
1.3 Filtering and truncating a sequence . . . 10
1.4 Solved examples . . . 10
1.4.1 Aggregations . . . 10
1.4.2 Transformations . . . 12
1.5 Summary . . . 13
1.6 Exercises . . . 14
1.6.1 Aggregations . . . 14
1.6.2 Transformations . . . 14
1.7 Discussion . . . 15
1.7.1 Functional programming as a paradigm . . . 15
1.7.2 The mathematical meaning of "variables" . . . 16
1.7.3 Iteration without loops . . . 17
1.7.4 Nameless functions in mathematical notation . . . 17
1.7.5 Named and nameless expressions and their uses . . . 19
1.7.6 Historical perspective on nameless functions . . . 20

2 Mathematical formulas as code. II. Mathematical induction 21
2.1 Tuple types . . . 21
2.1.1 Examples of using tuples . . . 21
2.1.2 Pattern matching for tuples . . . 22
2.1.3 Using tuples with collections . . . 23
2.1.4 Treating dictionaries as collections . . . 24
2.1.5 Solved examples: Tuples and collections . . . 27
2.1.6 Reasoning about type parameters in collections . . . 31
2.1.7 Exercises: Tuples and collections . . . 32
2.2 Converting a sequence into a single value . . . 33
2.2.1 Inductive definitions of aggregation functions . . . 33
2.2.2 Implementing functions by recursion . . . 35
2.2.3 Tail recursion . . . 35
2.2.4 Implementing general aggregation (`foldLeft`) . . . 39
2.2.5 Solved examples: using `foldLeft` . . . 41
2.2.6 Exercises: Using `foldLeft` . . . 44
2.3 Converting a single value into a sequence . . . 45

2.4 Transforming a sequence into another sequence . . . 47
2.5 Summary . . . 47
2.5.1 Solved examples . . . 48
2.5.2 Exercises . . . 55
2.6 Discussion and further developments . . . 58
2.6.1 Total and partial functions . . . 58
2.6.2 Scope and shadowing of pattern matching variables . . . 59
2.6.3 Lazy values and sequences. Iterators and streams . . . 59

3 The logic of types. I. Disjunctive types 63
3.1 Scala's case classes . . . 63
3.1.1 Tuple types with names . . . 63
3.1.2 Case classes with type parameters . . . 65
3.1.3 Tuples with one part and with zero parts . . . 66
3.1.4 Pattern matching for case classes . . . 67
3.2 Disjunctive types . . . 67
3.2.1 Motivation and first examples . . . 67
3.2.2 Solved examples: Pattern matching for disjunctive types . . . 69
3.2.3 Standard disjunctive types: `Option`, `Either`, `Try` . . . 72
3.3 Lists and trees as recursive disjunctive types . . . 78
3.3.1 The recursive type `List` . . . 78
3.3.2 Tail recursion with `List` . . . 80
3.3.3 Binary trees . . . 83
3.3.4 Rose trees . . . 84
3.3.5 Perfect-shaped trees . . . 84
3.3.6 Abstract syntax trees . . . 86
3.4 Summary . . . 88
3.4.1 Solved examples . . . 88
3.4.2 Exercises . . . 91
3.5 Discussion and further developments . . . 92
3.5.1 Disjunctive types as mathematical sets . . . 92
3.5.2 Disjunctive types in other programming languages . . . 94
3.5.3 Disjunctions and conjunctions in formal logic . . . 95

II Intermediate level 96

4 The logic of types. II. Curried functions 97
4.1 Functions that return functions . . . 97
4.1.1 Motivation and first examples . . . 97
4.1.2 Curried and uncurried functions . . . 98
4.1.3 Equivalence of curried and uncurried functions . . . 99
4.2 Fully parametric functions . . . 100
4.2.1 Examples. Function composition . . . 101
4.2.2 Laws of function composition . . . 103
4.2.3 Example: A function that is *not* fully parametric . . . 105
4.3 Symbolic calculations with nameless functions . . . 106
4.3.1 Calculations with curried functions . . . 106
4.3.2 Solved examples: Deriving a function's type from its code . . . 108
4.4 Summary . . . 111
4.4.1 Solved examples . . . 111
4.4.2 Exercises . . . 116

4.5 Discussion and further developments . . . 117
4.5.1 Higher-order functions . . . 117
4.5.2 Name shadowing and the scope of bound variables . . . 118
4.5.3 Operator syntax for function applications . . . 118
4.5.4 Deriving a function's code from its type . . . 120

5 The logic of types. III. The Curry-Howard correspondence 121
5.1 Values computed by fully parametric functions . . . 121
5.1.1 Motivation . . . 121
5.1.2 Type notation and $\mathcal{CH}$-propositions for standard type constructions . . . 122
5.1.3 Solved examples: Type notation . . . 125
5.1.4 Exercises: Type notation . . . 127
5.2 The logic of $\mathcal{CH}$-propositions . . . 128
5.2.1 Motivation and first examples . . . 128
5.2.2 The rules of proof for $\mathcal{CH}$-propositions . . . 129
5.2.3 Example: Proving a $\mathcal{CH}$-proposition and deriving code from type . . . 132
5.2.4 Failure of Boolean logic in reasoning about $\mathcal{CH}$-propositions . . . 135
5.3 Equivalence of types . . . 138
5.3.1 Logical identity does not correspond to type equivalence . . . 138
5.3.2 Arithmetic identity corresponds to type equivalence . . . 141
5.3.3 Type cardinalities and type equivalence . . . 145
5.3.4 Type equivalence involving function types . . . 147
5.4 Summary . . . 153
5.4.1 Solved examples . . . 154
5.4.2 Exercises . . . 162
5.5 Discussion and further developments . . . 163
5.5.1 Using the Curry-Howard correspondence for writing code . . . 163
5.5.2 Implications for designing new programming languages . . . 165
5.5.3 Uses of the void type (Scala's `Nothing`) . . . 166
5.5.4 Relationship between Boolean logic and constructive logic . . . 167
5.5.5 The constructive logic and the law of excluded middle . . . 168

6 Functors and contrafunctors 170
6.1 Practical use . . . 170
6.1.1 Motivation. Type constructors that wrap data . . . 170
6.1.2 Extended example: `Option` and the identity law . . . 171
6.1.3 Motivation for the composition law . . . 172
6.1.4 Functors: definition and examples . . . 173
6.1.5 Functor block expressions . . . 179
6.1.6 Examples of non-functors . . . 182
6.1.7 Contrafunctors . . . 186
6.1.8 Subtyping, covariance, and contravariance . . . 189
6.1.9 Solved examples: implementation of functors and contrafunctors . . . 191
6.1.10 Exercises: implementation of functors and contrafunctors . . . 195
6.2 Laws and structure . . . 196
6.2.1 Functor laws in the lifting notation . . . 196
6.2.2 Bifunctors . . . 197
6.2.3 Constructions of functors . . . 199
6.2.4 Constructions of contrafunctors . . . 206
6.2.5 Solved examples: How to recognize functors and contrafunctors . . . 208
6.3 Summary . . . 211
6.3.1 Exercises: Functor and contrafunctor constructions . . . 211

6.4 Further developments . . . 212
6.4.1 Profunctors . . . 212
6.4.2 Subtyping with injective or surjective conversion functions . . . 213

7 Reasoning about code. Techniques of symbolic derivation 215
7.1 This book's notation for code . . . 215
7.1.1 The nine constructions of fully parametric code . . . 215
7.1.2 Function composition and the pipe notation . . . 218
7.1.3 Functor and contrafunctor liftings . . . 219
7.2 Derivation techniques . . . 220
7.2.1 Standard functions for working with products . . . 220
7.2.2 Deriving laws for functions with known implementations . . . 220
7.2.3 Working with disjunctive types in matrix notation . . . 221
7.2.4 Derivations involving unknown functions with laws . . . 224
7.2.5 Exercises . . . 226

8 Typeclasses and functions of types 227
8.1 Motivation and first examples . . . 227
8.1.1 Constraining type parameters . . . 227
8.1.2 Functions of types and values . . . 227
8.1.3 Partial functions of types and values . . . 228
8.2 Implementing typeclasses . . . 229
8.2.1 Creating a partial function at type level . . . 229
8.2.2 Scala's `implicit` values . . . 231
8.2.3 Implementing typeclasses by making instances `implicit` . . . 232
8.2.4 Extension methods . . . 233
8.2.5 Solved examples: Implementing typeclasses in practice . . . 234
8.2.6 Typeclasses for type constructors . . . 239
8.3 Deriving typeclass instances via structural analysis of types . . . 241
8.3.1 Extractors . . . 241
8.3.2 Equality comparison: The `Eq` typeclass . . . 246
8.3.3 Semigroups . . . 249
8.3.4 Monoids . . . 253
8.3.5 Pointed functors: motivation and laws . . . 257
8.3.6 Pointed functors: structural analysis . . . 259
8.3.7 Co-pointed functors . . . 262
8.3.8 Pointed contrafunctors . . . 266
8.4 Summary . . . 268
8.4.1 Solved examples . . . 269
8.4.2 Exercises . . . 276
8.5 Further developments . . . 277
8.5.1 The existence of values for recursive types . . . 277
8.5.2 Proofs of associativity of `concat` for lists and arrays . . . 280
8.5.3 Higher-order type functions. Kinds . . . 282
8.5.4 Inductive typeclasses and their properties . . . 283
8.5.5 Typeclasses with several type parameters (type relations) . . . 286
8.5.6 Inheritance and automatic conversions of typeclasses . . . 287

9 Computations in functor blocks. I. Filterable functors and contrafunctors 290
9.1 Practical uses of filtering . . . 290
9.1.1 Examples and intuitions for the filtering operation . . . 291
9.1.2 The laws of filtering: Motivation and derivation . . . 292
9.1.3 Examples of non-filterable functors . . . 294

9.1.4 Solved examples: Programming with filterable functors . . . 295
9.1.5 Exercises: Programming with filterable functors . . . 300
9.2 Laws and structure . . . 301
9.2.1 Simplifying the filtering laws: Motivation for `deflate` . . . 301
9.2.2 Equivalence of `filter` and `deflate` . . . 303
9.2.3 Motivation and laws for `liftOpt` . . . 308
9.2.4 Constructions of filterable functors . . . 313
9.2.5 Filterable contrafunctors: motivation and examples . . . 323
9.2.6 Constructions of filterable contrafunctors . . . 326
9.3 Summary . . . 330
9.3.1 Solved examples . . . 330
9.3.2 Exercises . . . 335
9.4 Further developments . . . 336
9.4.1 Naturality laws and natural transformations . . . 336
9.4.2 Generalizing the laws of liftings. Kleisli functions . . . 339
9.4.3 Motivation for using category theory . . . 340

10 Computations in functor blocks. II. Semimonads and monads 344
10.1 Practical use of monads . . . 344
10.1.1 Motivation for semimonads: Nested iteration . . . 344
10.1.2 List-like monads . . . 346
10.1.3 Pass/fail monads . . . 354
10.1.4 Tree-like semimonads and monads . . . 356
10.1.5 The `Reader` monad . . . 360
10.1.6 The `Writer` monad . . . 363
10.1.7 The `State` monad . . . 364
10.1.8 The eager/lazy evaluation monad . . . 365
10.1.9 The continuation monad . . . 366
10.1.10 Exercises . . . 369
10.2 Laws of semimonads and monads . . . 370
10.2.1 Motivation for the semimonad laws . . . 370
10.2.2 The laws of `flatten` . . . 371
10.2.3 Verifying the associativity law via `flatten` . . . 373
10.2.4 From semimonads to monads: Motivation for the identity laws . . . 377
10.2.5 The monad identity laws in terms of `pure` and `flatten` . . . 378
10.2.6 Monad laws in terms of Kleisli functions . . . 378
10.2.7 Verifying the monad laws using Kleisli functions . . . 381
10.2.8 Constructions of semimonads and monads . . . 382
10.2.9 Exercises: laws and structure of monads . . . 394
10.3 Further developments . . . 396
10.3.1 Why monads must be covariant functors . . . 396
10.3.2 Equivalence of a natural transformation and a “lifting” . . . 397
10.3.3 Monads, effects, and runners . . . 399
10.3.4 Monads in category theory. Monad morphisms . . . 401
10.3.5 Constructions of polynomial monads . . . 405
10.3.6 Constructions of M-filterable functors and contrafunctors . . . 406

11 Applicative functors and contrafunctors 407
11.1 Motivation and first examples . . . 407
11.1.1 Generalizing the `zip` method from sequences to other types . . . 407
11.1.2 Gathering all errors during computations . . . 409
11.1.3 Monadic programs with independent effects . . . 410

11.2 Practical use of applicative functors . . . 411
11.2.1 Transposing a matrix via `map2` . . . 411
11.2.2 Data validation with error reporting . . . 412
11.2.3 Implementing the functions `map2`, `map3`, etc. The `ap` method . . . 412
11.2.4 The applicative `Reader` functor . . . 414
11.2.5 Single-traversal `fold` operations. I. Applicative "fold fusion" . . . 414
11.2.6 Single-traversal `fold` operations. II. Monadic "fold fusion" . . . 417
11.2.7 Parsing with applicative and monadic combinators . . . 419
11.2.8 Functor block syntax for applicative functors . . . 422
11.2.9 Exercises . . . 422
11.3 Laws and structure of applicative functors . . . 423
11.3.1 Equivalence of `map2`, `zip`, and `ap` . . . 423
11.3.2 The `Zippable` and `Applicative` typeclasses . . . 426
11.3.3 Motivation for the laws of `map2` . . . 427
11.3.4 Deriving the laws of `zip` from the laws of `map2` . . . 428
11.3.5 Commutative applicative functors and parallel computations . . . 430
11.3.6 Constructions of applicative functors . . . 432
11.4 Applicative contrafunctors and profunctors . . . 445
11.4.1 Applicative contrafunctors: Laws and constructions . . . 445
11.4.2 Applicative profunctors: Laws and constructions . . . 451
11.5 Discussion and further developments . . . 460
11.5.1 Equivalence of typeclass methods with laws . . . 460
11.5.2 Relationship between monads and applicative functors . . . 462
11.5.3 Applicative morphisms . . . 462
11.5.4 Deriving the laws of `ap` using category theory . . . 463
11.5.5 The pattern of "functorial" typeclasses in view of category theory . . . 466
11.5.6 Exercises . . . 468

12 Traversable functors **469**
12.1 Motivation . . . 469
12.1.1 From `reduce` and `foldLeft` to `foldMap` and `traverse` . . . 469
12.1.2 The `traverse` operation . . . 470
12.2 Practical use of folding and traversing operations . . . 471
12.2.1 Implementing `traverse` for various data types . . . 471
12.2.2 Aggregating tree-like data by folding. Breadth-first traversal . . . 473
12.2.3 Decorating a tree. I. Depth-first traversal . . . 475
12.2.4 Decorating a tree. II. Breadth-first traversal . . . 476
12.2.5 The `Traversable` typeclass. Implementing `scanLeft` via `traverse` . . . 479
12.2.6 Tasks that cannot be performed via `traverse` . . . 480
12.2.7 Recursion schemes. I. Folding operations . . . 481
12.2.8 Recursion schemes. II. Unfolding operations . . . 485
12.2.9 Recursion schemes. III. Traversing operations . . . 488
12.2.10 Exercises . . . 490
12.3 Laws and structure . . . 490
12.3.1 Equivalence of `reduce`, `foldLeft`, `foldMap`, and `toList`. Monoid morphisms . . . 490
12.3.2 The missing laws of `foldMap` and `reduce` . . . 498
12.3.3 All polynomial functors are foldable . . . 499
12.3.4 Equivalence of `traverse` and `sequence` . . . 500
12.3.5 Laws of `traverse` . . . 501
12.3.6 Laws of `sequence` . . . 505
12.3.7 All polynomial functors are traversable . . . 507
12.3.8 All polynomial bifunctors are bitraversable . . . 510

12.3.9 Exercises . . . 513
12.4 Discussion and further developments . . . 515
12.4.1 Traversable contrafunctors or profunctors are not useful . . . 515
12.4.2 Traversals for nested recursive types . . . 516

III Advanced level 521

13 Free typeclass constructions 522
13.1 Motivation for the free monad: create a DSL in five stages . . . 522
13.1.1 Stage 1: unevaluated expression trees . . . 522
13.1.2 Stage 2: implementing type safety in the DSLs . . . 524
13.1.3 Stage 3: implementing bound variables . . . 525
13.1.4 Stage 4: refactoring to a monadic DSL . . . 527
13.1.5 Stage 5: refactoring to reuse common code . . . 528
13.1.6 A first recipe for monadic DSLs . . . 530
13.1.7 Running a DSL program into another monad: Handling errors . . . 533
13.2 Different encodings of the free monad . . . 533
13.2.1 Motivation . . . 533
13.2.2 The raw tree encoding . . . 534
13.2.3 Deriving reduced encodings of the free monad . . . 535
13.2.4 Types with existential quantifiers . . . 536
13.3 Free constructions for other typeclasses . . . 538
13.3.1 Free constructions for inductive typeclasses . . . 538
13.3.2 Free constructions that assume other typeclasses . . . 538
13.4 Laws of free constructions . . . 538
13.5 Slides . . . 538
13.5.1 Exercises . . . 546
13.6 Properties of free constructions . . . 547
13.6.1 Free monad . . . 547
13.7 Working with quantified types . . . 550
13.7.1 The Yoneda identities . . . 550
13.7.2 Recursive type equations with different fixpoints . . . 552
13.7.3 The Church encoding of recursive types . . . 554
13.7.4 The co-Yoneda identities . . . 557
13.8 Discussion . . . 558
13.8.1 Beyond Yoneda: using parametricity to simplify quantified types . . . 558

14 Computations in functor blocks. III. Monad transformers 560
14.1 Practical use . . . 560
14.1.1 Combining monadic effects via functor composition . . . 560
14.1.2 Combining monads via monad transformers . . . 563
14.1.3 Monad transformers for standard monads . . . 565
14.1.4 Combining more than two monads: monad stacks . . . 571
14.1.5 A typeclass for monad transformers . . . 575
14.1.6 Lifts and runners for monad stacks . . . 575
14.1.7 Constructing lifts via type relations . . . 577
14.1.8 Constructing lifts via typeclasses for monad operations (“MTL-style”) . . . 578
14.2 Laws of monad transformers . . . 582
14.2.1 Motivation for the laws of lifts . . . 582
14.2.2 Motivation for the laws of runners . . . 584
14.2.3 Simplifying the laws of lifts and runners via category theory . . . 586

14.2.4 Summary of the laws of monad transformers . . . 589
14.2.5 Examples of invalid monad transformers . . . 590
14.2.6 Examples of failure to define a general monad transformer . . . 590
14.2.7 Functor composition with transformed monads . . . 593
14.2.8 Stacking two monads. Proofs . . . 593
14.2.9 Stacking any number of monads. Proofs . . . 596
14.3 Common properties of compositional transformers . . . 596
14.3.1 Motivation for the `swap` function . . . 597
14.3.2 Deriving the necessary laws for `swap` . . . 598
14.3.3 Deriving `swap` from `flatten` . . . 601
14.3.4 Monad transformer identity law. Proofs . . . 603
14.3.5 Monad transformer lifting laws. Proofs . . . 604
14.3.6 Monad transformer runner laws. Proofs . . . 605
14.3.7 Summary of results . . . 609
14.4 Composed-inside transformers. Linear monads . . . 609
14.4.1 Definitions of `swap` and `flatten` . . . 610
14.4.2 The laws of `swap`. Proofs . . . 610
14.4.3 Composition of transformers for linear monads . . . 615
14.5 Composed-outside transformers. Rigid monads . . . 615
14.5.1 Rigid monad construction 1 (product) . . . 616
14.5.2 Rigid monad construction 2 (composition) . . . 616
14.5.3 Rigid monad construction 3 (choice) . . . 617
14.6 Transformers for some monad constructions . . . 626
14.6.1 Transformer for products of monads. Proofs . . . 626
14.6.2 Transformer for free pointed monads. Proofs . . . 627
14.7 Transformers for recursive monads . . . 638
14.7.1 Transformer for free monads. Proofs . . . 638
14.7.2 Transformer for the `List` monad. Proofs . . . 650
14.8 Incomplete transformers . . . 664
14.8.1 Transformer for the state monad. Proofs . . . 664
14.8.2 Transformer for the continuation monad. Proofs . . . 668
14.8.3 Transformer for the generalized `Search` monad. Proofs . . . 669
14.8.4 The codensity monad and its transformer. Proofs . . . 672
14.8.5 Examples of monads with two different transformers . . . 676
14.9 Further developments . . . 677
14.9.1 Some properties of monad morphisms . . . 677
14.9.2 Does a composition of rigid monads have a `swap` function? . . . 678
14.9.3 Rigid functors and their properties . . . 682
14.9.4 Monadically natural lifting of MTL operations ("`Monatron`") . . . 689
14.9.5 Exercises . . . 702

IV Discussions **706**

15 Summary of the book **707**
15.1 Main points and results by chapter . . . 707
15.1.1 From Chapter 1 . . . 707
15.1.2 From Chapter 2 . . . 707
15.2 Topics not covered in this book . . . 707
15.2.1 Recursion in functor blocks and trampolines . . . 707
15.2.2 Strictness and laziness . . . 708
15.2.3 Combining typeclass laws . . . 708

15.2.4 Lenses, prisms, and other "functional optics" . . . 708
15.2.5 Comonads and comonad transformers . . . 708
15.2.6 Proving program correctness via dependent types . . . 708
15.2.7 Linear types and other non-standard types . . . 708
15.3 Additional exercises and problems . . . 708
15.3.1 Exercises . . . 708
15.3.2 Open problems . . . 710

16 "Applied functional type theory": A proposal 713
16.1 AFTT is not covered by courses in computer science . . . 713
16.2 AFTT is not category theory, type theory, or formal logic . . . 714

17 Essay: Software engineers and software artisans 718
17.1 Engineering disciplines . . . 718
17.2 Artisanship: Trades and crafts . . . 718
17.3 Programmers today are artisans, not engineers . . . 719
17.3.1 No requirements of licensing or formal study . . . 719
17.3.2 No mathematical formalism guides software development . . . 720
17.3.3 Programmers avoid academic terminology . . . 721
17.4 Towards true engineering in software . . . 721
17.5 Does software need engineers, or are artisans sufficient? . . . 723

18 Essay: Towards functional data engineering with Scala 724
18.1 Data is math . . . 724
18.2 Functional programming is math . . . 724
18.3 The power of abstraction . . . 725
18.4 Scala is Java on math . . . 726
18.5 Summary . . . 726

19 Essay: Why category theory is useful in functional programming 727
19.1 A "types/functions" category for a programming language . . . 727
19.2 The use of endofunctors . . . 728
19.3 The use of natural transformations . . . 728
19.4 Other properties of the "types/functions" category . . . 729
19.5 Some useful sub-categories of endofunctors . . . 729
19.5.1 Filterable endofunctors . . . 730
19.5.2 Monadic endofunctors . . . 730
19.6 Category theory and the laws of FP idioms . . . 731

V Appendixes 732

A Notations 733
A.1 Summary of notations . . . 733
A.2 Detailed explanations . . . 734

B Glossary of terms 739
B.1 How the term "algebra" is used in functional programming . . . 741

C Deriving code from types. The LJT algorithm 742
C.1 The LJ algorithm . . . 743
C.1.1 Extracting code from proofs . . . 744
C.2 The LJT algorithm . . . 746

D Parametricity theorem and naturality laws **748**
D.1 Practical use of parametricity properties . . . 749
D.1.1 Naturality and dinaturality laws . . . 749
D.1.2 Uniqueness of functor and contrafunctor liftings . . . 753
D.2 Commutativity laws for bifunctors and profunctors . . . 754
D.2.1 Proof of the profunctor commutativity law . . . 754
D.2.2 Commutativity laws for bifunctors and bi-contrafunctors . . . 757
D.3 Relational formulation of parametricity . . . 757
D.3.1 Relations between values of different types . . . 757
D.3.2 Relational product, co-product, and pair mapper. Relational lifting . . . 759
D.3.3 Properties of relational lifting. Simultaneous lifting . . . 766
D.3.4 Relational parametricity theorem . . . 768
D.3.5 Deriving the wedge law from relational parametricity . . . 773
D.3.6 Strong dinaturality: definition and general properties . . . 775
D.3.7 Strong dinaturality of `foldFn` . . . 783
D.4 Summary and further directions . . . 786

E Solutions of some exercises **788**

F A humorous disclaimer **809**

G GNU Free Documentation License **810**
G.0.0 Applicability and definitions . . . 810
G.0.1 Verbatim copying . . . 810
G.0.2 Copying in quantity . . . 810
G.0.3 Modifications . . . 810

List of Tables **812**

List of Figures **813**

Preface

This book is a reference text and a tutorial that teaches functional programmers how to reason mathematically about types and code, in a manner directly relevant to software practice.

The material ranges from introductory to advanced. The book assumes a certain amount of mathematical experience, at the level of familiarity with undergraduate algebra or calculus.

The vision of this book is to explain the mathematical theory that guides the practice of functional programming. So, all mathematical developments in this book are motivated by practical programming issues and are accompanied by Scala code illustrating their usage. For instance, the laws for standard typeclasses (functors, monads, etc.) are first motivated heuristically through code examples. Only then the laws are formulated as mathematical equations and proved rigorously.

To achieve a clearer presentation of the material, the book uses some non-standard notations (see Appendix A) and terminology (Appendix B). The presentation is self-contained, defining and explaining all required ideas, notations, and Scala language features.

All concepts and techniques are motivated and explained as simply as possible ("but not simpler") and also clarified via solved examples and exercises, which should be solved after absorbing the preceding material. More difficult examples and exercises are marked by an asterisk (*).

A software engineer needs to know only a few fragments of mathematical theory that will answer questions arising in the practice of functional programming. So, this book keeps theoretical material at the minimum; *vita brevis, ars longa*. Chapter 16 discusses the scope of the required theory. Mathematical generalizations are not pursued beyond proven practical relevance or direct pedagogical usefulness. This reduces the required mathematical knowledge to first notions of set theory, formal logic, and category theory. Concepts such as functors or natural transformations arise organically from the practice of reasoning about code and are first introduced without reference to category theory. This book does not use "introduction/elimination rules", "strong normalization", "complete partial orders", "adjoint functors", "pullbacks", "co-ends", or "topoi", as learning these concepts will neither help programmers write code nor answer any practical questions about code.

This book is also *not* an introduction to current theoretical research in functional programming. Instead, the focus is on material known to be practically useful. That includes constructions such as the "filterable functor" and "applicative contrafunctor" but excludes a number of theoretical developments that do not (yet?) appear to have significant applications.

The first part of the book is for beginners in functional programming. Readers already familiar with functional programming could skim the glossary (Appendix B) for unfamiliar terminology and then start the book with Chapter 5.

Chapters 5–6 begin using the code notation, such as Eq. (6.15). If that notation still appears hard to follow after going through Chapters 5–6, readers will benefit from working through Chapter 7, which summarizes the code notation more systematically and clarifies it with additional examples.

All code examples are intended only for explanation and illustration. As a rule, the code is not optimized for performance or stack safety.

The author thanks Joseph Kim and Jim Kleck for doing some of the exercises and reporting some errors in earlier versions of this book. The author also thanks Bill Venners for many helpful comments on the draft, and Harald Gliebe, Andreas Röhler, and Philip Schwarz for contributing corrections to the text via `github`. The author is grateful to Frederick Pitts and several anonymous `github` contributors who reported errors in the draft and made helpful suggestions, and to Barisere Jonathan for valuable assistance with setting up automatic builds.

Formatting conventions used in this book

- Text in boldface indicates a new concept or term that is being defined at that place in the text. Italics means logical emphasis. Example:

  An **aggregation** is a function from a sequence of values to a *single* value.

- Equations are numbered per chapter: Eq. (1.3). Statements, examples, and exercises are numbered per subsection: Example 1.4.1.6 is in subsection 1.4.1, which belongs to Chapter 1.

- Scala code is written inline using a small monospaced font, such as `flatMap` or `val a = "xyz"`. Longer code examples are written in separate code blocks and may also show the Scala interpreter's output for certain lines:

```
val s = (1 to 10).toList

scala> s.product
res0: Int = 3628800
```

- In the introductory chapters, type expressions and code examples are written in the syntax of Scala. Starting from Chapters 4–5, the book introduces and uses a new notation for types: e.g., the Scala type expression `((A, B)) => Option[A]` is written as $A \times B \to \mathbb{1} + A$. Chapters 4–7 also develop a new notation for code, intended for efficient reasoning about typeclass laws. For example, the functor composition law is written in the code notation as:

$$f^{\uparrow L} \fatsemi g^{\uparrow L} = (f \fatsemi g)^{\uparrow L} \quad ,$$

  where L is a functor and $f^{:A\to B}$ and $g^{:B\to C}$ are arbitrary functions of the specified types. The notation $f^{\uparrow L}$ denotes the function f lifted to the functor L and replaces Scala's syntax `x.map(f)` where `x` is of type `L[A]`. The symbol $\fatsemi$ denotes the forward composition of functions (Scala's method `andThen`). Appendix A summarizes these notational conventions for types and code.

- Frequently used methods of standard typeclasses, such as `pure`, `flatMap`, `flatten`, `filter`, etc., are denoted by shorter words and are labeled by the type constructor they belong to. For instance, the book talks about the methods `pure`, `flatten`, and `flatMap` for a monad M but denotes the same methods by pu_M, ftn_M, and flm_M when writing code formulas and proofs of laws.

- Derivations are written in a two-column format. The right column contains formulas in the code notation. The left column gives an explanation or indicates the property or law used to derive the expression at right from the previous expression. A green underline shows the parts of an expression that will be rewritten in the *next* step:

$$\begin{aligned}
\text{expect to equal } \text{pu}_M : \quad & \underline{\text{pu}_M^{\uparrow \text{Id}}} \fatsemi \text{pu}_M \fatsemi \text{ftn}_M \\
\text{lifting to the identity functor} : \quad & = \text{pu}_M \fatsemi \underline{\text{pu}_M \fatsemi \text{ftn}_M} \\
\text{left identity law of } M : \quad & = \text{pu}_M \quad .
\end{aligned}$$

  A green underline is sometimes also used at the last step of a derivation, to indicate the subexpression that resulted from the most recent rewriting. Other than providing hints to help remember the steps, the green underlines *play no role* in symbolic calculations.

- The symbol □ is used occasionally to indicate the end of a definition, a derivation, or a proof.

Part I

Beginner level

1 Mathematical formulas as code. I. Nameless functions

1.1 Translating mathematics into code

1.1.1 First examples

We begin by implementing some computational tasks in Scala.

Example 1.1.1.1: Factorial of 10 Find the product of integers from 1 to 10 (the **factorial** of 10).

Solution First, we write a mathematical formula for the result (usually denoted by 10!):

$$10! = 1 * 2 * ... * 10 \quad , \qquad \text{or in mathematical notation :} \quad 10! = \prod_{k=1}^{10} k \quad .$$

We can then write Scala code in a way that resembles the last formula:

```
scala> (1 to 10).product
res0: Int = 3628800
```

The code `(1 to 10).product` is an **expression**, which means that (1) the code can be evaluated and yields a value, and (2) the code can be used inside a larger expression. For example, we could write:

```
scala> 100 + (1 to 10).product + 100   // This code contains ‘(1 to 10).product‘ as a sub-expression.
res0: Int = 3629000

scala> 3628800 == (1 to 10).product
res1: Boolean = true
```

The Scala interpreter indicates that the result of `(1 to 10).product` is a value 3628800 of type `Int`. If we need to define a name for that value, we use the "`val`" syntax:

```
scala> val fac10 = (1 to 10).product
fac10: Int = 3628800
```

Example 1.1.1.2: Factorial as a function Define a function to compute the factorial of an integer n.

A mathematical formula for this function can be written as:

$$f(n) = \prod_{k=1}^{n} k \quad .$$

The corresponding Scala code is:

```
def f(n: Int) = (1 to n).product
```

In Scala's `def` syntax, we need to specify the type of a function's argument; in this case, we write `n: Int`. In the usual mathematical notation, types of arguments are either not written at all, or written separately from the formula:

$$f(n) = \prod_{k=1}^{n} k \quad , \qquad \forall n \in \mathbb{N} \quad . \tag{1.1}$$

Equation (1.1) indicates that n must be from the set of positive integers, denoted by $\mathbb{N}$ in mathematics. This is similar to specifying the type `(n: Int)` in the Scala code. So, the argument's type in the code specifies the **domain** of a function (the set of admissible arguments).

Having defined the function `f`, we can now apply it to an integer value `10` (or, as programmers say, "call" the function `f` with argument `10`):

```
scala> f(10)
res6: Int = 3628800
```

It is a **type error** to apply `f` to a non-integer value:

```
scala> f("abc")
<console>:13: error: type mismatch;
 found   : String("abc")
 required: Int
```

1.1.2 Nameless functions

Both the code written above and Eq. (1.1) involve *naming* the function as "f". Sometimes a function does not really need a name, — say, if the function is used only once. "Nameless" mathematical functions may be denoted using the symbol $\to$ (pronounced "maps to") like this:

$$x \to \text{(some formula)} \quad .$$

So, the mathematical notation for the nameless factorial function is:

$$n \to \prod_{k=1}^{n} k \quad .$$

This reads as "a function that maps n to the product of all k where k goes from 1 to n". The Scala expression implementing this mathematical formula is:

```
(n: Int) => (1 to n).product
```

This expression shows Scala's syntax for a **nameless function**. Here,

```
n: Int
```

is the function's **argument**, while

```
(1 to n).product
```

is the function's **body**. The function arrow (`=>`) separates the argument from the body.[1]

Functions in Scala (whether named or nameless) are treated as values, which means that we can also define a Scala value as:

```
scala> val fac = (n: Int) => (1 to n).product
fac: Int => Int = <function1>
```

We see that the value `fac` has the type `Int => Int`, which means that the function `fac` takes an integer (`Int`) argument and returns an integer result value. What is the value of the function `fac` *itself*? As we have just seen, the Scala interpreter prints `<function1>` as the "value" of `fac`. An alternative Scala interpreter called `ammonite`[2] prints this:

```
scala@ val fac = (n: Int) => (1 to n).product  //IGNORETHIS
fac: Int => Int = ammonite.$sess.cmd0$$$Lambda$1675/2107543287@1e44b638
```

The long number could indicate an address in memory. So, we may imagine that a "function value" represents a block of compiled code. That code will run and evaluate the function's body whenever the function is applied to an argument.

Once defined, a function can be applied to an argument like this:

[1] In mathematics, the "maps to" symbol is $\mapsto$, but this book uses a simpler arrow symbol ($\to$) that is visually similar. Many programming languages use the symbols `->` or `=>` for the function arrow; see Table 1.2.

[2] https://ammonite.io/

```
scala> fac(10)
res1: Int = 3628800
```

However, functions can be used without naming them. We may directly apply a nameless factorial function to an integer argument `10` instead of writing `fac(10)`:

```
scala> ((n: Int) => (1 to n).product)(10)
res2: Int = 3628800
```

We would rarely write code like this. Instead of creating a nameless function and then applying it right away to an argument, it is easier to evaluate the expression symbolically by substituting `10` instead of `n` in the function body:

```
((n: Int) => (1 to n).product)(10) == (1 to 10).product
```

If a nameless function uses the argument several times, as in this code:

```
((n: Int) => n*n*n + n*n)(12345)
```

it is still easier to substitute the argument and to eliminate the nameless function. We could write:

```
12345*12345*12345 + 12345*12345
```

but, of course, it is better to avoid repeating the value `12345`. To achieve that, we may define `n` as a value in an **expression block** like this:

```
scala> { val n = 12345; n*n*n + n*n }
res3: Int = 322687002
```

Defined in this way, the value `n` is visible only within the expression block. Outside the block, another value named `n` could be defined independently of this `n`. For this reason, the definition of `n` is called a **local-scope** definition.

Nameless functions are convenient when they are themselves arguments of other functions, as we will see next.

Example 1.1.2.1: prime numbers Define a function that takes an integer argument n and determines whether n is a prime number.

A simple mathematical formula for this function can be written using the "forall" symbol ($\forall$) as:

$$\text{isPrime}\,(n) = \forall k \in [2, n-1]\,.\,(n\%k) \neq 0 \quad . \tag{1.2}$$

This formula has two clearly separated parts: first, a range of integers from 2 to $n-1$, and second, a requirement that all these integers k should satisfy the given condition: $(n\%k) \neq 0$. Formula (1.2) is translated into Scala code as:

```
def isPrime(n: Int) = (2 to n-1).forall(k => n % k != 0)
```

This code looks closely similar to the mathematical notation, except for the arrow after k that introduces a nameless function `k => n % k != 0`. We do not need to specify the type `Int` for the argument `k` of that nameless function. The Scala compiler knows that `k` is going to iterate over the *integer* elements of the range `(2 to n-1)`, which effectively forces `k` to be of type `Int` because types must match.

We can now apply the function `isPrime` to some integer values:

```
scala> isPrime(12)
res3: Boolean = false

scala> isPrime(13)
res4: Boolean = true
```

As we can see from the output above, the function `isPrime` returns a value of type `Boolean`. Therefore, the function `isPrime` has type `Int => Boolean`.

A function that returns a `Boolean` value is called a **predicate**.

In Scala, it is strongly recommended (although often not mandatory) to specify the return type of named functions. The required syntax looks like this:

```
def isPrime(n: Int): Boolean = (2 to n-1).forall(k => n % k != 0)
```

1.1.3 Nameless functions and bound variables

The code for `isPrime` differs from the mathematical formula (1.2) in two ways.

One difference is that the interval $[2, n-1]$ is in front of `forall`. Another is that the Scala code uses a nameless function `(k => n % k != 0)`, while Eq. (1.2) does not seem to use such a function.

To understand the first difference, we need to keep in mind that the Scala syntax such as `(2 to n-1).forall(k => ...)` means to apply a function called `forall` to *two* arguments: the first argument is the range `(2 to n-1)`, and the second argument is the nameless function `(k => ...)`. In Scala, the **method** syntax `x.f(z)`, and the equivalent **infix** syntax `x f z`, means that a function `f` is applied to its *two* arguments, `x` and `z`. In the ordinary mathematical notation, this would be $f(x, z)$. Infix notation is widely used when it is easier to read: for instance, we write $x + y$ rather than something like $plus\,(x, y)$.

A single-argument function could be also defined as a method, and then the syntax is `x.f`, as in the expression `(1 to n).product` we have seen before.

The methods `product` and `forall` are already provided in the Scala standard library, so it is natural to use them. If we want to avoid the method syntax, we could define a function `forAll` with two arguments and write code like this:

```
forAll(2 to n-1, k => n % k != 0)
```

This would bring the syntax closer to Eq. (1.2). However, there still remains the second difference: The symbol k is used as an *argument* of a nameless function `(k => n % k != 0)` in the Scala code, while the formula:

$$\forall k \in [2, n-1]\,.\,(n\%k) \neq 0 \tag{1.3}$$

does not seem to define such a function but defines the symbol k that goes over the range $[2, n-1]$. The variable k is then used for writing the predicate $(n\%k) \neq 0$.

Let us investigate the role of k more closely. The mathematical variable k is actually defined *only inside* the expression "$\forall k$: ..." and makes no sense outside that expression. This becomes clear by looking at Eq. (1.2): The variable k is not present in the left-hand side and could not possibly be used there. The name "k" is defined only in the right-hand side, where it is first mentioned as the arbitrary element $k \in [2, n-1]$ and then used in the sub-expression "$n\%k$".

So, the mathematical notation in Eq. (1.3) says two things: First, we use the name k for integers from 2 to $n-1$. Second, for each of those k we evaluate the expression $(n\%k) \neq 0$, which can be viewed as a certain given *function of* k that returns a `Boolean` value. Translating the mathematical notation into code, it is therefore natural to use the nameless function $k \to (n\%k) \neq 0$ and to write Scala code applying this nameless function to each element of the range $[2, n-1]$ and checking that all result values be `true`:

```
(2 to n-1).forall(k => n % k != 0)
```

Just as the mathematical notation defines the variable k only in the right-hand side of Eq. (1.2), the argument `k` of the nameless Scala function `k => n % k != 0` is defined only within that function's body and cannot be used in any code outside the expression `n % k != 0`.

Variables that are defined only inside an expression and are invisible outside are called **bound variables**, or "variables bound in an expression". Variables that are used in an expression but are defined outside it are called **free variables**, or "variables occurring free in an expression". These concepts apply equally well to mathematical formulas and to Scala code. For example, in the mathematical expression $k \to (n\%k) \neq 0$ (which is a nameless function), the variable k is bound (it is defined only within that expression) but the variable n is free (it is defined outside that expression).

The main difference between free and bound variables is that bound variables can be *locally renamed* at will, unlike free variables. To see this, consider that we could rename k to z and write instead of Eq. (1.2) an equivalent definition:

$$\text{isPrime}\,(n) = \forall z \in [2, n-1]\,.\,(n\%z) \neq 0 \quad .$$

```
def isPrime(n: Int): Boolean = (2 to n-1).forall(z => n % z != 0)
```

The argument `z` in the nameless function `z => n % z != 0` may be renamed without changing the result of the entire program. No code outside that function needs to be changed after renaming `z`. But the value `n` is defined outside and cannot be renamed "locally" (i.e., only within the sub-expression). If we wanted to rename `n` in the sub-expression `z => n % z != 0`, we would also need to change all other code that defines and uses `n` *outside* that expression, or else the program would become incorrect.

Mathematical formulas use bound variables in various constructions such as $\forall k.\, p(k)$, $\exists k.\, p(k)$, $\sum_{k=a}^{b} f(k)$, $\int_0^1 k^2 dk$, $\lim_{n\to\infty} f(n)$, and $\text{argmax}_k f(k)$. When translating mathematical expressions into code, we need to recognize the presence of bound variables, which the mathematical notation does not make quite so explicit. For each bound variable, we create a nameless function whose argument is that variable, e.g., `k=>p(k)` or `k=>f(k)` for the examples just shown. Then our code will correctly reproduce the behavior of bound variables in mathematical expressions.

As an example, the mathematical formula $\forall k \in [1, n]\,.\, p(k)$ has a bound variable k and is translated into Scala code as:

```
(1 to n).forall(k => p(k))
```

At this point we can apply a simplification trick to this code. The nameless function $k \to p(k)$ does exactly the same thing as the (named) function p: It takes an argument, which we may call k, and returns $p(k)$. So, we can simplify the Scala code above to:

```
(1 to n).forall(p)
```

The simplification of $x \to f(x)$ to just f is always possible for functions f of a single argument.[3]

1.2 Aggregating data from sequences

Consider the task of counting how many even numbers there are in a given list L of integers. For example, the list $[5, 6, 7, 8, 9]$ contains *two* even numbers: 6 and 8.

A mathematical formula for this task can be written using the "sum" operation (denoted by $\sum$):

$$\text{countEven}(L) = \sum_{k \in L} \text{isEven}(k) \quad ,$$

$$\text{isEven}(k) = \begin{cases} 1 & \text{if } (k\%2) = 0 \\ 0 & \text{otherwise} \end{cases} \quad .$$

Here we defined a helper function `isEven` in order to write more easily a formula for `countEven`. In mathematics, complicated formulas are often split into simpler parts by defining helper expressions.

We can write the Scala code similarly. We first define the helper function `isEven`; the Scala code can be written in a style quite similar to the mathematical formula:

```
def isEven(k: Int): Int = (k % 2) match {
  case 0 => 1 // First, check if it is zero.
  case _ => 0 // The underscore matches everything else.
}
```

For such a simple computation, we could also write shorter code using a nameless function:

```
val isEven = (k: Int) => if (k % 2 == 0) 1 else 0
```

Given this function, we now need to translate into Scala code the expression $\sum_{k \in L} \text{is_even}(k)$. We can represent the list L using the data type `List[Int]` from the Scala standard library.

To compute $\sum_{k \in L} \text{is_even}(k)$, we must apply the function `isEven` to each element of the list L, which will produce a list of some (integer) results, and then we will need to add all those results

[3] Certain features of Scala allow programmers to write code that looks like `f(x)` but actually uses an automatic type conversion for the argument `x` or additional hidden arguments of the function `f`. In those cases, replacing the code `x => f(x)` by `f` will fail to compile. This problem does not appear when working with simple functions.

together. It is convenient to perform these two steps separately. This can be done with the functions `map` and `sum`, defined in the Scala standard library as methods for the data type `List`.

The method `sum` is similar to `product` and is defined for any `List` of numerical types (`Int`, `Float`, `Double`, etc.). It computes the sum of all numbers in the list:

```
scala> List(1, 2, 3).sum
res0: Int = 6
```

The method `map` needs more explanation. This method takes a *function* as its second argument and applies that function to each element of the list. All the results are stored in a *new* list, which is then returned as the result value:

```
scala> List(1, 2, 3).map(x => x*x + 100*x)
res1: List[Int] = List(101, 204, 309)
```

In this example, the argument of `map` is the nameless function $x \to x^2 + 100x$. This function will be used repeatedly by `map` to transform each integer from `List(1, 2, 3)`, creating a new list as a result.

It is equally possible to define the transforming function separately, give it a name, and then use it as the argument to `map`:

```
scala> def func1(x: Int): Int = x*x + 100*x
func1: (x: Int)Int

scala> List(1, 2, 3).map(func1)
res2: List[Int] = List(101, 204, 309)
```

Short functions are often defined inline, while longer functions are defined separately with a name.

A method, such as `map`, can be also used with a "dotless" (**infix**) syntax:

```
scala> List(1, 2, 3) map func1       // Same as List(1, 2, 3).map(func1)
res3: List[Int] = List(101, 204, 309)
```

If the transforming function `func1` is used only once, and especially for a simple computation such as $x \to x * x + 100 * x$, it is easier to work with a nameless function.

We can now combine the methods `map` and `sum` to define `countEven`:

```
def countEven(s: List[Int]) = s.map(isEven).sum
```

This code can be also written using a nameless function instead of `isEven`:

```
def countEven(s: List[Int]): Int = s
    .map { k => if (k % 2 == 0) 1 else 0 }
    .sum
```

In Scala, methods are often used one after another, as if in a chain. For instance, `s.map(...).sum` means: first apply `s.map(...)`, which returns a *new* list; then apply `sum` to that new list. To make the code more readable, we may put each of the chained methods on a new line.

To test this code, let us run it in the Scala interpreter. In order to let the interpreter work correctly with code entered line by line, the dot character needs to be at the *end* of the line. (In compiled code, the dots may be at the beginning of line since the compiler reads the entire file at once.)

```
scala> def countEven(s: List[Int]): Int = s.
         map { k => if (k % 2 == 0) 1 else 0 }.
         sum
countEven: (s: List[Int])Int

scala> countEven(List(1,2,3,4,5))
res0: Int = 2

scala> countEven( List(1,2,3,4,5).map(x => x * 2) )
res1: Int = 5
```

Note that the Scala interpreter prints the types differently for named functions (i.e., functions declared using `def`). It prints `(s: List[Int])Int` for a function of type `List[Int] => Int`.

1.3 Filtering and truncating a sequence

In addition to the methods `sum`, `product`, `map`, `forall` that we have already seen, the Scala standard library defines many other useful methods. We will now take a look at using the methods `max`, `min`, `exists`, `size`, `filter`, and `takeWhile`.

The methods `max`, `min`, and `size` are self-explanatory:

```
scala> List(10, 20, 30).max
res2: Int = 30

scala> List(10, 20, 30).min
res3: Int = 10

scala> List(10, 20, 30).size
res4: Int = 3
```

The methods `forall`, `exists`, `filter`, and `takeWhile` require a predicate as an argument. The `forall` method returns `true` if and only if the predicate returns `true` for all values in the list. The `exists` method returns `true` if and only if the predicate holds (returns `true`) for at least one value in the list. These methods can be written as mathematical formulas like this:

$$\text{forall}\,(S, p) = \forall k \in S.\,\big(p(k) = \text{true}\big) \quad ,$$
$$\text{exists}\,(S, p) = \exists k \in S.\,\big(p(k) = \text{true}\big) \quad .$$

However, we will use Scala syntax for operations such as "removing elements from a list".

The `filter` method returns a list that contains only the values for which a predicate returns `true`:

```
scala> List(1, 2, 3, 4, 5).filter(k => k != 3)           // Exclude the value 3.
res5: List[Int] = List(1, 2, 4, 5)
```

The `takeWhile` method truncates a given list. More precisely, `takeWhile` returns a new list that contains the initial portion of values from the original list for which predicate remains `true`:

```
scala> List(1, 2, 3, 4, 5).takeWhile(k => k != 3)    // Truncate at the value 3.
res6: List[Int] = List(1, 2)
```

In all these cases, the predicate's argument, `k`, will be of the same type as the elements in the list. In the examples shown above, the elements are integers (i.e., the lists have type `List[Int]`), therefore `k` must be of type `Int`.

The methods `sum` and `product` are defined for lists of numeric types, such as `Int` or `Float`. The methods `max` and `min` are defined on lists of "orderable" types (including `String`, `Boolean`, and the numeric types). The other methods are defined for lists of all types.

Using these methods, we can solve many problems that involve transforming and aggregating data stored in lists, arrays, sets, and other data structures that work as "containers storing values". In this context, a **transformation** is a function taking a container with values and returning a new container with changed values. (We speak of "transformation" even though the original container *remains unchanged*.) Examples of transformations are `filter` and `map`. An **aggregation** is a function taking a container of values and returning a *single* value. Examples of aggregations are `max` and `sum`.

Writing programs by chaining together various methods of transformation and aggregation is known as programming in the **map/reduce style**.

1.4 Solved examples

1.4.1 Aggregations

Example 1.4.1.1 Improve the code for `isPrime` by limiting the search to $k \leq \sqrt{n}$:

$$\text{isPrime}\,(n) = \forall k \in [2, n-1] \text{ such that if } k * k \leq n \text{ then } (n\%k) \neq 0 \quad .$$

Solution Use `takeWhile` to truncate the initial list when $k * k \leq n$ becomes false:

```
def isPrime(n: Int): Boolean =
  (2 to n-1)
    .takeWhile(k =>  k*k <= n)
    .forall(k =>  n % k != 0)
```

Example 1.4.1.2 Compute this product of absolute values: $\prod_{k=1}^{10} |\sin(k+2)|$.
Solution

```
(1 to 10)
  .map(k => math.abs(math.sin(k + 2)))
  .product
```

Example 1.4.1.3 Compute $\sum_{k \in [1,10];\, \cos k > 0} \sqrt{\cos k}$ (the sum goes only over k such that $\cos k > 0$).
Solution

```
(1 to 10)
  .filter(k => math.cos(k) > 0)
  .map(k => math.sqrt(math.cos(k)))
  .sum
```

It is safe to compute $\sqrt{\cos k}$, because we have first filtered the list by keeping only values k for which $\cos k > 0$. Let us check that this is so:

```
scala> (1 to 10).toList.filter(k => math.cos(k) > 0).map(x => math.cos(x))
res0: List[Double] = List(0.5403023058681398, 0.28366218546322625, 0.9601702866503661,
     0.7539022543433046)
```

Example 1.4.1.4 Compute the average of a non-empty list of type `List[Double]`,

$$\text{average}(s) = \frac{1}{n} \sum_{i=0}^{n-1} s_i \quad .$$

Solution We need to divide the sum by the length of the list:

```
def average(s: List[Double]): Double = s.sum / s.size

scala> average(List(1.0, 2.0, 3.0))
res0: Double = 2.0
```

Example 1.4.1.5 Given n, compute the Wallis product[4] truncated up to $\frac{2n}{2n+1}$:

$$\text{wallis}(n) = \frac{2}{1}\frac{2}{3}\frac{4}{3}\frac{4}{5}\frac{6}{5}\frac{6}{7}\cdots\frac{2n}{2n+1} \quad .$$

Solution Define the helper function `wallis_frac(i)` that computes the ith fraction. The method `toDouble` converts integers to `Double` numbers:

```
def wallis_frac(i: Int): Double = (2*i).toDouble/(2*i - 1)*(2*i)/(2*i + 1)

def wallis(n: Int) = (1 to n).map(wallis_frac).product

scala> math.cos(wallis(10000))  // Should be close to 0.
res0: Double = 3.9267453954401036E-5

scala> math.cos(wallis(100000)) // Should be even closer to 0.
res1: Double = 3.926966362362075E-6
```

The cosine of `wallis(n)` tends to zero for large n because the limit of the Wallis product is $\frac{\pi}{2}$.

[4] https://en.wikipedia.org/wiki/Wallis_product

Example 1.4.1.6 Check numerically that $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$. First, define a function of n that computes a partial sum of that series until $k = n$. Then compute the partial sum for a large value of n and compare with the limit value.

Solution

```
def euler_series(n: Int): Double = (1 to n).map(k => 1.0 / k / k).sum

scala> euler_series(100000)
res0: Double = 1.6449240668982423

scala> val pi = math.Pi
pi: Double = 3.141592653589793

scala> pi * pi / 6
res1: Double = 1.6449340668482264
```

Example 1.4.1.7 Check numerically the infinite product formula:

$$\prod_{k=1}^{\infty} \left(1 - \frac{x^2}{k^2}\right) = \frac{\sin \pi x}{\pi x} \quad .$$

Solution Compute this product up to $k = n$ for $x = 0.1$ with a large value of n, say $n = 10^5$, and compare with the right-hand side:

```
def sine_product(n: Int, x: Double): Double = (1 to n).map(k => 1.0 - x*x/k/k).product

scala> sine_product(n = 100000, x = 0.1) // Arguments may be named, for clarity.
res0: Double = 0.9836317414461351

scala> math.sin(pi * 0.1) / pi / 0.1
res1: Double = 0.9836316430834658
```

Example 1.4.1.8 Define a function p that takes a list of integers and a function `f: Int => Int`, and returns the largest value of $f(x)$ among all x in the list.

Solution

```
def p(s: List[Int], f: Int => Int): Int = s.map(f).max
```

Here is a test for this function:

```
scala> p(List(2, 3, 4, 5), x => 60 / x)
res0: Int = 30
```

1.4.2 Transformations

Example 1.4.2.1 Given a list of lists, `s: List[List[Int]]`, select the inner lists of size at least 3. The result must be again of type `List[List[Int]]`.

Solution To "select the inner lists" means to compute a *new* list containing only the desired inner lists. We use `filter` on the outer list `s`. The predicate for the filter is a function that takes an inner list and returns `true` if the size of that list is at least 3. Write the predicate as a nameless function, `t => t.size >= 3`, where `t` is of type `List[Int]`:

```
def f(s: List[List[Int]]): List[List[Int]] = s.filter(t => t.size >= 3)

scala> f(List(  List(1,2), List(1,2,3), List(1,2,3,4)  ))
res0: List[List[Int]] = List(List(1, 2, 3), List(1, 2, 3, 4))
```

The Scala compiler deduces from the code that the type of `t` is `List[Int]` because we apply `filter` to a *list of lists* of integers.

Example 1.4.2.2 Find all integers $k \in [1, 10]$ such that there are at least three different integers j, where $1 \leq j \leq k$, each j satisfying the condition $j * j > 2 * k$.

Mathematical notation	Scala code
$x \to \sqrt{x^2+1}$	`x => math.sqrt(x*x + 1)`
$[1, 2, ..., n]$	`(1 to n)`
$[f(1), ..., f(n)]$	`(1 to n).map(k => f(k))`
$\sum_{k=1}^{n} k^2$	`(1 to n).map(k => k*k).sum`
$\prod_{k=1}^{n} f(k)$	`(1 to n).map(f).product`
$\forall k \in [1,...,n].\, p(k)$ holds	`(1 to n).forall(k => p(k))`
$\exists k \in [1,...,n].\, p(k)$ holds	`(1 to n).exists(k => p(k))`
$\sum_{k \in S \text{ such that } p(k) \text{ holds}} f(k)$	`s.filter(p).map(f).sum`

Table 1.1: Translating mathematics into code.

Solution

```
scala> (1 to 10).toList.filter(k => (1 to k).filter(j => j*j > 2*k).size >= 3)
res0: List[Int] = List(6, 7, 8, 9, 10)
```

The argument of the outer `filter` is a nameless function that also uses a `filter`. The inner expression

```
(1 to k).filter(j => j*j > 2*k).size >= 3
```

(shown at left) computes the list of j's that satisfy the condition $j * j > 2 * k$, and then compares the size of that list with 3. In this way, we impose the requirement that there should be at least 3 values of j. We can see how the Scala code closely follows the mathematical formulation of the task.

1.5 Summary

Functional programs are mathematical formulas translated into code. Table 1.1 shows how to implement some often used mathematical constructions in Scala.

What problems can one solve with this knowledge?

- Compute mathematical expressions involving sums, products, and quantifiers, based on integer ranges, such as $\sum_{k=1}^{n} f(k)$.
- Transform and aggregate data from lists using `map`, `filter`, `sum`, and other methods from the Scala standard library.

What are examples of problems that are *not* solvable with these tools?

- Example 1: Compute the smallest $n \geq 1$ such that $f(f(f(...f(0)...))) \geq 1000$, where the given function f is applied n times.
- Example 2: Given a list s of numbers, compute the list r of running averages:

$$r_n = \frac{1}{n} \sum_{k=0}^{n-1} s_k \quad .$$

- Example 3: Perform binary search over a sorted list of integers.

These computations involve a general case of *mathematical induction*.

Library functions we have seen so far, such as `map` and `filter`, implement a restricted class of iterative operations on lists: namely, operations that process each element of a given list independently

and accumulate results. In those cases, the number of iterations is known (or at least bounded) in advance. For instance, when computing `s.map(f)`, the number of function applications is given by the size of the initial list. However, Example 1 requires applying a function f repeatedly until a given condition holds — that is, repeating for an *initially unknown* number of times. So it is impossible to write an expression containing `map`, `filter`, `takeWhile`, etc., that solves Example 1. We could write the solution of Example 1 as a formula by using mathematical induction, but we have not yet seen how to implement that in Scala code.

Example 2 can be formulated as a definition of a new list r by induction: the base case is $r_0 = s_0$, and the inductive step is $r_i = s_i + r_{i-1}$ for $i = 1, 2, 3, ...$ However, operations such as `map` and `filter` cannot compute r_i depending on the value of r_{i-1}.

Example 3 defines the search result by induction: the list is split in half, and search is performed recursively (i.e., using the inductive hypothesis) in the half that contains the required value. This computation requires an initially unknown number of steps.

Chapter 2 explains how to implement these tasks using recursion.

1.6 Exercises

1.6.1 Aggregations

Exercise 1.6.1.1 Define a function that computes a "staggered factorial" (denoted by $n!!$) for positive integers. It is defined as either $1 \cdot 3 \cdot ... \cdot n$ or as $2 \cdot 4 \cdot ... \cdot n$, depending on whether n is even or odd. For example, $8!! = 384$ and $9!! = 945$.

Exercise 1.6.1.2 Machin's formula[5] converges to π faster than Example 1.4.1.5:

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239} \quad ,$$

$$\arctan \frac{1}{n} = \frac{1}{n} - \frac{1}{3}\frac{1}{n^3} + \frac{1}{5}\frac{1}{n^5} - ... = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} n^{-2k-1} \quad .$$

Implement a function that computes the series for $\arctan \frac{1}{n}$ up to a given number of terms, and compute an approximation of π using this formula. Show that 12 terms of the series are already sufficient for a full-precision `Double` approximation of π.

Exercise 1.6.1.3 Using the function `isPrime`, check numerically the Euler product formula[6] for the Riemann's zeta function $\zeta(4)$. It is known[7] that $\zeta(4) = \frac{\pi^4}{90}$:

$$\zeta(4) = \prod_{k \geq 2;\ k \text{ is prime}} \frac{1}{1 - \frac{1}{p^4}} = \frac{\pi^4}{90} \quad .$$

1.6.2 Transformations

Exercise 1.6.2.1 Define a function `add20` of type `List[List[Int]] => List[List[Int]]` that adds 20 to every element of every inner list. A sample test:

```
scala> add20( List( List(1), List(2, 3) ) )
res0: List[List[Int]] = List(List(21), List(22, 23))
```

Exercise 1.6.2.2 An integer n is called a "3-factor" if it is divisible by only three different integers j such that $2 \leq j < n$. Compute the set of all "3-factor" integers n among $n \in [1, ..., 1000]$.

[5] http://turner.faculty.swau.edu/mathematics/materialslibrary/pi/machin.html
[6] https://en.wikipedia.org/wiki/Proof_of_the_Euler_product_formula_for_the_Riemann_zeta_function
[7] https://tinyurl.com/yxey4tsd

Exercise 1.6.2.3 Given a function `f: Int => Boolean`, an integer n is called a "3-f" if there are only three different integers $j \in [1, ..., n]$ such that $f(j)$ returns `true`. Define a function that takes f as an argument and returns a sequence of all "3-f" integers among $n \in [1, ..., 1000]$. What is the type of that function? Implement Exercise 1.6.2.2 using that function.

Exercise 1.6.2.4 Define a function `see100` of type `List[List[Int]] => List[List[Int]]` that selects only those inner lists whose largest value is at least 100. Test with:

```
scala> see100( List( List(0, 1, 100), List(60, 80), List(1000) ) )
res0: List[List[Int]] = List(List(0, 1, 100), List(1000))
```

Exercise 1.6.2.5 Define a function of type `List[Double] => List[Double]` that "normalizes" the list: it finds the element having the largest absolute value and, if that value is nonzero, divides all elements by that value and returns a new list; otherwise returns the original list. Test with:

```
scala> normalize(List(1.0, 4.0, 2.0))
res0: List[Double] = List(0.25, 1.0, 0.5)
```

1.7 Discussion

1.7.1 Functional programming as a paradigm

Functional programming (FP) is a **paradigm** of programming — an approach that guides programmers to write code in specific ways, applicable to a wide range of tasks.

The main idea of FP is to write code *as a mathematical expression or formula*. This approach allows programmers to derive code through logical reasoning rather than through guessing, similarly to how books on mathematics reason about mathematical formulas and derive results systematically, without guessing or "debugging." Like mathematicians and scientists who reason about formulas, functional programmers can *reason about code* systematically and logically, based on rigorous principles. This is possible only because code is written as a mathematical formula.

Mathematical intuition is useful for programming tasks because it is backed by the vast experience of working with data over millennia of human history. It took centuries to invent flexible and powerful notation, such as $\sum_{k \in S} p(k)$, and to develop the corresponding rules of calculation. Converting formulas into code, FP capitalizes on the power of these reasoning tools.

As we have seen, the Scala code for certain computational tasks corresponds quite closely to mathematical formulas (although programmers do have to write out some details that are omitted in the mathematical notation). Just as in mathematics, large code expressions may be split into smaller expressions when needed. Expressions can be easily reused, composed in various ways, and written independently from each other. Over the years, the FP community has developed a toolkit of functions (such as `map`, `filter`, etc.) that proved to be especially useful in real-life programming, although many of them are not standard in mathematical literature.

Mastering FP involves practicing to reason about programs as formulas "translated into code", building up the specific kind of applied mathematical intuition, and getting familiar with mathematical concepts adapted to a programmer's needs. The FP community has discovered a number of specific programming idioms founded on mathematical principles but driven by practical necessities of writing software. This book explains the theory behind those idioms, starting from code examples and heuristic ideas, and gradually building up the techniques of rigorous reasoning.

This chapter explored the first significant idiom of FP: iterative calculations performed without loops, in the style of mathematical expressions. This technique can be easily used in any programming language that supports nameless functions.

1.7.2 The mathematical meaning of "variables"

The usage of variables in functional programming is similar to how mathematical literature uses variables. In mathematics, **variables** are used first of all as *arguments* of functions; e.g., the formula:

$$f(x) = x^2 + x$$

contains the variable x and defines a function f that takes x as its argument (to be definite, assume that x is an integer) and computes the value $x^2 + x$. The body of the function is the expression $x^2 + x$.

Mathematics has the convention that a variable, such as x, does not change its value within a formula. Indeed, there is no mathematical notation even to talk about "changing" the value of x *inside* the formula $x^2 + x$. It would be quite confusing if a mathematics textbook said "before adding the last x in the formula $x^2 + x$, we change that x by adding 4 to it". If the "last x" in $x^2 + x$ needs to have a 4 added to it, a mathematics textbook will just write the formula $x^2 + x + 4$.

Arguments of nameless functions are also immutable. Consider, for example:

$$f(n) = \sum_{k=0}^{n} (k^2 + k) \quad .$$

Here, n is the argument of the function f, while k is the argument of the nameless function $k \to k^2 + k$. Neither n nor k can be "modified" in any sense within the expressions where they are used. The symbols k and n stand for some integer values, and these values are immutable. Indeed, it is meaningless to say that we "modified the integer 4". In the same way, we cannot modify k.

So, a variable in mathematics remains constant *within the expression* where it is defined; in that expression, a variable is essentially a "named constant". Of course, a function f can be applied to different values x, to compute a different result $f(x)$ each time. However, a given value of x will remain unmodified within the body of the function f while $f(x)$ is being computed.

Functional programming adopts this convention from mathematics: variables are immutable named constants. (Scala also has *mutable* variables, but we will not consider them in this book.)

In Scala, function arguments are immutable within the function body:

```
def f(x: Int) = x * x + x // Cannot modify ‘x‘ here.
```

The *type* of each mathematical variable (such as integer, vector, etc.) is also fixed. Each variable is a value from a specific set (e.g., the set of all integers, the set of all vectors, etc.). Mathematical formulas such as $x^2 + x$ do not express any "checking" that x is indeed an integer and not, say, a vector, in the middle of evaluating $x^2 + x$. The types of all variables are checked in advance.

Functional programming adopts the same view: Each argument of each function must have a **type** that represents the set of possible allowed values for that function argument. The programming language's compiler will automatically check the types of all arguments in advance, *before* the program runs. A program that calls functions on arguments of incorrect types will not compile.

The second usage of **variables** in mathematics is to denote expressions that will be reused. For example, one writes: let $z = \frac{x-y}{x+y}$ and now compute $\cos z + \cos 2z + \cos 3z$. Again, the variable z remains immutable, and its type remains fixed.

In Scala, this construction (defining an expression to be reused later) is written with the "`val`" syntax. Each variable defined using "`val`" is a named constant, and its type and value are fixed at the time of definition. Type annotations for "`val`"'s are optional in Scala. For instance, we could write:

```
val x: Int = 123
```

or we could omit the type annotation `: Int` and write more concisely:

```
val x = 123
```

Here, it is clear that this `x` is an integer. Nevertheless, it is often helpful to write out the types. If we do so, the compiler will check that the types match correctly and give an error message whenever wrong types are used. For example, a type error is detected when using a `String` instead of an `Int`:

```
scala> val x: Int = "123"
<console>:11: error: type mismatch;
 found   : String("123")
 required: Int
        val x: Int = "123"
                     ^
```

1.7.3 Iteration without loops

A distinctive feature of the FP paradigm is handling of iteration without writing loops.

Iterative computations are ubiquitous in mathematics. As an example, consider the formula for the standard deviation (σ) estimated from a data sample $[x_1, ..., x_n]$:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} x_i^2 - \frac{1}{n(n-1)}\left(\sum_{i=1}^{n} x_i\right)^2} \quad .$$

This expression is computed by iterating over values of the index i. And yet, no mathematics textbook uses loops or says "now repeat this formula ten times". Indeed, it would be pointless to evaluate a formula such as $x^2 + x$ ten times, because the result of $x^2 + x$ remains the same every time. It is also meaningless to "repeat" an equation such as $(x-1)(x^2+x+1) = x^3 - 1$.

Instead of loops, mathematicians write *expressions* such as $\sum_{i=1}^{n} s_i$, where symbols such as $\sum_{i=1}^{n}$ denote the results of entire iterative computations. Such computations are defined using mathematical induction. The FP paradigm has developed rich tools for translating mathematical induction into code. This chapter focuses on methods such as `map`, `filter`, and `sum`, that implement certain kinds of iterative computations. These and other similar methods can be combined in flexible ways, enabling programmers to write iterative code without loops. For example, the computation of σ according to the formula shown above may be implemented by code that looks like this:

```
def sigma(xs: Seq[Double]): Double = {
  val n = xs.length.toDouble
  val xsum = xs.sum
  val x2sum = xs.map(x => x * x).sum
  math.sqrt(x2sum / (n - 1) - xsum * xsum / n / (n - 1))
}

scala> sigma(Seq(10, 20, 30))
res0: Double = 10.0
```

The programmer can avoid writing loops because all iterative computations are delegated to functions such as `map`, `filter`, `sum`, and others. It is the job of the library and the compiler to translate those high-level functions into low-level machine code. The machine code *will* likely contain loops, but the programmer does not need to see that machine code or to reason about it.

1.7.4 Nameless functions in mathematical notation

Functions in mathematics are mappings from one set to another. A function does not necessarily *need* a name; the mapping just needs to be defined. However, nameless functions have not been widely used in the conventional mathematical notation. It turns out that nameless functions are important in functional programming because, in particular, they allow programmers to write code with a straightforward and consistent syntax.

Nameless functions contain bound variables that are invisible outside the function's scope. This property is directly reflected by the prevailing mathematical conventions. Compare the formulas:

$$f(x) = \int_0^x \frac{dx}{1+x} \quad ; \quad f(x) = \int_0^x \frac{dz}{1+z} \quad .$$

The mathematical convention is that one may rename the integration variable at will, and so these formulas define the same function f.

In programming, the only situation when a variable "may be renamed at will" is when the variable represents an argument of a function. It follows that the notations $\frac{dx}{1+x}$ and $\frac{dz}{1+z}$ correspond to a nameless function whose argument was renamed from x to z. In FP notation, this nameless function would be denoted as $z \to \frac{1}{1+z}$, and the integral rewritten as code such as:

```
integration(0, x, { z => 1.0 / (1 + z) } )
```

Now consider the traditional mathematical notation for summation, for instance:

$$\sum_{k=0}^{x} \frac{1}{1+k} \quad .$$

In that sum, the bound variable k is introduced under the $\sum$ symbol; but in integrals, the bound variable follows the special symbol "d". This notational inconsistency could be removed if we were to use nameless functions explicitly, for example:

$$\text{denote summation by } \sum_{0}^{x} \left(k \to \frac{1}{1+k} \right) \text{ instead of } \sum_{k=0}^{x} \frac{1}{1+k} \quad ,$$

$$\text{denote integration by } \int_0^x \left(z \to \frac{1}{1+z} \right) \text{ instead of } \int_0^x \frac{dz}{1+z} \quad .$$

In this notation, the new summation symbol $\sum_0^x$ does not mention the name "k" but takes a function as an argument. Similarly, the new integration symbol $\int_0^x$ does not mention "z" and does not use the special symbol "d" but now takes a function as an argument. Written in this way, the operations of summation and integration become *functions* that take functions as arguments. The above summation may be written in a consistent and straightforward manner as a Scala function:

```
summation(0, x, { y => 1.0 / (1 + y) } )
```

We could implement `summation(a, b, g)` as:

```
def summation(a: Int, b: Int, g: Int => Double): Double = (a to b).map(g).sum

scala> summation(1, 10, x => math.sqrt(x))
res0: Double = 22.4682781862041
```

Integration requires longer code since the computations are more complicated. Simpson's rule[8] is an algorithm for approximate numerical integration, defined by the formulas:

$$\text{integration}\,(a, b, g, \varepsilon) = \frac{\delta}{3}\big(g(a) + g(b) + 4s_1 + 2s_2\big) \quad ,$$

$$\text{where} \quad n = 2\left\lfloor \frac{b-a}{\varepsilon} \right\rfloor , \quad \delta_x = \frac{b-a}{n} \quad ,$$

$$s_1 = \sum_{k=1,3,\ldots,n-1} g(a + k\delta_x) \quad , \qquad s_2 = \sum_{k=2,4,\ldots,n-2} g(a + k\delta_x) \quad .$$

Here is a straightforward line-by-line translation of these formulas into Scala, with some tests:

```
def integration(a: Double, b: Double, g: Double => Double, eps: Double): Double = {
   // First, we define some helper values and functions corresponding
   // to the definitions "where n = ..." in the mathematical formulas.
   val n: Int = 2 * ((b - a) / eps).toInt
   val delta_x = (b - a) / n
   val s1 = (1 to (n - 1) by 2).map { k => g(a + k * delta_x) }.sum
```

[8] https://en.wikipedia.org/wiki/Simpson%27s_rule

```
   val s2 = (2 to (n - 2) by 2).map { k => g(a + k * delta_x) }.sum
   // Now we can write the expression for the final result.
   delta_x / 3 * (g(a) + g(b) + 4 * s1 + 2 * s2)
}

scala> integration(0, 5, x => x*x*x*x, eps = 0.01)     // The exact answer is 625.
res0: Double = 625.0000000004167

scala> integration(0, 7, x => x*x*x*x*x*x, eps = 0.01) // The exact answer is 117649.
res1: Double = 117649.00000014296
```

The entire code is one large *expression*, with a few sub-expressions (`s1`, `s2`, etc.) defined for within the **local scope** of the function (that is, within the function's body). The code contains no loops. This is similar to the way a mathematical text would define Simpson's rule. In other words, this code is written in the FP paradigm. Similar code can be written in any programming language that supports nameless functions as arguments of other functions.

1.7.5 Named and nameless expressions and their uses

It is a significant advantage if a programming language supports unnamed (or "nameless") expressions. To see this, consider a familiar situation where we take the absence of names for granted.

In today's programming languages, we may directly write expressions such as `(x+123)*y/(4+x)`. Note that the entire expression does not need to have a name. Parts of that expression (e.g., the sub-expressions `x+123` or `4+x`) also do not have separate names. It would be inconvenient if we *needed* to assign a name to each sub-expression. The code for `(x+123)*y/(4+x)` would then look like this:

```
{
  val r0 = 123
  val r1 = x + r0
  val r2 = r1 * y
  val r3 = 4
  val r4 = r3 + x
  val r5 = r2 / r4     // Do we still remember what ‘r2‘ means?
  r5
}
```

This style of programming resembles assembly languages, where every sub-expression — that is, every step of every calculation — must be assigned a separate memory address or a CPU register.

Programmers become more productive when their programming language supports nameless expressions. This is also common practice in mathematics; names are assigned when needed, but most expressions remain nameless.

It is also useful if data structures can be created without names. For instance, a **dictionary** (also called a "map" or a "hashmap") may be created in Scala with this code:

```
Map("a" -> 1, "b" -> 2, "c" -> 3)
```

This is a nameless expression whose value is a dictionary. In programming languages that do not have such a construction, programmers have to write special code that creates an initially empty dictionary and then fills it step by step with values:

```
// Scala code creating a dictionary:
Map("a" -> 1, "b" -> 2, "c" -> 3)

// Shortest Java code for the same:
new HashMap<String, Integer>() {{
   put("a", 1);
   put("b", 2);
   put("c", 3);
}}
```

Nameless functions are useful for the same reason as other nameless values: they allow us to build larger programs from simpler parts in a uniform way.

1.7.6 Historical perspective on nameless functions

Nameless functions were first used in 1936 in a theoretical programming language called "λ-calculus". In that language,[9] all functions are nameless and have a single argument. The Greek letter λ (pronounced as "lambda") is a syntax separator that denotes function arguments in nameless functions. For example, the nameless function $x \to x+1$ could be written as $\lambda x.\ add\ x\ 1$ in λ-calculus if it had a function *add* for adding integers (but it does not).

In most programming languages that were in use until around 1990, all functions required names. But by 2015, the use of nameless functions in the `map`/`reduce` programming style turned out to be so productive that most newly created languages included nameless functions, while older languages added that feature. Table 1.2 shows the year when various languages supported nameless functions.

What this book calls a "nameless function" is also known as an anonymous function, a function expression, a function literal, a closure, a lambda function, a lambda expression, or just a "lambda". This book prefers the clearer term "nameless function".

Language	**Year**	**Code for $k \to k+1$**
λ-calculus	1936	$\lambda k.\ add\ k\ 1$
typed λ-calculus	1940	$\lambda k : int.\ add\ k\ 1$
LISP	1958	`(lambda (k) (+ k 1))`
Standard ML	1973	`fn (k: int) => k + 1`
Caml	1985	`fun (k: int) -> k + 1`
Haskell	1990	`\ k -> k + 1`
Oz	1991	`fun {$ K} K + 1`
R	1993	`function(k) k + 1`
Python 1.0	1994	`lambda k: k + 1`
JavaScript	1995	`function(k) { return k + 1; }`
Mercury	1995	`func(K) = K + 1`
Ruby	1995	`lambda { |k| k + 1 }`
Lua 3.1	1998	`function(k) return k + 1 end`
Scala	2003	`(k: Int) => k + 1`
F#	2005	`fun (k: int) -> k + 1`
C# 3.0	2007	`delegate(int k) { return k + 1; }`
Clojure	2009	`(fn [k] (+ k 1))`
C++ 11	2011	`[] (int k) { return k + 1; }`
Go	2012	`func(k int) { return k + 1 }`
Julia	2012	`function(k:: Int) k + 1 end`
Kotlin	2012	`{ k: Int -> k + 1 }`
Swift	2014	`{ (k:int) -> int in return k + 1 }`
Java 8	2014	`(int k) -> k + 1`
Rust	2015	`|k: i32| k + 1`

Table 1.2: Nameless functions in programming languages.

[9] Although called a "calculus," it is a (drastically simplified) *programming language*, not related to differential or integral calculus. Also, the symbol λ has no particular significance other than a syntactic separator used in functions. Practitioners of functional programming usually do not need to study any λ-calculus. The practically relevant knowledge that comes from λ-calculus will be explained in Chapter 4.

2 Mathematical formulas as code. II. Mathematical induction

We will now study more flexible ways of working with data collections in the functional programming paradigm. The Scala standard library has methods for performing general iterative computations, that is, computations defined by induction. Translating mathematical induction into code is the focus of this chapter.

First, we need to become fluent in using tuple types with Scala collections.

2.1 Tuple types

2.1.1 Examples of using tuples

Many standard library methods in Scala work with tuple types. A simple example of a tuple is a *pair* of values, e.g., a pair of an integer and a string. The Scala syntax for this type of pair is:

```
val a: (Int, String) = (123, "xyz")
```

The type expression `(Int, String)` denotes the type of this pair.

A **triple** is defined in Scala like this:

```
val b: (Boolean, Int, Int) = (true, 3, 4)
```

Pairs and triples are examples of tuples. A **tuple** can contain any number of values, which may be called **parts** of a tuple (they are also called **fields** of a tuple). The parts of a tuple can have different types, but the type of each part is fixed once and for all. Also, the number of parts in a tuple is fixed. It is a **type error** to use incorrect types in a tuple, or an incorrect number of parts of a tuple:

```
scala> val bad: (Int, String) = (1,2)
<console>:11: error: type mismatch;
 found   : Int(2)
 required: String
       val bad: (Int, String) = (1,2)
                                  ^

scala> val bad: (Int, String) = (1,"a",3)
<console>:11: error: type mismatch;
 found   : (Int, String, Int)
 required: (Int, String)
       val bad: (Int, String) = (1,"a",3)
                                ^
```

Parts of a tuple can be accessed by number, starting from 1. The Scala syntax for **tuple accessor** methods looks like `._1`, for example:

```
scala> val a = (123, "xyz")
a: (Int, String) = (123,xyz)

scala> a._1
res0: Int = 123

scala> a._2
res1: String = xyz
```

It is a type error to access a tuple part that does not exist:

```
scala> a._0
<console>:13: error: value _0 is not a member of (Int, String)
       a._0
         ^

scala> a._5
<console>:13: error: value _5 is not a member of (Int, String)
       a._5
         ^
```

Type errors are detected at compile time, before any computations begin.

Tuples can be **nested** such that any part of a tuple can be itself a tuple:

```
scala> val c: (Boolean, (String, Int), Boolean) = (true, ("abc", 3), false)
c: (Boolean, (String, Int), Boolean) = (true,(abc,3),false)

scala> c._1
res0: Boolean = true

scala> c._2
res1: (String, Int) = (abc,3)
```

To define functions whose arguments are tuples, we could use the tuple accessors. An example of such a function is:

```
def f(p: (Boolean, Int), q: Int): Boolean = p._1 && (p._2 > q)
```

The first argument, `p`, of this function, has a tuple type. The function body uses accessor methods (`._1` and `._2`) to compute the result value. Note that the second part of the tuple `p` is of type `Int`, so it is valid to compare it with an integer `q`. It would be a type error to compare the *tuple* `p` with an *integer* using the expression `p > q`. It would be also a type error to apply the function `f` to an argument `p` that has a wrong type, e.g., the type `(Int, Int)` instead of `(Boolean, Int)`.

2.1.2 Pattern matching for tuples

Instead of using accessor methods when working with tuples, it is often convenient to use **pattern matching**. Pattern matching occurs in two situations in Scala:

- destructuring definition: `val` *pattern* `= ...`
- `case` expression: `case` *pattern* `=> ...`

```
scala> val g = (1, 2, 3)
g: (Int, Int, Int) = (1,2,3)

scala> val (x, y, z) = g
x: Int = 1
y: Int = 2
z: Int = 3
```

An example of a **destructuring definition** is shown at left. The value `g` is a tuple of three integers. After defining `g`, we define the three variables `x`, `y`, `z` *at once* in a single `val` definition. We imagine that this definition "destructures" the data structure contained in `g` and decomposes it into three parts, then assigns the names `x`, `y`, `z` to these parts. The types of `x`, `y`, `z` are also assigned automatically.

In the example above, the left-hand side of the destructuring definition contains a tuple pattern `(x, y, z)` that looks like a tuple, except that its parts are names `x`, `y`, `z` that are so far *undefined*. These names are called **pattern variables**. The destructuring definition checks whether the structure of the value of `g` "matches" the given pattern. (If `g` does not contain a tuple with exactly three parts, the definition will fail.) This computation is called **pattern matching**.

Pattern matching is often used for working with tuples. The expression `{case (a, b, c) => ...}` called a **case expression** (shown at left) performs pattern matching on its argument. The pattern matching will "destructure" (i.e., decompose) a tuple and try to match it to the given pattern `(a, b, c)`. In this pattern, `a`, `b`, `c` are as yet undefined new variables, — that is, they are pattern variables. If the pattern

```
scala> (1, 2, 3) match { case (a, b, c) => a + b + c }
res0: Int = 6
```

matching succeeds, the pattern variables `a`, `b`, `c` are assigned their values, and the function body can proceed to perform its computation. In this example, the pattern variables `a`, `b`, `c` will be assigned values 1, 2, and 3, and so the expression evaluates to 6.

Pattern matching is especially convenient for nested tuples. Here is an example where a nested tuple `p` is destructured by pattern matching:

```
def t1(p: (Int, (String, Int))): String = p match {
  case (x, (str, y)) => str + (x + y).toString
}

scala> t1((10, ("result is ", 2)))
res0: String = result is 12
```

The type structure of the argument `(Int, (String, Int))` is visually repeated in the pattern `(x, (str, y))`, making it clear that `x` and `y` become integers and `str` becomes a string after pattern matching.

If we rewrite the code of `t1` using the tuple accessor methods instead of pattern matching, the code will look like this:

```
def t2(p: (Int, (String, Int))): String = p._2._1 + (p._1 + p._2._2).toString
```

This code is shorter but harder to read. For example, it is not immediately clear that `p._2._1` is a string. It is also harder to modify this code: Suppose we want to change the type of the tuple `p` to `((Int, String), Int)`. Then the new code is:

```
def t3(p: ((Int, String), Int)): String = p._1._2 + (p._1._1 + p._2).toString
```

It takes time to verify, by going through every accessor method, that the function `t3` computes the same expression as `t2`. In contrast, the code is changed easily when using the pattern matching expression instead of the accessor methods. We only need to change the type and the pattern:

```
def t4(p: ((Int, String), Int)): String = p match {
  case ((x, str), y) => str + (x + y).toString
}
```

It is easy to see that `t4` and `t1` compute the same result. Also, the names of pattern variables may be chosen to get more clarity.

Sometimes we only need to use certain parts of a tuple in a pattern match. The following syntax is used to make that clear:

```
scala> val (x, _, _, z) = ("abc", 123, false, true)
x: String = abc
z: Boolean = true
```

The underscore symbol (_) denotes the parts of the pattern that we want to ignore. The underscore will always match any value regardless of its type.

Scala has a shorter syntax for functions such as `{case (x, y) => y}` that extract elements from tuples. The syntax looks like `(t => t._2)` or equivalently `_._2`, as illustrated here:

```
scala> val p: ((Int, Int )) => Int = { case (x, y) => y }
p: ((Int, Int)) => Int = <function1>

scala> p((1, 2))
res0: Int = 2

scala> val q: ((Int, Int )) => Int = (t => t._2)
q: ((Int, Int)) => Int = <function1>

scala> q((1, 2))
res1: Int = 2

scala> Seq( (1,10), (2,20), (3,30) ).map(_._2)
res2: Seq[Int] = List(10, 20, 30)
```

2.1.3 Using tuples with collections

Tuples can be combined with any other types without restrictions. For instance, we can define a tuple of functions:

```
val q: (Int => Int, Int => Int) = (x => x + 1, x => x - 1)
```

We can create a list of tuples:

```
val r: List[(String, Int)] = List(("apples", 3), ("oranges", 2), ("pears", 0))
```

We could define a tuple of lists of tuples of functions, or any other combination.

Here is an example of using the standard method `map` to transform a list of tuples. (As usual, we speak of "data transformation" even though the original list remains unchanged.) The argument of `map` must be a function taking a tuple as its argument. It is convenient to use pattern matching for writing such functions:

```
scala> val basket: List[(String, Int)] = List(("apples", 3), ("pears", 2), ("lemons", 0))
basket: List[(String, Int)] = List((apples,3), (pears,2), (lemons,0))

scala> basket.map { case (fruit, count) => count * 2 }
res0: List[Int] = List(6, 4, 0)

scala> basket.map { case (fruit, count) => count * 2 }.sum
res1: Int = 10
```

In this way, we can use the standard methods such as `map`, `filter`, `max`, `sum` to manipulate sequences of tuples. The names of the pattern variables "`fruit`", "`count`" are chosen to help us remember the meaning of the parts of tuples.

We can easily transform a list of tuples into a list of values of a different type:

```
scala> basket.map { case (fruit, count) =>
  val isAcidic = (fruit == "lemons")
  (fruit, isAcidic)
}
res2: List[(String, Boolean)] = List((apples,false), (pears,false), (lemons,true))
```

In the Scala syntax, a nameless function written with braces `{ ... }` may define local values in its body. The return value of the function is the last expression written in the function body. In this example, the return value of the nameless function is the tuple `(fruit, isAcidic)`.

2.1.4 Treating dictionaries as collections

In the Scala standard library, tuples are frequently used as types of intermediate values. For instance, tuples are used when iterating over dictionaries. The Scala type `Map[K, V]` represents a dictionary with keys of type `K` and values of type `V`. Here `K` and `V` are **type parameters**. Type parameters represent unknown types that will be chosen later, when working with values having specific types.

In order to create a dictionary with given keys and values, we can write:

```
Map(("apples", 3), ("oranges", 2), ("pears", 0))
```

The same result is obtained by first creating a sequence of key/value *pairs* and then converting that sequence into a dictionary via the method `toMap`:

```
List(("apples", 3), ("oranges", 2), ("pears", 0)).toMap
```

The same method works for other collection types such as `Seq`, `Vector`, `Stream`, and `Array`.

The Scala library defines a special infix syntax for pairs via the arrow symbol `->`. The expression `x -> y` is equivalent to the pair `(x, y)`:

```
scala> "apples" -> 3
res0: (String, Int) = (apples,3)
```

With this syntax, the code for creating a dictionary is easier to read:

```
Map("apples" -> 3, "oranges" -> 2, "pears" -> 0)
```

The method `toSeq` converts a dictionary into a sequence of pairs:

```
scala> Map("apples" -> 3, "oranges" -> 2, "pears" -> 0).toSeq
res20: Seq[(String, Int)] = ArrayBuffer((apples,3), (oranges,2), (pears,0))
```

The `ArrayBuffer` is one of the many list-like data structures in the Scala library. All these data structures are subtypes of the common "sequence" type `Seq`. The methods defined in the Scala standard library sometimes return different implementations of the `Seq` type for reasons of performance.

The standard library has several methods that need tuple types, such as `map` and `filter` (when used with dictionaries), `toMap`, `zip`, and `zipWithIndex`. The methods `flatten`, `flatMap`, `groupBy`, and `sliding` also work with most collection types, including dictionaries and sets. It is important to become familiar with these methods, because it will help writing code that uses sequences, sets, and dictionaries. Let us now look at these methods one by one.

The methods `map` and `toMap` Chapter 1 showed how the `map` method works on sequences: the expression `xs.map(f)` applies a given function `f` to each element of the sequence `xs`, gathering the results in a new sequence. In this sense, we can say that the `map` method "iterates over" sequences. The `map` method works similarly on dictionaries, except that iterating over a dictionary of type `Map[K, V]` when applying `map` looks like iterating over a sequence of *pairs*, `Seq[(K, V)]`. If `d: Map[K, V]` is a dictionary, the argument `f` of `d.map(f)` must be a function operating on tuples of type `(K, V)`. Typically, such functions are written using `case` expressions:

```
val fruitBasket = Map("apples" -> 3, "pears" -> 2, "lemons" -> 0)

scala> fruitBasket.map { case (fruit, count) => count * 2 }
res0: Seq[Int] = ArrayBuffer(6, 4, 0)
```

When using `map` to transform a dictionary into a sequence of pairs, the result is again a dictionary. But when an intermediate result is not a sequence of pairs, we may need to use `toMap`:

```
scala> fruitBasket.map { case (fruit, count) => (fruit, count * 2) }
res1: Map[String,Int] = Map(apples -> 6, pears -> 4, lemons -> 0)

scala> fruitBasket.map { case (fruit, count) => (fruit, count, count*2) }.
         map { case (fruit, _, count2) => (fruit, count2 / 2) }.toMap
res2: Map[String,Int] = Map(apples -> 3, pears -> 2, lemons -> 0)
```

The method `filter` works on dictionaries by iterating on key/value pairs. The filtering predicate must be a function of type `((K, V)) => Boolean`. For example:

```
scala> fruitBasket.filter { case (fruit, count) => count > 0 }
res2: Map[String,Int] = Map(apples -> 3, pears -> 2)
```

The methods `zip` and `zipWithIndex` The `zip` method takes *two* sequences and produces a sequence of pairs, taking one element from each sequence:

```
scala> val s = List(1, 2, 3)
s: List[Int] = List(1, 2, 3)

scala> val t = List(true, false, true)
t: List[Boolean] = List(true, false, true)

scala> s.zip(t)
res3: List[(Int, Boolean)] = List((1,true), (2,false), (3,true))

scala> s zip t
res4: List[(Int, Boolean)] = List((1,true), (2,false), (3,true))
```

In the last line, the equivalent "dotless" infix syntax (`s zip t`) is shown to illustrate a syntax convention of Scala that we will sometimes use.

The `zip` method works equally well on dictionaries: in that case, dictionaries are automatically converted to sequences of pairs before applying `zip`.

The `zipWithIndex` method creates a sequence of pairs where the second value in the pair is a zero-based index:

```
scala> List("a", "b", "c").zipWithIndex
res5: List[(String, Int)] = List((a,0), (b,1), (c,2))
```

The method `flatten` converts nested sequences to "flattened" ones:

```
scala> List(List(1, 2), List(2, 3), List(3, 4)).flatten
res6: List[Int] = List(1, 2, 2, 3, 3, 4)
```

Here, "flattening" means the concatenation of the inner sequences. In Scala, sequences are concatenated using the operation `++`. For example:

```
scala> List(1, 2, 3) ++ List(4, 5, 6) ++ List(0)
res7: List[Int] = List(1, 2, 3, 4, 5, 6, 0)
```

So, one can say that the `flatten` method inserts the operation `++` between all the inner sequences.

By definition, `flatten` removes *only one* level of nesting at the *top* of the data type. If applied to a `List[List[List[Int]]]`, the `flatten` method returns a `List[List[Int]]` with inner lists unchanged:

```
scala> List(List(List(1), List(2)), List(List(2), List(3))).flatten
res8: List[List[Int]] = List(List(1), List(2), List(2), List(3))
```

The method `flatMap` is closely related to `flatten` and can be seen as a shortcut, equivalent to first applying `map` and then `flatten`:

```
scala> List(1,2,3,4).map(n => (1 to n).toList)
res9: List[List[Int]] = List(List(1), List(1, 2), List(1, 2, 3), List(1, 2, 3, 4))

scala> List(1,2,3,4).map(n => (1 to n).toList).flatten
res10: List[Int] = List(1, 1, 2, 1, 2, 3, 1, 2, 3, 4)

scala> List(1,2,3,4).flatMap(n => (1 to n).toList)
res11: List[Int] = List(1, 1, 2, 1, 2, 3, 1, 2, 3, 4)
```

The `flatMap` operation transforms a sequence by mapping each element to possibly a different number of new elements.

At first sight it may be unclear why `flatMap` is useful, as `map` and `flatten` appear to be unrelated. (Should we also combine `filter` and `flatten` into a "`flatFilter`"?) However, we will see later in this book that `flatMap` describes nested iterations and can be generalized to many other data types. This chapter's examples and exercises will illustrate the use of `flatMap` with sequences.

The method `groupBy` rearranges a sequence into a dictionary where some elements of the original sequence are grouped together into subsequences. For example, given a sequence of words, we can group all words that start with the letter `"y"` into one subsequence, and all other words into another subsequence. This is accomplished by the following code:

```
scala> Seq("wombat", "xenon", "yogurt", "zebra").groupBy(s => if (s startsWith "y") 1 else 2)
res12: Map[Int,Seq[String]] = Map(1 -> List(yogurt), 2 -> List(wombat, xenon, zebra))
```

The argument of the `groupBy` method is a *function* that computes a "key" out of each sequence element. The key can have an arbitrarily chosen type. (In the current example, that type is `Int`.) The result of `groupBy` is a dictionary that maps each key to the sub-sequence of values that have that key. (In the current example, the type of the dictionary is therefore `Map[Int, Seq[String]]`.) The order of elements in the sub-sequences remains the same as in the original sequence.

As another example of using `groupBy`, the following code will group together all numbers that have the same remainder after division by 3:

```
scala> List(1,2,3,4,5).groupBy(k => k % 3)
res13: Map[Int,List[Int]] = Map(2 -> List(2, 5), 1 -> List(1, 4), 0 -> List(3))
```

The method `sliding` creates a sequence of sliding windows of a given width:

```
scala> (1 to 10).sliding(4).toList
res14: List[IndexedSeq[Int]] = List(Vector(1, 2, 3, 4), Vector(2, 3, 4, 5), Vector(3, 4, 5, 6),
    Vector(4, 5, 6, 7), Vector(5, 6, 7, 8), Vector(6, 7, 8, 9), Vector(7, 8, 9, 10))
```

After creating a nested sequence, we can apply an aggregation operation to the inner sequences. For example, the following code computes a sliding-window average with window width 50 over an

array of 100 numbers:

```
scala> (1 to 100).map(x => math.cos(x)).sliding(50).map(_.sum / 50).take(5).toList
res15: List[Double] = List(-0.005153079196990285, -0.0011160413780774369, 0.003947079736951305,
     0.005381273944717851, 0.0018679497047270743)
```

The method `sortBy` sorts a sequence according to a sorting key. The argument of `sortBy` is a *function* that computes the sorting key from a sequence element. In this way, we can sort elements in an arbitrary way:

```
scala> Seq(1, 2, 3).sortBy(x => -x)
res0: Seq[Int] = List(3, 2, 1)

scala> Seq("xx", "z", "yyy").sortBy(word => word)        // Sort alphabetically.
res1: Seq[String] = List(xx, yyy, z)

scala> Seq("xx", "z", "yyy").sortBy(word => word.length) // Sort by word length.
res2: Seq[String] = List(z, xx, yyy)
```

Sorting by the elements themselves, as we have done here with `.sortBy(word => word)`, is only possible if the element's type has a well-defined ordering. For strings, this is the alphabetic ordering, and for integers, the standard arithmetic ordering. For such types, a convenience method `sorted` is defined, and works equivalently to `sortBy(x => x)`:

```
scala> Seq("xx", "z", "yyy").sorted
res3: Seq[String] = List(xx, yyy, z)
```

2.1.5 Solved examples: Tuples and collections

Example 2.1.5.1 For a given sequence x_i, compute the sequence of pairs $b_i = (\cos x_i, \sin x_i)$.

Hint: use `map`, assume `xs:Seq[Double]`.

Solution We need to produce a sequence that has a pair of values corresponding to each element of the original sequence. This transformation is exactly what the `map` method does. So, the code is:

```
xs.map { x => (math.cos(x), math.sin(x)) }
```

Example 2.1.5.2 Count how many times $\cos x_i > \sin x_i$ occurs in a sequence x_i.

Hint: use `count`, assume `xs:Seq[Double]`.

Solution The method `count` takes a predicate and returns the number of sequence elements for which the predicate is `true`:

```
xs.count { x => math.cos(x) > math.sin(x) }
```

We could also reuse the solution of Exercise 2.1.5.1 that computed the cosine and the sine values. The code would then become:

```
xs.map { x => (math.cos(x), math.sin(x)) }
  .count { case (cosine, sine) => cosine > sine }
```

Example 2.1.5.3 For given sequences a_i and b_i, compute the sequence of differences $c_i = a_i - b_i$.

Hint: use `zip`, `map`, and assume `as` and `bs` are of type `Seq[Double]` and have equal length.

Solution We can use `zip` on `as` and `bs`, which gives a sequence of pairs:

```
as.zip(bs): Seq[(Double, Double)]
```

We then compute the differences $a_i - b_i$ by applying `map` to this sequence:

```
as.zip(bs).map { case (a, b) => a - b }
```

Example 2.1.5.4 In a given sequence p_i, count how many times $p_i > p_{i+1}$ occurs.

Hint: use `zip` and `tail`.

Solution Given `ps:Seq[Double]`, we can compute `ps.tail`. The result is a sequence that is 1 element shorter than `ps`, for example:

```
scala> val ps = Seq(1,2,3,4)
ps: Seq[Int] = List(1, 2, 3, 4)

scala> ps.tail
res0: Seq[Int] = List(2, 3, 4)
```

Taking a `zip` of the two sequences `ps` and `ps.tail`, we get a sequence of pairs:

```
scala> ps.zip(ps.tail)
res1: Seq[(Int, Int)] = List((1,2), (2,3), (3,4))
```

Note that `ps.tail` is one element shorter than `ps`, and the resulting sequence of pairs is also one element shorter than `ps`. So, it is not necessary to truncate `ps` before computing `ps.zip(ps.tail)`. Now apply the `count` method:

```
ps.zip(ps.tail).count { case (a, b) => a > b }
```

Example 2.1.5.5 For a given $k > 0$, compute the sequence $c_i = \max(b_{i-k}, ..., b_{i+k})$, starting at $i = k$.

Solution Applying the `sliding` method to a list gives a list of nested lists:

```
val b = List(1, 2, 3, 4, 5)     // An example of a possible sequence ‘b‘.

scala> b.sliding(3).toList
res0: List[List[Int]] = List(List(1, 2, 3), List(2, 3, 4), List(3, 4, 5))
```

For each i, we need to obtain a list of $2k + 1$ nearby elements $(b_{i-k}, ..., b_{i+k})$. So, we need to use `sliding(2 * k + 1)` to obtain a window of the required size. Now we can compute the maximum of each of the nested lists by using the `map` method on the outer list, with the `max` method applied to the nested lists. So, the argument of the `map` method must be the function `nested => nested.max`:

```
def c(b: List[Int], k: Int) = b.sliding(2 * k + 1).toList.map(nested => nested.max)
```

In Scala, this code can be written more concisely using the syntax:

```
def c(b: List[Int], k: Int) = b.sliding(2 * k + 1).toList.map(_.max)
```

because `_.max` means the nameless function `x => x.max`. Let us test this code:

```
scala> c(b = List(1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1), k = 1)   // Write the argument names for clarity.
res0: Seq[Int] = List(3, 4, 5, 6, 6, 6, 5, 4, 3)
```

Example 2.1.5.6 Create a 10×10 multiplication table as a dictionary of type `Map[(Int, Int), Int]`. For example, a 3×3 multiplication table would be given by this dictionary:

```
Map( (1, 1) -> 1, (1, 2) -> 2, (1, 3) -> 3, (2, 1) -> 2,
  (2, 2) -> 4, (2, 3) -> 6, (3, 1) -> 3, (3, 2) -> 6, (3, 3) -> 9 )
```

Hint: use `flatMap` and `toMap`.

Solution We are required to make a dictionary that maps pairs of integers `(x, y)` to `x * y`. Begin by creating the list of *keys* for that dictionary, which must be a list of pairs `(x, y)` of the form `List((1,1), (1,2), ..., (2,1), (2,2), ...)`. We need to iterate over a sequence of values of `x`; and for each `x`, we then need to iterate over another sequence to provide values for `y`. Try this computation:

```
scala> val s = List(1, 2, 3).map(x => List(1, 2, 3))
s: List[List[Int]] = List(List(1, 2, 3), List(1, 2, 3), List(1, 2, 3))
```

We would like to get `List((1,1), (1,2), 1,3))` etc., and so we use `map` on the inner list with a nameless function `y => (1, y)` that converts a number into a tuple:

```
scala> List(1, 2, 3).map { y => (1, y) }
res0: List[(Int, Int)] = List((1,1), (1,2), (1,3))
```

The curly braces in `{y => (1, y)}` are only for clarity. We could also use round parentheses and write `List(1, 2, 3).map(y => (1, y))`.

Now, we need to have `(x, y)` instead of `(1, y)` in the argument of `map`, where `x` iterates over `List(1, 2, 3)` in the outside scope. Using this `map` operation, we obtain:

```
scala> val s = List(1, 2, 3).map(x => List(1, 2, 3).map { y => (x, y) })
s: List[List[(Int, Int)]] = List(List((1,1), (1,2), (1,3)), List((2,1), (2,2), (2,3)), List((3,1),
     (3,2), (3,3)))
```

This is almost what we need, except that the nested lists need to be concatenated into a single list. This is exactly what `flatten` does:

```
scala> val s = List(1, 2, 3).map(x => List(1, 2, 3).map { y => (x, y) }).flatten
s: List[(Int, Int)] = List((1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3))
```

It is shorter to write `.flatMap(...)` instead of `.map(...).flatten`:

```
scala> val s = List(1, 2, 3).flatMap(x => List(1, 2, 3).map { y => (x, y) })
s: List[(Int, Int)] = List((1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3))
```

This is the list of keys for the required dictionary. The dictionary needs to map each *pair* of integers `(x, y)` to `x * y`. To create that dictionary, we will apply `toMap` to a sequence of pairs `(key, value)`, which in our case needs to be of the form of a nested tuple `((x, y), x * y)`. To achieve this, we use `map` with a function that computes the product and creates these nested tuples:

```
scala> val s = List(1, 2, 3).flatMap(x => List(1, 2, 3).map { y => (x, y) }).
  map { case (x, y) => ((x, y), x * y) }
s: List[((Int, Int), Int)] = List(((1,1),1), ((1,2),2), ((1,3),3), ((2,1),2), ((2,2),4), ((2,3),6),
     ((3,1),3), ((3,2),6), ((3,3),9))
```

We can simplify this code if we notice that we are first mapping each `y` to a tuple `(x, y)`, and later map each tuple `(x, y)` to a nested tuple `((x, y), x * y)`. Instead, the entire computation can be done in the inner `map` operation:

```
scala> val s = List(1, 2, 3).flatMap(x => List(1, 2, 3).map { y => ((x, y), x * y) } )
s: List[((Int, Int), Int)] = List(((1,1),1), ((1,2),2), ((1,3),3), ((2,1),2), ((2,2),4), ((2,3),6),
     ((3,1),3), ((3,2),6), ((3,3),9))
```

It remains to convert this list of tuples to a dictionary with `toMap`. Also, for better readability, we can use Scala's pair syntax, `key -> value`, which is equivalent to writing the tuple `(key, value)`:

```
(1 to 10).flatMap(x => (1 to 10).map { y => (x, y) -> x * y }).toMap
```

Example 2.1.5.7 For a given sequence x_i, compute the maximum of all of the numbers x_i, x_i^2, $\cos x_i$, $\sin x_i$. Hint: use `flatMap` and `max`.

Solution We will compute the required value if we take `max` of a list containing all of the numbers. To do that, first map each element of the list `xs: Seq[Double]` into a sequence of three numbers:

```
scala> val xs = List(0.1, 0.5, 0.9)         // An example list of some ‘Double‘ numbers.
xs: List[Double] = List(0.1, 0.5, 0.9)

scala> xs.map { x => Seq(x, x*x, math.cos(x), math.sin(x)) }
res0: List[Seq[Double]] = List(List(0.1, 0.010000000000000002, 0.9950041652780258,
     0.09983341664682815), List(0.5, 0.25, 0.8775825618903728, 0.479425538604203), List(0.9, 0.81,
     0.6216099682706644, 0.7833269096274834))
```

This list is almost what we need, except we need to `flatten` it:

```
scala> res0.flatten
res1: List[Double] = List(0.1, 0.010000000000000002, 0.9950041652780258, 0.09983341664682815, 0.5,
     0.25, 0.8775825618903728, 0.479425538604203, 0.9, 0.81, 0.6216099682706644, 0.7833269096274834)
```

It remains to take the maximum of the resulting numbers:

```
scala> res1.max
res2: Double = 0.9950041652780258
```

The final code (starting from a given sequence `xs`) is:

```
xs.flatMap { x => Seq(x, x*x, math.cos(x), math.sin(x)) }.max
```

Example 2.1.5.8 From a dictionary of type `Map[String, String]` mapping names to addresses, and assuming that the addresses do not repeat, compute a dictionary of type `Map[String, String]` mapping the addresses back to names.

Solution Iterating over a dictionary looks like iterating over a list of `(key, value)` pairs:

```
dict.map { case (name, addr) => (addr, name) }   // The result is converted to Map automatically.
```

Example 2.1.5.9 Write the solution of Example 2.1.5.8 as a function with type parameters `Name` and `Addr` instead of the fixed type `String`.

Solution In Scala, the syntax for type parameters in a function definition is:

```
def rev[Name, Addr](...) = ...
```

The type of the argument is `Map[Name, Addr]`, while the type of the result is `Map[Addr, Name]`. So, we use the type parameters `Name` and `Addr` in the type signature of the function. The final code is:

```
def rev[Name, Addr](dict: Map[Name, Addr]): Map[Addr, Name] =
  dict.map { case (name, addr) => (addr, name) }
```

The body of the function `rev` remains the same as in Example 2.1.5.8; only the type signature changes. This is because the procedure for reversing a dictionary works in the same way for dictionaries of any type. So the body of the function `rev` does not actually need to know the types of the keys and values in the dictionary. For this reason, it was easy for us to change the specific type `String` into type parameters in that function.

When the function `rev` is applied to a dictionary of a specific type, the Scala compiler will automatically set the type parameters `Name` and `Addr` that fit the required types of the dictionary's keys and values. For example, if we apply `rev` to a dictionary of type `Map[Boolean, Seq[String]]`, the type parameters will be set automatically as `Name = Boolean` and `Addr = Seq[String]`:

```
scala> val d = Map(true -> Seq("x", "y"), false -> Seq("z", "t"))
d: Map[Boolean, Seq[String]] = Map(true -> List(x, y), false -> List(z, t))

scala> rev(d)
res0: Map[Seq[String], Boolean] = Map(List(x, y) -> true, List(z, t) -> false)
```

Type parameters can be also set explicitly when using the function `rev`. If the type parameters are chosen incorrectly, the program will not compile:

```
scala> rev[Boolean, Seq[String]](d)
res1: Map[Seq[String],Boolean] = Map(List(x, y) -> true, List(z, t) -> false)

scala> rev[Int, Double](d)
<console>:14: error: type mismatch;
 found   : Map[Boolean,Seq[String]]
 required: Map[Int,Double]
       rev[Int, Double](d)
                        ^
```

Example 2.1.5.10* Given a sequence `words: Seq[String]` of some "words", compute a sequence of type `Seq[(Seq[String], Int)]`, where each inner sequence should contain all the words having the same length, paired with the integer value showing that length. The resulting sequence must be ordered by increasing length of words. So, the input `Seq("the", "food", "is", "good")` should produce:

```
Seq((Seq("is"), 2), (Seq("the"), 3), (Seq("food", "good"), 4))
```

Solution Begin by grouping the words by length. The library method `groupBy` takes a function that computes a "grouping key" from each element of a sequence. To group by word length (computed via the method `length`), we write:

```
words.groupBy { word => word.length }
```

or, more concisely, `words.groupBy(_.length)`. The result of this expression is a dictionary that maps each length to the list of words having that length:

```
scala> words.groupBy(_.length)
res0: Map[Int,Seq[String]] = Map(2 -> List(is), 4 -> List(food, good), 3 -> List(the))
```

This is close to what we need. If we convert this dictionary to a sequence, we will get a list of pairs:

```
scala> words.groupBy(_.length).toSeq
res1: Seq[(Int, Seq[String])] = ArrayBuffer((2,List(is)), (4,List(food, good)), (3,List(the)))
```

It remains to swap the length and the list of words and to sort the result by increasing length. We can do this in any order: first sort, then swap; or first swap, then sort. The final code is:

```
words
  .groupBy(_.length)
  .toSeq
  .sortBy { case (len, words) => len }
  .map { case (len, words) => (words, len) }
```

This can be written somewhat shorter if we use the code `_._1` (equivalent to `x => x._1`) for selecting the first parts from pairs and `swap` for swapping the two elements of a pair:

```
words.groupBy(_.length).toSeq.sortBy(_._1).map(_.swap)
```

However, the program may now be harder to read and to modify.

2.1.6 Reasoning about type parameters in collections

In Example 2.1.5.10 we have applied a chain of operations to a sequence. Let us add comments showing the type of the intermediate result after each operation:

```
words // Seq[String]
  .groupBy(_.length)                          // Map[Int, Seq[String]]
  .toSeq                                      // Seq[ (Int, Seq[String]) ]
  .sortBy { case (len, words) => len }        // Seq[ (Int, Seq[String]) ]
  .map { case (len, words) => (words, len) }  // Seq[ (Seq[String], Int) ]
```

In computations like this, the Scala compiler verifies at each step that the operations are applied to values of the correct types. Writing down the intermediate types will help us write correct code.

For instance, `sortBy` is defined for sequences but not for dictionaries, so it would be a type error to apply `sortBy` to a dictionary without first converting it to a sequence using `toSeq`. The type of the intermediate result after `toSeq` is `Seq[ (Int, Seq[String]) ]`, and the `sortBy` operation is applied to that sequence. So the sequence element matched by `{ case (len, words) => len }` is a tuple `(Int, Seq[String])`, which means that the pattern variables `len` and `words` must have types `Int` and `Seq[String]` respectively. It would be a type error to use the sorting key function `{ case (len, words) => words }`: the sorting key can be an integer `len`, but not a string sequence `words` (because sorting by string sequences is not automatically defined).

If we visualize how the type of the sequence should change at every step, we can more quickly understand how to implement the required task. Begin by writing down the intermediate types that would be needed during the computation:

```
words: Seq[String]           // Need to group by word length. The result of groupBy() has type:
  Map[Int, Seq[String]]      // Need to sort by word length; cannot sort a dictionary!
                             // Need to convert this dictionary to a sequence:
  Seq[ (Int, Seq[String]) ]  // Now sort this by the ‘Int‘ value. Sorting does not change the types.
                             // It remains to swap the parts of all tuples in the sequence:
  Seq[ (Seq[String], Int) ]  // We are done.
```

Having written down these types, we are better assured that the computation can be done correctly. Writing the code becomes straightforward, since we are guided by the already known types of the intermediate results:

```
words.groupBy(_.length).toSeq.sortBy(_._1).map(_.swap)
```

This example illustrates the main benefits of reasoning about types: it gives direct guidance about how to organize the computation, together with a greater confidence about code correctness.

2.1.7 Exercises: Tuples and collections

Exercise 2.1.7.1 Find all pairs i, j within $(0, 1, ..., 9)$ such that $i + 4 * j > i * j$.

Hint: use `flatMap` and `filter`.

Exercise 2.1.7.2 Same task as in Exercise 2.1.7.1, but for i, j, k and the condition $i + 4 * j + 9 * k > i * j * k$.

Exercise 2.1.7.3 Given two sequences `p: Seq[String]` and `q: Seq[Boolean]` of equal length, compute a `Seq[String]` with those elements of `p` for which the corresponding element of `q` is `true`.

Hint: use `zip`, `map`, `filter`.

Exercise 2.1.7.4 Convert a `Seq[Int]` into a `Seq[(Int, Boolean)]` where the `Boolean` value is `true` when the element is followed by a larger value. For example, the input sequence `Seq(1,3,2,4)` is to be converted into `Seq((1,true),(3,false),(2,true),(4,false))`. (The last element, 4, has no following element.)

Exercise 2.1.7.5 Given `p: Seq[String]` and `q: Seq[Int]` of equal length, compute a `Seq[String]` that contains the strings from `p` ordered according to the corresponding numbers from `q`. For example, if `p = Seq("a", "b", "c")` and `q = Seq(10, -1, 5)` then the result must be `Seq("b", "c", "a")`.

Exercise 2.1.7.6 Write the solution of Exercise 2.1.7.5 as a function with type parameter `A` instead of the fixed type `String`. The required type signature and a sample test:

```
def reorder[A](p: Seq[A], q: Seq[Int]): Seq[A] = ???    // In Scala, ??? means "not yet implemented".

scala> reorder(Seq(6.0,2.0,8.0,4.0), Seq(20,10,40,30))  // Test with type parameter A = Double.
res0: Seq[Double] = List(2.0, 6.0, 4.0, 8.0)
```

Exercise 2.1.7.7 Given `p:Seq[String]` and `q:Seq[Int]` of *equal* length and assuming that values in `q` do not repeat, compute a `Map[Int, String]` mapping numbers from `q` to the corresponding strings from `p`.

Exercise 2.1.7.8 Write the solution of Exercise 2.1.7.7 as a function with type parameters `P` and `Q` instead of the fixed types `String` and `Int`. The return type of the function should be `Map[Q, P]`. Run some tests using types `P = Double` and `Q = Set[Boolean]`.

Exercise 2.1.7.9 Given a `Seq[(String, Int)]` showing a list of purchased items (where item names may repeat), compute a `Map[String, Int]` showing the total counts. So, for the input:

```
Seq(("apple", 2), ("pear", 3), ("apple", 5), ("lemon", 2), ("apple", 3))
```

the output must be `Map("apple" -> 10, "pear" -> 3, "lemon" -> 2)`.

Hint: use `groupBy`, `map`, `sum`.

Exercise 2.1.7.10 Given a `Seq[List[Int]]`, compute a new `Seq[List[Int]]` where each new inner list contains 3 largest elements from the old inner list (or fewer than 3 if the old inner list is shorter).

Hint: use `map`, `sortBy`, `take`.

Exercise 2.1.7.11 **(a)** Given two sets, `p: Set[Int]` and `q: Set[Int]`, compute a set of type `Set[(Int, Int)]` as the **Cartesian product** of the sets `p` and `q`. This is the set of all pairs `(x, y)` where `x` is an element from the set `p` and `y` is an element from the set `q`.

(b) Implement this computation as a function with type parameters `I`, `J` instead of `Int`. The required type signature and a sample test:

```
def cartesian[I, J](p: Set[I], q: Set[J]): Set[(I, J)] = ???

scala> cartesian(Set("a", "b"), Set(10, 20))
res0: Set[(String, Int)] = Set((a,10), (a,20), (b,10), (b,20))
```

Hint: use `flatMap` and `map` on sets.

Exercise 2.1.7.12* Given a `Seq[Map[Person, Amount]]`, showing the amounts various people paid on each day, compute a `Map[Person, Seq[Amount]]`, showing the sequence of payments for each person. Assume that `Person` and `Amount` are type parameters. The required type signature and a sample test:

```
def payments[Person, Amount](data: Seq[Map[Person, Amount]]): Map[Person, Seq[Amount]] = ???
// On day 1, Tarski paid 10 and Gödel paid 20. On day 2, Church paid 100 and Gentzen paid 50, etc.
scala> payments(Seq(Map("Tarski" -> 10, "Gödel" -> 20), Map("Church" -> 100, "Gentzen" -> 50),
     Map("Tarski" -> 50), Map("Banach" -> 15, "Gentzen" -> 35)))
res0: Map[String, Seq[Int]] = Map(Genzten -> List(50, 35), Church -> List(100), Banach -> List(15),
     Tarski -> List(10, 50), Gödel -> List(20))
```

Hint: use `flatMap`, `groupBy`, `mapValues` on dictionaries.

2.2 Converting a sequence into a single value

Until this point, we have been working with sequences using methods such as `map` and `zip`. These techniques are powerful but still insufficient for solving certain problems.

A simple computation that is impossible to do using `map` is obtaining the sum of a sequence of numbers. The standard library method `sum` already does this; but we cannot re-implement `sum` ourselves by using `map`, `zip`, or `filter`. These operations always compute *new sequences*, while we need to compute a single value (the sum of all elements) from a sequence.

We have seen a few library methods such as `count`, `length`, and `max` that compute a single value from a sequence; but we still cannot implement `sum` using these methods. What we need is a more general way of converting a sequence to a single value, such that we could ourselves implement `sum`, `count`, `max`, and other similar computations.

Another task not solvable with `map`, `sum`, etc., is to compute a floating-point number from a given sequence of decimal digits (including a "dot" character):

```
def digitsToDouble(ds: Seq[Char]): Double = ???

scala> digitsToDouble(Seq('2', '0', '4', '.', '5'))
res0: Double = 204.5
```

Why is it impossible to implement this function by using `map`, `sum`, `zip`, and other methods we have seen so far? In fact, the same task for integer numbers (instead of floating-point numbers) *can* be implemented via `length`, `map`, `sum`, and `zip`:

```
def digitsToInt(ds: Seq[Int]): Int = {
  val n = ds.length
  // Compute a sequence of powers of 10, e.g., [1000, 100, 10, 1].
  val powers: Seq[Int] = (0 to n - 1).map(k => math.pow(10, n - 1 - k).toInt)
  // Sum the powers of 10 with coefficients from ‘ds‘.
  (ds zip powers).map { case (d, p) => d * p }.sum
}

scala> digitsToInt(Seq(2,4,0,5))
res0: Int = 2405
```

For this task, the required computation can be written as the formula:

$$r = \sum_{k=0}^{n-1} d_k * 10^{n-1-k} \quad .$$

The sequence of powers of 10 can be computed separately and "zipped" with the sequence of digits d_k. However, for floating-point numbers, the sequence of powers of 10 depends on the position of the "dot" character. Methods such as `map` or `zip` cannot compute a sequence whose next elements depend on previous elements and the dependence is described by some custom function.

2.2.1 Inductive definitions of aggregation functions

Mathematical induction is a general way of expressing the dependence of next values on previously computed values. To define a function from a sequence to a single value (e.g., an aggregation function `f: Seq[Int] => Int`) via mathematical induction, we need to specify two computations:

- The **base case** of the induction: We need to specify what value the function `f` returns for an empty sequence, `Seq()`. The standard method `isEmpty` can be used to detect empty sequences. In case the function `f` is only defined for non-empty sequences, we need to specify what the function `f` returns for a one-element sequence such as `Seq(x)`, with any `x`.

- The **inductive step**: Assuming that the function `f` is already computed for some sequence `xs` (the **inductive assumption**), how to compute the function `f` for a sequence with one more element `x`? The sequence with one more element is written as `xs :+ x`. So, we need to specify how to compute `f(xs :+ x)` assuming that `f(xs)` is already known.

Once these two computations are specified, the function `f` is defined (and can in principle be computed) for an arbitrary input sequence.

With this approach, the inductive definition of the method `sum` looks like this: The base case is that the sum of an empty sequence is 0. That is, `Seq().sum == 0`. The inductive step says that when the result `xs.sum` is already known for a sequence `xs`, and we have a sequence that has one more element `x`, then the new result is equal to `xs.sum + x`. In code, this is `(xs :+ x).sum == xs.sum + x`.

The inductive definition of the function `digitsToInt` goes like this: The base case is an empty sequence of digits, `Seq()`, and the result is `0`. This is a convenient base case even if we never need to apply `digitsToInt` to an empty sequence. The inductive step: If `digitsToInt(xs)` is already known for a sequence `xs` of digits, and we have a sequence `xs :+ x` with one more digit `x`, then:

```
digitsToInt(xs :+ x) == digitsToInt(xs) * 10 + x
```

Let us write inductive definitions for the methods `length`, `max`, and `count`.

The method `length` Base case: The length of an empty sequence is zero, so `Seq().length == 0`.

Inductive step: if `xs.length` is known then `(x +: xs).length == xs.length + 1`.

The method `max` The maximum element of a sequence is undefined for empty sequences.

Base case: for a one-element sequence, `Seq(x).max == x`.

Inductive step: if `xs.max` is known then `(x +: xs).max == math.max(x, xs.max)`.

The method `count` computes the number of a sequence's elements satisfying a predicate `p`.

Base case: for an empty sequence, `Seq().count(p) == 0`.

Inductive step: if `xs.count(p)` is known then `(x +: xs).count(p) == xs.count(p) + c`, where we define `c = 1` when `p(x) == true` and `c = 0` otherwise.

When a function is defined by induction, proving a property of that function will usually involve a "proof by induction". As an example, let us prove that `(xs ++ ys).length = xs.length + ys.length`. We use induction on the length of the sequence `xs`. In the base case, we need to prove that the property holds for the base case of the function's definition. In the base case, we need to prove that the property holds for an empty sequence `xs` (and an arbitrary sequence `ys`). To verify the base case, we write: `(Seq() ++ ys).length == ys.length`. In the inductive step of the proof, we assume that the property already holds for some `xs` and `ys` and prove that the property will then hold for `x +: xs` instead of `xs`. To verify that, we use the associativity law of the concatenation operation (to be proved in Statement 8.5.2.1), which allows us to write: `(x +: xs) ++ ys == x +: (xs ++ ys)`. Then we get:

```
((x +: xs) ++ ys).length == (x +: (xs ++ ys)).length == 1 + (xs.length + ys.length)
  == (x +: xs).length + ys.length   // Right-hand side of the property for (x +: xs)
```

In this way, we show that the property holds for `x +: xs` and `ys` assuming it holds for `xs` and `ys`.

There are two main ways of translating mathematical induction into code. The first way is to write a recursive function. The second way is to use a standard library function, such as `foldLeft` or `reduce`. Most often it is better to use the standard library functions, but sometimes the code is more transparent when using explicit recursion. So, let us consider each of these ways in turn.

2.2.2 Implementing functions by recursion

A **recursive function** is any function that calls itself somewhere within its own body. The call to itself is the **recursive call**. Recursion may be used to implement functions defined by induction.

When the body of a recursive function is evaluated, it may repeatedly call itself with different arguments until a result value can be computed *without* any recursive calls. The repeated recursive calls correspond to inductive steps, and the last call corresponds to the base case of the inductive definition. It is an error (an infinite loop) if the base case is never reached, as in this example:

```
scala> def infiniteLoop(x: Int): Int = infiniteLoop(x+1)
infiniteLoop: (x: Int)Int

scala> infiniteLoop(2) // You will need to press Ctrl-C to stop this.
```

We translate mathematical induction into code by first writing a condition to decide whether we have the base case or the inductive step. As an example, let us define `sum` by recursion. The base case returns `0`, while the inductive step returns a value computed from the recursive call. In this example, the `if`/`else` expression will separate the base case from the inductive step. In the inductive step, it is convenient to split the given sequence `s` into its first element `x`, or the "head" of `s`, and the remainder ("tail") sequence `xs`. So, we split `s` as `s = x +: xs` rather than as `s = xs :+ x`.[1]

```
def sum(s: Seq[Int]): Int = if (s.isEmpty) 0 else {
  val x = s.head  // To split s = x +: xs, compute x
  val xs = s.tail // and xs.
  sum(xs) + x     // Call sum(...) recursively.
}
```

For computing the sum of a numerical sequence, the order of summation does not matter. However, the order of operations *will* matter for many other computational tasks. We need to choose whether the inductive step should split the sequence as `s = x +: xs` or as `s = xs :+ x`, depending on the task at hand.

Let us implement `digitsToInt` according to the inductive definition shown in Section 2.2.1:

```
def digitsToInt(s: Seq[Int]): Int = if (s.isEmpty) 0 else {
  val x = s.last              // To split s = xs :+ x, compute x
  val xs = s.take(s.length - 1)  // and xs.
  digitsToInt(xs) * 10 + x // Call digitsToInt(...) recursively.
}
```

In this example, it is important to split the sequence `s` into `xs :+ x` and not into `x +: xs`. The reason is that digits increase their numerical value from right to left, so the correct result is computed if we split `s` into `xs :+ x` and multiply `digitsToInt(xs)` by 10.

These examples show how mathematical induction is converted into recursive code. This approach often works but has two technical problems. The first problem is that the code will fail due to a stack overflow when the input sequence `s` is long enough. In the next subsection, we will see how this problem is solved (at least in some cases) using tail recursion.

The second problem is that all inductively defined functions will use the same code for checking the base case and for splitting the sequence `s` into the subsequence `xs` and the extra element `x`. This repeated common code can be put into a library function, and the Scala library provides such functions. We will look at using them in Section 2.2.4.

2.2.3 Tail recursion

The code of `lengthS` will fail for large enough sequences. To see why, consider an inductive definition of the `length` method as a function `lengthS`:

```
def lengthS(s: Seq[Int]): Int =
  if (s.isEmpty) 0
  else 1 + lengthS(s.tail)

scala> lengthS((1 to 1000).toList)
res0: Int = 1000
```

[1] It is easier to remember the meaning of `x +: xs` and `xs :+ x` if we note that the *col*on (`:`) always points to the *col*lection (`xs`) and the plus sign (`+`) to a single element (`x`) that is being added.

```
scala> val s = (1 to 100000).toList
s: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
     23, 24, 25, 26, 27, 28, 29, 30, 31, 32, ...

scala> lengthS(s)
java.lang.StackOverflowError
  at .lengthS(<console>:12)
  at .lengthS(<console>:12)
  at .lengthS(<console>:12)
...
```

The problem is not due to insufficient main memory: we *are* able to compute and hold in memory the entire sequence `s`. The problem is with the code of the function `lengthS`. This function calls itself *inside* the expression `1 + lengthS(...)`. So, we can visualize how the computer evaluates this code:

```
lengthS(Seq(1, 2, ..., 100000))
  = 1 + lengthS(Seq(2, ..., 100000))
  = 1 + (1 + lengthS(Seq(3, ..., 100000)))
  = ...
```

The code of `lengthS` will repeat the inductive step, that is, the "`else`" part of the "`if/else`", about 100,000 times. Each time, the sub-expression with nested computations `1+(1+(...))` will get larger. This intermediate sub-expression needs to be held somewhere in memory, until at some point the function body goes into the base case and returns a value. When that happens, the entire intermediate sub-expression will contain about 100,000 nested function calls still waiting to be evaluated. This sub-expression is held in a special area of memory called **stack memory**, where the not-yet-evaluated nested function calls are held in the order of their calls on a stack. Due to the way computer memory is managed, the stack memory has a fixed size and cannot grow automatically. So, when the intermediate expression becomes large enough, it causes an overflow of the stack memory and crashes the program.

One way to avoid stack overflows is to use a trick called **tail recursion**. Using tail recursion means rewriting the code so that all recursive calls occur at the end positions (at the "tails") of the function body. In other words, each recursive call must be *itself* the last computation in the function body, rather than placed inside other computations. Here is an example of tail-recursive code:

```
def lengthT(s: Seq[Int], res: Int): Int =
  if (s.isEmpty) res
  else lengthT(s.tail, res + 1)
```

In this code, one of the branches of the `if/else` returns a fixed value without doing any recursive calls, while the other branch returns the result of a recursive call to `lengthT(...)`. In the code of `lengthT`, recursive calls never occur within any sub-expressions.

It is not a problem that the recursive call to `lengthT` has some sub-expressions such as `res + 1` as its arguments, because all these sub-expressions will be computed *before* `lengthT` is recursively called. The recursive call to `lengthT` is the *last* computation performed by this branch of the `if/else`. A tail-recursive function can have many `if/else` or `match/case` branches, with or without recursive calls; but all recursive calls must be always the last expressions returned.

The Scala compiler will always use tail recursion when possible. Additionally, Scala has a feature for verifying that a function's code is tail-recursive: the `tailrec` annotation. If a function with a `tailrec` annotation is not tail-recursive (or is not recursive at all), the program will not compile.

The code of `lengthT` with a `tailrec` annotation looks like this:

```
@tailrec def lengthT(s: Seq[Int], res: Int): Int =
  if (s.isEmpty) res
  else lengthT(s.tail, res + 1)
```

Let us trace the evaluation of this function on an example:

```
lengthT(Seq(1, 2, 3), 0)
  = lengthT(Seq(2, 3), 0 + 1) // = lengthT(Seq(2, 3), 1)
  = lengthT(Seq(3), 1 + 1)    // = lengthT(Seq(3), 2)
  = lengthT(Seq(), 2 + 1)     // = lengthT(Seq(), 3)
  = 3
```

All sub-expressions such as `1 + 1` and `2 + 1` are computed *before* recursive calls to `lengthT`. Because of that, sub-expressions do not grow within the stack memory. This is the main benefit of tail recursion.

How did we rewrite the code of `lengthS` into the tail-recursive code of `lengthT`? An important difference between `lengthS` and `lengthT` is the additional argument, `res`, called the **accumulator argument**.

This argument is equal to an intermediate result of the computation. The next intermediate result (`res + 1`) is computed and passed on to the next recursive call via the accumulator argument. In the base case of the recursion, the function now returns the accumulated result, `res`, rather than `0`, because at that time the computation is finished.

Rewriting code by adding an accumulator argument to achieve tail recursion is called the **accumulator technique** or the "accumulator trick".

One consequence of using the accumulator trick is that the function `lengthT` now always needs a value for the accumulator argument. However, our goal is to implement a function such as `length(s)` with just one argument, `s: Seq[Int]`. We can define `length(s) = lengthT(s, ???)` if we supply an initial accumulator value. The correct initial value for the accumulator is `0`, since in the base case (an empty sequence `s`) we need to return `0`.

So, a tail-recursive implementation of `lengthT` requires us to define *two* functions: the tail-recursive `lengthT` and the main function that will set the initial value of the accumulator argument. To emphasize that `lengthT` is a helper function, one could define it *inside* the main function:

```
def length[A](xs: Seq[A]): Int = {
  @tailrec def lengthT(s: Seq[A], res: Int): Int = {
    if (s.isEmpty) res
    else lengthT(s.tail, res + 1)
  }
  lengthT(xs, 0)
}
```

When `length` is implemented like that, users will not be able to call `lengthT` directly, because `lengthT` is only visible within the body of the `length` function.

Another possibility in Scala is to use a **default value** for the `res` argument:

```
@tailrec def length[A](s: Seq[A], res: Int = 0): Int =
  if (s.isEmpty) res
  else length(s.tail, res + 1)
```

Giving a default value for a function argument is the same as defining *two* functions: one with that argument and one without. For example, the syntax:

```
def f(x: Int, y: Boolean = false): Int = ... // Function body.
```

is equivalent to defining two functions (with the same name):

```
def f(x: Int, y: Boolean) = ...      // Define the function body here.
def f(x: Int): Int = f(Int, false)   // Call the function defined above.
```

Using a default argument, we can define the tail-recursive helper function and the main function at once, making the code shorter.

The accumulator trick works in a large number of cases, but it may be far from obvious how to introduce the accumulator argument, what its initial value must be, and how to define the inductive step for the accumulator. In the example with the `lengthT` function, the accumulator trick works because of the special mathematical property of the expression being computed:

$$1 + (1 + (1 + (\ldots + 0))) = (((0 + 1) + 1) + \ldots) + 1 \quad .$$

This equation follows from the **associativity law** of addition. So, the computation can be rearranged to group all additions to the left. In code, it means that intermediate expressions are computed immediately before making recursive calls; this avoids the growth of the intermediate expressions.

Usually, the accumulator trick works because some associativity law is present. In that case, we are able to rearrange the order of recursive calls so that these calls always occur outside all other sub-expressions — that is, in tail positions. However, not all computations obey a suitable associativity law. Even if a code rearrangement exists, it may not be immediately obvious how to find it.

As an example, let us implement a tail-recursive version of the function `digitsToInt` from the previous subsection, where the recursive call is within a sub-expression `digitsToInt(xs) * 10 + x`. To

transform the code into a tail-recursive form, we need to rearrange the main computation:

$$r = d_{n-1} + 10 * (d_{n-2} + 10 * (d_{n-3} + 10 * (... + 10 * d_0))) \quad ,$$

so that the operations group to the left. We can do this by rewriting r as:

$$r = ((d_0 * 10 + d_1) * 10 + ...) * 10 + d_{n-1} \quad .$$

It follows that the digit sequence `s` must be split into the *leftmost* digit and the rest, `s == s.head +: s.tail`. So, a tail-recursive implementation of the above formula is:

```
@tailrec def fromDigits(s: Seq[Int], res: Int = 0):Int =
  // ‘res‘ is the accumulator.
  if (s.isEmpty) res
  else fromDigits(s.tail, 10 * res + s.head)

scala> fromDigits(Seq(1, 2, 3, 4))
res0: Int = 1234
```

Despite a similarity between this code and the code of `digitsToInt` from the previous subsection, the implementation of `fromDigits` cannot be directly derived from the inductive definition of `digitsToInt`. We need a separate proof that `fromDigits(s, 0)` computes the same result as `digitsToInt(s)`. This can be proved by using the following property:

Statement 2.2.3.1 For any `s: Seq[Int]` and `r: Int`, the following equation holds:

```
fromDigits(s, r) == digitsToInt(s) + r * math.pow(10, s.length)
```

Proof We use induction on the length of `s`. To shorten the proof, denote sequences by $[1,2,3]$ instead of `Seq(1, 2, 3)` and temporarily write $d(s)$ instead of `digitsToInt(s)` and $f(s,r)$ instead of `fromDigitsT(s, r)`. Then an inductive definition of $f(s,r)$ is:

$$f([],r) = r \quad , \qquad f([x]{+}{+}s,r) = f(s, 10 * r + x) \quad . \tag{2.1}$$

Denoting the length of a sequence s by $|s|$, we reformulate Statement 2.2.3.1 as:

$$f(s,r) = d(s) + r * 10^{|s|} \quad . \tag{2.2}$$

We prove Eq. (2.2) by induction. For the base case $s = []$, we have $f([],r) = r$ and $d([]) + r * 10^0 = r$ since $d([]) = 0$ and $|s| = 0$. The resulting equality $r = r$ proves the base case.

To prove the inductive step, we assume that Eq. (2.2) holds for a given sequence s. Then write the inductive step (we will use the symbol $\stackrel{?}{=}$ to denote equations we still need to prove):

$$f([x]{+}{+}s,r) \stackrel{?}{=} d([x]{+}{+}s) + r * 10^{|s|+1} \quad . \tag{2.3}$$

We will transform the left-hand side and the right-hand side separately, hoping to obtain the same expression. The left-hand side of Eq. (2.3):

$$\begin{aligned} & f([x]{+}{+}s,r) \\ \text{use Eq. (2.1)}: \quad & = f(s, 10 * r + x) \\ \text{use Eq. (2.2)}: \quad & = d(s) + (10 * r + x) * 10^{|s|} \quad . \end{aligned}$$

The right-hand side of Eq. (2.3) contains $d([x]{+}{+}s)$, which we somehow need to simplify. Assuming that $d(s)$ correctly calculates a number from its digits, we use a property of decimal notation: a digit x in front of n other digits has the value $x * 10^n$. This property can be formulated as an equation:

$$d([x]{+}{+}s) = x * 10^{|s|} + d(s) \quad . \tag{2.4}$$

So, the right-hand side of Eq. (2.3) can be rewritten as:

$$\begin{aligned} & d([x]{+}{+}s) + r * 10^{|s|+1} \\ \text{use Eq. (2.4)}: \quad & = x * 10^{|s|} + d(s) + r * 10^{|s|+1} \\ \text{factor out } 10^{|s|}: \quad & = d(s) + (10 * r + x) * 10^{|s|} \quad . \end{aligned}$$

We have successfully transformed both sides of Eq. (2.3) to the same expression.

We have not yet proved that the function d satisfies the property in Eq. (2.4). The proof also uses induction. Begin by writing the code of d in a short notation:

$$d([]) = 0 \quad , \qquad d(s{+}{+}[y]) = d(s) * 10 + y \quad . \tag{2.5}$$

The base case is Eq. (2.4) with $s = []$. It is proved by:

$$x = d([]{+}{+}[x]) = d([x]{+}{+}[]) = x * 10^0 + d([]) = x \quad .$$

The inductive step assumes Eq. (2.4) for a given x and a given sequence s, and needs to prove that for any y, the same property holds with $s{+}{+}[y]$ instead of s:

$$d([x]{+}{+}s{+}{+}[y]) \stackrel{?}{=} x * 10^{|s|+1} + d(s{+}{+}[y]) \quad . \tag{2.6}$$

The left-hand side of Eq. (2.6) is transformed into its right-hand side like this:

$$\begin{aligned}
& d([x]{+}{+}s{+}{+}[y]) \\
\text{use Eq. (2.5)}: \quad & = d([x]{+}{+}s) * 10 + y \\
\text{use Eq. (2.4)}: \quad & = (x * 10^{|s|} + d(s)) * 10 + y \\
\text{expand parentheses}: \quad & = x * 10^{|s|+1} + d(s) * 10 + y \\
\text{use Eq. (2.5)}: \quad & = x * 10^{|s|+1} + d(s{+}{+}[y]) \quad .
\end{aligned}$$

This demonstrates Eq. (2.6) and concludes the proof.

2.2.4 Implementing general aggregation (`foldLeft`)

An **aggregation** converts a sequence of values into a single value. In general, the type of the result may be different from the type of sequence elements. To describe that general situation, we introduce type parameters, `A` and `B`, so that the input sequence is of type `Seq[A]` and the aggregated value is of type `B`. Then an inductive definition of any aggregation function `f: Seq[A] => B` looks like this:

- (Base case.) For an empty sequence, we have `f(Seq()) = b0`, where `b0: B` is a given value.
- (Inductive step.) Assuming that `f(xs) = b` is already computed, we define `f(xs :+ x) = g(x, b)` where `g` is a given function with type signature `g: (A, B) => B`.

The code implementing `f` is written using recursion:

```
def f[A, B](s: Seq[A]): B =
  if (s.isEmpty) b0
  else g(s.last, f(s.take(s.length - 1)))
```

We can now refactor this code into a generic utility function, by making `b0` and `g` into parameters. A possible implementation is:

```
def f[A, B](s: Seq[A], b: B, g: (A, B) => B): B =
  if (s.isEmpty) b
  else g(s.last, f(s.take(s.length - 1), b, g)
```

However, this implementation is not tail-recursive. Applying `f` to a sequence of, say, three elements, `Seq(x, y, z)`, will create an intermediate expression `g(z, g(y, g(x, b)))`. This expression will grow with the length of `s`, which is not acceptable. To rearrange the computation into a tail-recursive form, we need to start the base case at the innermost call `g(x, b)`, then compute `g(y, g(x, b))` and continue. In other words, we need to traverse the sequence starting from its *leftmost* element `x`, rather than starting from the right. So, instead of splitting the sequence `s` into `s.take(s.length - 1) :+ s.last` as we did in the code of `f`, we need to split `s` into `s.head +: s.tail`. Let us also exchange the order of the arguments of `g`, in order to be more consistent with the way this code is implemented in the Scala library. The resulting code is tail-recursive:

```
@tailrec def leftFold[A, B](s: Seq[A], b: B, g: (B, A) => B): B =
  if (s.isEmpty) b
  else leftFold(s.tail, g(b, s.head), g)
```

We call this function a "left fold" because it aggregates (or "folds") the sequence starting from the leftmost element.

In this way, we have defined a general method of computing any inductively defined aggregation function on a sequence. The function `leftFold` implements the logic of aggregation defined via mathematical induction. Using `leftFold`, we can write concise implementations of methods such as `sum`, `max`, and many other aggregation functions. The method `leftFold` already contains all the code necessary to set up the base case and the inductive step. The programmer just needs to specify the expressions for the initial value `b` and for the updater function `g`.

As a first example, let us use `leftFold` for implementing the `sum` method:

```
def sum(s: Seq[Int]): Int = leftFold(s, 0, (x, y) => x + y )
```

To understand in detail how `leftFold` works, let us trace the evaluation of this function when applied to `Seq(1, 2, 3)`:

```
sum(Seq(1, 2, 3)) == leftFold(Seq(1, 2, 3), 0, g)
                     // Here, g = (x, y) => x + y, so g(x, y) = x + y.
  == leftFold(Seq(2, 3), g(0, 1), g)                  // g (0, 1) = 1.
  == leftFold(Seq(2, 3), 1, g)   // Now expand the code of ‘leftFold‘.
  == leftFold(Seq(3), g(1, 2), g)     // g(1, 2) = 3; expand the code.
  == leftFold(Seq(), g(3, 3), g)      // g(3, 3) = 6; expand the code.
  == 6
```

The second argument of `leftFold` is the accumulator argument. The initial value of the accumulator is specified when first calling `leftFold`. At each iteration, the new accumulator value is computed by calling the updater function `g`, which uses the previous accumulator value and the value of the next sequence element. To visualize the process of recursive evaluation, it is convenient to write a table showing the sequence elements and the accumulator values as they are updated:

Current element `x`	**Old accumulator value**	**New accumulator value**
1	0	1
2	1	3
3	3	6

We implemented `leftFold` only as an illustration. Scala's library has a method called `foldLeft` implementing the same logic using a slightly different type signature. To see this difference, compare the implementation of `sum` using our `leftFold` function and using the standard `foldLeft` method:

```
def sum(s: Seq[Int]): Int = leftFold(s, 0, (x, y) => x + y )

def sum(s: Seq[Int]): Int = s.foldLeft(0) { (x, y) => x + y }
```

The syntax of `foldLeft` makes it more convenient to use a nameless function as the updater argument of `foldLeft`, since curly braces separate that argument from others. We will use the standard `foldLeft` method from now on.

In general, the type of the accumulator value can be different from the type of the sequence elements. An example is an implementation of `count`:

```
def count[A](s: Seq[A], p: A => Boolean): Int =
  s.foldLeft(0) { (x, y) => x + (if (p(y)) 1 else 0) }
```

The accumulator is of type `Int`, while the sequence elements can have an arbitrary type, parameterized by `A`. The `foldLeft` method works in the same way for all types of accumulators and all types of sequence elements.

Since `foldLeft` is tail-recursive, stack overflows will not occur even with long sequences. The method `foldLeft` is available in the Scala library for all collections, including dictionaries and sets.

It is important to gain experience using the `foldLeft` method. The Scala library contains several other methods similar to `foldLeft`, such as `foldRight`, `fold`, and `reduce`. In the following sections, we will mostly focus on `foldLeft` because the other fold-like operations are similar.

2.2.5 Solved examples: using `foldLeft`

Example 2.2.5.1 Use `foldLeft` for implementing the `max` function for integer sequences. Return the special value `Int.MinValue` for empty sequences.

Solution Begin by writing an inductive formulation of the `max` function for sequences. Base case: For an empty sequence, return `Int.MinValue`. Inductive step: If `max` is already computed on a sequence `xs`, say `max(xs) = b`, the value of `max` on a sequence `xs :+ x` is `math.max(b, x)`. So, the code is:

```
def max(s: Seq[Int]): Int = s.foldLeft(Int.MinValue) { (b, x) => math.max(b, x) }
```

If we are sure that the function will never be called on empty sequences, we can implement `max` in a simpler way by using the `reduce` method:

```
def max(s: Seq[Int]): Int = s.reduce { (x, y) => math.max(x, y) }
```

Example 2.2.5.2 For a given non-empty sequence `xs: Seq[Double]`, compute the minimum, the maximum, and the mean as a tuple $(x_{\text{min}}, x_{\text{max}}, x_{\text{mean}})$. The sequence should be traversed only once; i.e., the entire code must be `xs.foldLeft(...)`, using `foldLeft` only once.

Solution Without the requirement of using a single traversal, we would write

```
(xs.min, xs.max, xs.sum / xs.length)
```

However, this code traverses `xs` at least three times, since each of the aggregations `xs.min`, `xs.max`, and `xs.sum` iterates over `xs`. We need to combine the four inductive definitions of `min`, `max`, `sum`, and `length` into a single inductive definition of some function. What is the type of that function's return value? We need to accumulate intermediate values of *all four* numbers (`min`, `max`, `sum`, and `length`) in a tuple. So the required type of the accumulator is `(Double, Double, Double, Double)`. To avoid repeating a long type expression, we can define a type alias for it, say, `D4`:

```
scala> type D4 = (Double, Double, Double, Double)
defined type alias D4
```

The updater updates each of the four numbers according to the definitions of their inductive steps:

```
def update(p: D4, x: Double): D4 = p match { case (min, max, sum, length) =>
   (math.min(x, min), math.max(x, max), x + sum, length + 1)
}
```

Now we can write the code of the required function:

```
def f(xs: Seq[Double]): (Double, Double, Double) = {
  val init: D4 = (Double.PositiveInfinity, Double.NegativeInfinity, 0, 0)
  val (min, max, sum, length) = xs.foldLeft(init)(update)
  (min, max, sum/length)
}

scala> f(Seq(1.0, 1.5, 2.0, 2.5, 3.0))
res0: (Double, Double, Double) = (1.0,3.0,2.0)
```

Example 2.2.5.3 Implement the `map` method for sequences by using `foldLeft`. The input sequence should be of type `Seq[A]` and the output sequence of type `Seq[B]`, where `A` and `B` are type parameters. The required type signature of the function and a sample test:

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B] = ???

scala> map(List(1, 2, 3)) { x => x * 10 }
res0: Seq[Int] = List(10, 20, 30)
```

Solution The required code should build a new sequence by applying the function `f` to each element. How can we build a new sequence using `foldLeft`? The evaluation of `foldLeft` consists of iterating over the input sequence and accumulating some result value, which is updated at each iteration. Since the result of a `foldLeft` is always equal to the last computed accumulator value, it follows that the new sequence should *be* the accumulator value. So, we need to update the accumulator by appending the value `f(x)`, where `x` is the current element of the input sequence:

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B] =
  xs.foldLeft(Seq[B]()) { (acc, x) => acc :+ f(x) }
```

Example 2.2.5.4 Implement the function `digitsToInt` using `foldLeft`.

Solution The inductive definition of `digitsToInt` is directly translated into code:

```
def digitsToInt(d: Seq[Int]): Int =
  d.foldLeft(0){ (n, x) => n * 10 + x }
```

Example 2.2.5.5* Implement the function `digitsToDouble` using `foldLeft`. The argument is of type `Seq[Char]`. As a test, the expression `digitsToDouble(Seq('3','4','.','2','5'))` must evaluate to `34.25`. Assume that all input characters are either digits or a dot (so, negative numbers are not supported).

Solution The evaluation of a `foldLeft` on a sequence of digits will visit the sequence from left to right. The updating function should work the same as in `digitsToInt` until a dot character is found. After that, we need to change the updating function. So, we need to remember whether a dot character has been seen. The only way for `foldLeft` to "remember" any data is to hold that data in the accumulator value. We can choose the type of the accumulator according to our needs. So, for this task we can choose the accumulator to be a *tuple* that contains, for instance, the floating-point result constructed so far and a `Boolean` flag showing whether we have already seen the dot character.

To see what `digitsToDouble` must do, let us consider how the evaluation of `digitsToDouble(Seq('3', '4', '.', '2', '5'))` should go. We can write a table showing the intermediate result at each iteration. This will hopefully help us figure out what the accumulator and the updater function $g(...)$ must be:

Current digit c	**Previous result** n	**New result** $n' = g(n, c)$
`'3'`	0.0	3.0
`'4'`	3.0	34.0
`'.'`	34.0	34.0
`'2'`	34.0	34.2
`'5'`	34.2	34.25

While the dot character was not yet seen, the updater function multiplies the previous result by 10 and adds the current digit. After the dot character, the updater function must add to the previous result the current digit divided by a factor that represents increasing powers of 10. In other words, the update computation $n' = g(n, c)$ must be defined by:

$$g(n, c) = \begin{cases} n * 10 + c \quad , & \text{if the digit is before the dot character.} \\ n + c/f \quad , & \text{if after the dot character, where } f = 10, 100, 1000, ... \text{ for each new digit.} \end{cases}$$

The updater function g has only two arguments: the current digit and the previous accumulator value. So, the changing factor f must be *part of* the accumulator value, and must be multiplied by 10 at each digit after the dot. If the factor f is not a part of the accumulator value, the function g will not have enough information for computing the next accumulator value correctly. So, the updater computation must be $n' = g(n, c, f)$, not $n' = g(n, c)$.

For this reason, we choose the accumulator type as a tuple `(Double, Boolean, Double)` where the first number is the result n computed so far, the `Boolean` flag indicates whether the dot was already seen, and the third number is f, that is, the power of 10 by which the current digit will be divided if the

dot was already seen. Initially, the accumulator tuple will be equal to `(0.0, false, 10.0)`. Then the updater function is implemented like this:

```
def update(acc: (Double, Boolean, Double), c: Char): (Double, Boolean, Double) =
  acc match { case (num, flag, factor) =>
    if (c == '.') (num, true, factor) // Set flag to ‘true‘ after a dot character was seen.
    else {
      val digit = c - '0'
      if (flag) (num + digit / factor, flag, factor * 10) // This digit is after the dot.
      else (num * 10 + digit, flag, factor)               // This digit is before the dot.
    }
  }
```

Now we can implement `digitsToDouble` like this:

```
def digitsToDouble(d: Seq[Char]): Double = {
  val initAcc = (0.0, false, 10.0)
  val (num, _, _) = d.foldLeft(initAcc)(update)
  num
}

scala> digitsToDouble(Seq('3','4','.','2','5'))
res0: Double = 34.25
```

The result of calling `d.foldLeft` is a tuple `(num, flag, factor)`, in which only the first part, `num`, is needed. In Scala's pattern matching syntax, the underscore (`_`) denotes pattern variables whose values are not needed in the code. We could get the first part using the accessor method `._1`, but the code will be more readable if we show all parts of the tuple `(num, _, _)`.

Example 2.2.5.6 Implement a function `toPairs` that converts a sequence of type `Seq[A]` to a sequence of pairs, `Seq[(A, A)]`, by putting together the adjacent elements pairwise. If the initial sequence has an odd number of elements, a given default value of type `A` is used to fill the last pair. The required type signature and an example test:

```
def toPairs[A](xs: Seq[A], default: A): Seq[(A, A)] = ???

scala> toPairs(Seq(1, 2, 3, 4, 5, 6), -1)
res0: Seq[(Int, Int)] = List((1,2), (3,4), (5,6))

scala> toPairs(Seq("a", "b", "c"), "<nothing>")
res1: Seq[(String, String)] = List((a,b), (c,<nothing>))
```

Solution We need to accumulate a sequence of pairs, and each pair needs two values. However, we iterate over values in the input sequence one by one. So, a new pair can be made only once every two iterations. The accumulator needs to hold the information about the current iteration being even or odd. For odd-numbered iterations, the accumulator also needs to store the previous element that is still waiting for its pair. Therefore, we choose the type of the accumulator to be a tuple `(Seq[(A, A)], Seq(A))`. The first sequence is the intermediate result, and the second sequence is the "holdover": it holds the previous element for odd-numbered iterations and is empty for even-numbered iterations. Initially, the accumulator should be empty. An example evaluation is:

Current element `x`	Previous accumulator	Next accumulator
`"a"`	`(Seq(), Seq())`	`(Seq(), Seq("a"))`
`"b"`	`(Seq(), Seq("a"))`	`(Seq(("a","b")), Seq())`
`"c"`	`(Seq(("a","b")), Seq())`	`(Seq(("a","b")), Seq("c"))`

Now it becomes clear how to implement the updater function:

```
type Acc = (Seq[(A, A)], Seq[A])     // Type alias, for brevity.
def updater(acc: Acc, x: A): Acc = acc match {
    case (result, Seq())      => (result, Seq(x))
    case (result, Seq(prev))  => (result :+ ((prev, x)), Seq())
  }
```

We will call `foldLeft` with this updater and then perform some post-processing to make sure we create the last pair in case the last iteration is odd-numbered, i.e., when the "holdover" is not empty

after `foldLeft` is finished. In this implementation, we use pattern matching to decide whether a sequence is empty:

```
def toPairs[A](xs: Seq[A], default: A): Seq[(A, A)] = {
  type Acc = (Seq[(A, A)], Seq[A])     // Type alias, for brevity.
  def init: Acc = (Seq(), Seq())
  def updater(acc: Acc, x: A): Acc = acc match {
    case (result, Seq())      => (result, Seq(x))
    case (result, Seq(prev))  => (result :+ ((prev, x)), Seq())
  }
  val (result, holdover) = xs.foldLeft(init)(updater)
  holdover match {    // May need to append the last element to the result.
    case Seq()    => result
    case Seq(x)   => result :+ ((x, default))
  }
}
```

This code shows examples of partial functions that are applied safely. One of these partial functions is used in the expression

```
holdover match {
  case Seq()    => ...
  case Seq(a)   => ...
}
```

This code works when `holdover` is empty or has length 1 but fails for longer sequences. In the implementation of `toPairs`, the value of `holdover` will always be a sequence of length at most 1, so it is safe to use this partial function.

2.2.6 Exercises: Using `foldLeft`

Exercise 2.2.6.1 Implement a function `fromPairs` that performs the inverse transformation to the `toPairs` function defined in Example 2.2.5.6. The required type signature and a sample test are:

```
def fromPairs[A](xs: Seq[(A, A)]): Seq[A] = ???

scala> fromPairs(Seq((1, 2), (3, 4)))
res0: Seq[Int] = List(1, 2, 3, 4)
```

Hint: This can be done with `foldLeft` or with `flatMap`.

Exercise 2.2.6.2 Implement the `flatten` method for sequences by using `foldLeft`. The required type signature and a sample test are:

```
def flatten[A](xxs: Seq[Seq[A]]): Seq[A] = ???

scala> flatten(Seq(Seq(1, 2, 3), Seq(), Seq(4)))
res0: Seq[Int] = List(1, 2, 3, 4)
```

Exercise 2.2.6.3 Use `foldLeft` to implement the `zipWithIndex` method for sequences. The required type signature and a sample test:

```
def zipWithIndex[A](xs: Seq[A]): Seq[(A, Int)] = ???

scala> zipWithIndex(Seq("a", "b", "c", "d"))
res0: Seq[String] = List((a, 0), (b, 1), (c, 2), (d, 3))
```

Exercise 2.2.6.4 Use `foldLeft` to implement a function `filterMap` that combines `map` and `filter` for sequences. The required type signature and a sample test:

```
def filterMap[A, B](xs: Seq[A])(pred: A => Boolean)(f: A => B): Seq[B] = ???

scala> filterMap(Seq(1, 2, 3, 4)) { x => x > 2 } { x => x * 10 }
res0: Seq[Int] = List(30, 40)
```

Exercise 2.2.6.5* Split a sequence into subsequences ("batches") of length not larger than a given maximum length n. The required type signature and a sample test:

```
def byLength[A](xs: Seq[A], length: Int): Seq[Seq[A]] = ???
```

```
scala> byLength(Seq("a", "b", "c", "d"), 2)
res0: Seq[Seq[String]] = List(List(a, b), List(c, d))

scala> byLength(Seq(1, 2, 3, 4, 5, 6, 7), 3)
res1: Seq[Seq[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7))
```

Exercise 2.2.6.6* Split a sequence into batches by "weight" computed via a given function. The total weight of items in any batch should not be larger than a given maximum weight. The required type signature and a sample test:

```
def byWeight[A](xs: Seq[A], maxW: Double)(w: A => Double): Seq[Seq[A]] = ???

scala> byWeight((1 to 10).toList, 5.75){ x => math.sqrt(x) }
res0: Seq[Seq[Int]] = List(List(1, 2, 3), List(4, 5), List(6, 7), List(8), List(9), List(10))
```

Exercise 2.2.6.7* Use `foldLeft` to implement a `groupBy` function. The type signature and a test:

```
def groupBy[A, K](xs: Seq[A])(by: A => K): Map[K, Seq[A]] = ???

scala> groupBy(Seq(1, 2, 3, 4, 5)){ x => x % 2 }
res0: Map[Int, Seq[Int]] = Map(1 -> List(1, 3, 5), 0 -> List(2, 4))
```

Hints: The accumulator should be of type `Map[K, Seq[A]]`. To work with dictionaries, you will need to use the methods `getOrElse` and `updated`. The method `getOrElse` fetches a value from a dictionary by key, and returns the given default value if the dictionary does not contain that key:

```
scala> Map("a" -> 1, "b" -> 2).getOrElse("a", 300)
res0: Int = 1

scala> Map("a" -> 1, "b" -> 2).getOrElse("c", 300)
res1: Int = 300
```

The method `updated` produces a new dictionary that contains a new value for the given key, whether or not that key already exists in the dictionary:

```
scala> Map("a" -> 1, "b" -> 2).updated("c", 300) // Key is new.
res0: Map[String,Int] = Map(a -> 1, b -> 2, c -> 300)

scala> Map("a" -> 1, "b" -> 2).updated("a", 400) // Key already exists.
res1: Map[String,Int] = Map(a -> 400, b -> 2)
```

2.3 Converting a single value into a sequence

An aggregation converts ("folds") a sequence into a single value; the opposite operation ("unfolding") converts a single value into a sequence. An example of this task is to compute the sequence of decimal digits for a given integer:

```
def digitsOf(x: Int): Seq[Int] = ???

scala> digitsOf(2405)
res0: Seq[Int] = List(2, 4, 0, 5)
```

We cannot implement `digitsOf` using `map`, `zip`, or `foldLeft`, because these methods work only if we *already have* a sequence; but the function `digitsOf` needs to create a new sequence. We could create a sequence via the expression `(1 to n)` if the required length of the sequence were known in advance. However, the function `digitsOf` must produce a sequence whose length is determined by a condition that we cannot easily evaluate in advance.

A general "unfolding" operation needs to build a sequence whose length is not determined in advance. This kind of sequence is called a **stream**. The elements of a stream are computed only when necessary (unlike the elements of `List` or `Array`, which are all computed in advance). The unfolding operation will compute next elements on demand; this creates a stream. We can then apply `takeWhile` to the stream, in order to stop it when a certain condition holds. Finally, if required, the truncated stream may be converted to a list or another type of sequence. In this way, we can

generate a sequence of initially unknown length according to any given requirements.

The Scala library has a general stream-producing function `Stream.iterate`.[2] This function has two arguments, the initial value and a function that computes the next value from the previous one:

```
scala> Stream.iterate(2) { x => x + 10 }
res0: Stream[Int] = Stream(2, ?)
```

The stream is ready to start computing the next elements of the sequence (so far, only the first element, `2`, has been computed). In order to see the next elements, we need to stop the stream at a finite size and then convert the result to a list:

```
scala> Stream.iterate(2) { x => x + 10 }.take(6).toList
res1: List[Int] = List(2, 12, 22, 32, 42, 52)
```

If we try to evaluate `toList` on a stream without first limiting its size via `take` or `takeWhile`, the program will keep producing more elements until it runs out of memory and crashes.

Streams are similar to sequences, and methods such as `map`, `filter`, and `flatMap` are also defined for streams. For instance, the method `drop` skips a given number of initial elements:

```
scala> Seq(10, 20, 30, 40, 50).drop(3)
res2: Seq[Int] = List(40, 50)

scala> Stream.iterate(2) { x => x + 10 }.drop(3)
res3: Stream[Int] = Stream(32, ?)
```

This example shows that in order to evaluate `drop(3)`, the stream had to compute its elements up to `32` (but the subsequent elements are still not computed).

To figure out the code for `digitsOf`, we first write this function as a mathematical formula. To compute the digits of, say, $n = 2405$, we need to divide n repeatedly by 10, getting a sequence n_k of intermediate numbers ($n_0 = 2405$, $n_1 = 240$, ...) and the corresponding sequence of last digits, $n_k \bmod 10$ (in this example: 5, 0, ...). The sequence n_k is defined using mathematical induction:

- Base case: $n_0 = n$, where n is a given initial integer.
- Inductive step: $n_{k+1} = \left\lfloor \frac{n_k}{10} \right\rfloor$ for $k = 1, 2, ...$

Here $\left\lfloor \frac{n_k}{10} \right\rfloor$ is the mathematical notation for the integer division by 10. Let us tabulate the evaluation of the sequence n_k for $n = 2405$:

$k =$	0	1	2	3	4	5	6
$n_k =$	2405	240	24	2	0	0	0
$n_k \bmod 10 =$	5	0	4	2	0	0	0

The numbers n_k will remain all zeros after $k = 4$. It is clear that the useful part of the sequence is before it becomes all zeros. In this example, the sequence n_k needs to be stopped at $k = 4$. The sequence of digits then becomes $[5, 0, 4, 2]$, and we need to reverse it to obtain $[2, 4, 0, 5]$. For reversing a sequence, the Scala library has the standard method `reverse`. So, a complete implementation for `digitsOf` is:

```
def digitsOf(n: Int): Seq[Int] =
  if (n == 0) Seq(0) else { // n == 0 is a special case.
    Stream.iterate(n) { nk => nk / 10 }
      .takeWhile { nk => nk != 0 }
      .map { nk => nk % 10 }
      .toList.reverse
  }
```

We can shorten the code by using the syntax such as `(_ % 10)` instead of `{ nk => nk % 10 }`,

```
def digitsOf(n: Int): Seq[Int] =
  if (n == 0) Seq(0) else { // n == 0 is a special case.
    Stream.iterate(n)(_ / 10)
      .takeWhile(_ != 0)
      .map(_ % 10)
      .toList.reverse
  }
```

[2] In a future version of Scala 3, the `Stream` class will be replaced by `LazyList`.

The type signature of the method `Stream.iterate` can be written as

```
def iterate[A](init: A)(next: A => A): Stream[A]
```

This shows a close correspondence to a definition by mathematical induction. The base case is the first value, `init`, and the inductive step is a function, `next`, that computes the next element from the previous one. It is a general way of creating sequences whose length is not determined in advance.

2.4 Transforming a sequence into another sequence

We have seen methods such as `map` and `zip` that transform sequences into sequences. However, these methods cannot express a general transformation where the elements of the new sequence are defined by induction and depend on previous elements. An example of this kind is computing the partial sums of a given sequence x_i, say $b_k = \sum_{i=0}^{k-1} x_i$. This formula defines $b_0 = 0$, $b_1 = x_0$, $b_2 = x_0 + x_1$, $b_3 = x_0 + x_1 + x_2$, etc. A definition via mathematical induction may be written like this:

- Base case: $b_0 = 0$.
- Inductive step: Given b_k, we define $b_{k+1} = b_k + x_k$ for $k = 0, 1, 2, \ldots$

The Scala library method `scanLeft` implements a general sequence-to-sequence transformation defined in this way. The code implementing the partial sums is

```
def partialSums(xs: Seq[Int]): Seq[Int] = xs.scanLeft(0){ (x, y) => x + y }

scala> partialSums(Seq(1, 2, 3, 4))
res0: Seq[Int] = List(0, 1, 3, 6, 10)
```

The first argument of `scanLeft` is the base case, and the second argument is an updater function describing the inductive step.

In general, the type of elements of the second sequence is different from that of the first sequence. The updater function takes an element of the first sequence and a previous element of the second sequence, and returns the next element of the second sequence. Note that the result of `scanLeft` is one element longer than the original sequence, because the base case provides an initial value.

Until now, we have seen that `foldLeft` is sufficient to re-implement almost every method that works on sequences, such as `map`, `filter`, or `flatten`. Let us show, as an illustration, how to implement the method `scanLeft` via `foldLeft`. In the implementation, the accumulator contains the previous element of the second sequence together with a growing fragment of that sequence, which is updated as we iterate over the first sequence. The code is:

```
1 def scanLeft[A, B](xs: Seq[A])(b0: B)(next: (B, A) => B): Seq[B] = {
2   val init: (B, Seq[B]) = (b0, Seq(b0))
3   val (_, result) = xs.foldLeft(init) {
4     case ((b, seq), x) =>
5       val newB = next(b, x)
6       (newB, seq :+ newB)
7   }
8   result
9 }
```

To implement the (nameless) updater function for `foldLeft` in lines 4–6, we used a Scala feature that makes it easier to define functions with several arguments containing tuples. In our case, the updater function in `foldLeft` has two arguments: the first is a tuple `(B, Seq[B])`, the second is a value of type `A`. Although the pattern expression `case ((b, seq), x) => ...` appears to match a nested tuple, it is just a special syntax. In reality, this expression matches the two arguments of the updater function and, at the same time, destructures the tuple argument as `(b, seq)`.

2.5 Summary

We have seen a number of ways for translating mathematical induction into Scala code.

What problems can we solve now?

- Compute mathematical expressions involving arbitrary recursion.
- Use the accumulator trick to enforce tail recursion.
- Implement functions with type parameters.
- Use arbitrary inductive (i.e., recursive) formulas to:
  - convert sequences to single values (aggregation or "folding");
  - create new sequences from single values ("unfolding");
  - transform existing sequences into new sequences.

Definition by induction	Scala code example
$f([]) = b\ ;\ \ f(s{+}{+}[x]) = g(f(s), x)$	`f(xs) = xs.foldLeft(b)(g)`
$f([]) = b\ ;\ \ f([x]{+}{+}s) = g(x, f(s))$	`f(xs) = xs.foldRight(b)(g)`
$x_0 = b\ ;\ \ x_{k+1} = g(x_k)$	`xs = Stream.iterate(b)(g)`
$y_0 = b\ ;\ \ y_{k+1} = g(y_k, x_k)$	`ys = xs.scanLeft(b)(g)`

Table 2.1: Implementing mathematical induction.

Table 2.1 shows Scala code implementing those tasks. Iterative calculations are implemented by translating mathematical induction directly into code. In the functional programming paradigm, the programmer does not need to write loops or use array indices. Instead, the programmer reasons about sequences as mathematical values: "Starting from this value, we get that sequence, then transform it into that other sequence," etc. This is a powerful way of working with sequences, dictionaries, and sets. Many kinds of programming errors (such as using an incorrect array index) are avoided from the outset, and the code is shorter and easier to read than code written via loops.

What problems cannot be solved with these tools? We cannot implement a non-tail-recursive function without stack overflow (i.e., without unlimited growth of intermediate expressions). The accumulator trick does not always work! In some cases, it is impossible to implement tail recursion in a given recursive computation. An example of such a computation is the "merge-sort" algorithm where the function body must contain two recursive calls within a single expression. (It is impossible to rewrite *two* recursive calls as *one* tail call.)

What if our recursive code cannot be transformed into tail-recursive code via the accumulator trick, but the recursion depth is so large that stack overflows occur? There exist special techniques (e.g., "continuations" and "trampolines") that convert non-tail-recursive code into iterative code without stack overflows. Those techniques are beyond the scope of this book.

2.5.1 Solved examples

Example 2.5.1.1 Compute the smallest n such that $f(f(f(...f(1)...) \geq 1000$, where the function f is applied n times. Write this as a function taking f, 1, and 1000 as arguments. Test with $f(x) = 2x + 1$.

Solution Define a stream of values $[1, f(1), f(f(1)), ...]$ and use `takeWhile` to stop the stream when the values reach 1000. The number n is then found as the length of the resulting sequence:

```
scala> Stream.iterate(1)(x => 2 * x + 1).takeWhile(x => x < 1000).toList
res0: List[Int] = List(1, 3, 7, 15, 31, 63, 127, 255, 511)

scala> Stream.iterate(1)(x => 2 * x + 1).takeWhile(x => x < 1000).length
res1: Int = 9
```

Example 2.5.1.2 **(a)** For a given `Stream[Int]`, compute the stream of the largest values seen so far.

(b) Compute the stream of k largest values seen so far (k is a given integer parameter).

Solution We cannot use `max` or sort the entire stream, since the length of the stream is not known in advance. So we need to use `scanLeft`, which will build the output stream one element at a time.

(a) Maintain the largest value seen so far in the accumulator of the `scanLeft`:

```
def maxSoFar(xs: Stream[Int]): Stream[Int] =
  xs.scanLeft(xs.head) { (max, x) => math.max(max, x) }.drop(1)
```

We use `drop(1)` to remove the initial value (`xs.head`) because it is not useful for our result but is always produced by `scanLeft`.

To test this function, let us define a stream whose values go up and down:

```
val s = Stream.iterate(0)(x => 1 - 2 * x)

scala> s.take(10).toList
res0: List[Int] = List(0, 1, -1, 3, -5, 11, -21, 43, -85, 171)

scala> maxSoFar(s).take(10).toList
res1: List[Int] = List(0, 1, 1, 3, 3, 11, 11, 43, 43, 171)
```

(b) We again use `scanLeft`, where now the accumulator needs to keep the largest k values seen so far. There are two ways of maintaining this accumulator: First, to have a sequence of k values that we sort and truncate each time. Second, to use a specialized data structure such as a priority queue that automatically keeps values sorted and its length bounded. For the purposes of this example, let us avoid using specialized data structures:

```
def maxKSoFar(xs: Stream[Int], k: Int): Stream[Seq[Int]] = {
  // The initial value of the accumulator is an empty Seq() of type Seq[Int].
  xs.scanLeft(Seq[Int]()) { (seq, x) =>
  // Sort in descending order, and take the first k values.
    (seq :+ x).sorted.reverse.take(k)
  }.drop(1) // Skip the undesired first value.
}

scala> maxKSoFar(s, 3).take(10).toList
res2: List[Seq[Int]] = List(List(0), List(1, 0), List(1, 0, -1), List(3, 1, 0), List(3, 1, 0),
     List(11, 3, 1), List(11, 3, 1), List(43, 11, 3), List(43, 11, 3), List(171, 43, 11))
```

Example 2.5.1.3 Find the last element of a non-empty sequence. (Hint: use `reduce`.)

Solution This function is available in the Scala library as the standard method `last` on sequences. Here we need to re-implement it using `reduce`. Begin by writing an inductive definition:

- (Base case.) `last(Seq(x)) == x`.
- (Inductive step.) `last(x +: xs) == last(xs)` assuming `xs` is non-empty.

The `reduce` method implements an inductive aggregation similarly to `foldLeft`, except that for `reduce` the base case always returns `x` for a 1-element sequence `Seq(x)`. This is exactly what we need here, so the inductive definition is directly translated into code, with the updater function $g(x, y) = y$:

```
def last[A](xs: Seq[A]): A = xs.reduce { (x, y) => y }
```

Example 2.5.1.4 **(a)** Count the occurrences of each distinct word in a string:

```
def countWords(s: String): Map[String, Int] = ???

scala> countWords("a quick a quick a brown a fox")
res0: Map[String, Int] = Map(a -> 4, quick -> 2, brown -> 1, fox -> 1)
```

(b) Count the occurrences of each distinct element in a sequence of type `Seq[A]`.

Solution **(a)** We split the string into an array of words via `s.split(" ")`, and apply a `foldLeft` to that array, since the computation is a kind of aggregation over the array of words. The accumulator of the aggregation will be the dictionary of word counts for all the words seen so far:

```
def countWords(s: String): Map[String, Int] = {
  val init: Map[String, Int] = Map()
  s.split(" ").foldLeft(init) { (dict, word) =>
```

```
    val newCount = dict.getOrElse(word, 0) + 1
    dict.updated(word, newCount)
  }
}
```

An alternative, shorter implementation of the same function is

```
def countWords(s: String): Map[String, Int] = s.split(" ").groupBy(w => w).mapValues(_.length)
```

The `groupBy` creates a dictionary in one function call rather than one entry at a time. But the resulting dictionary contains word lists instead of word counts, so we use `mapValues`:

```
scala> "a a b b b c".split(" ").groupBy(w => w)
res0: Map[String,Array[String]] = Map(b -> Array(b, b, b), a -> Array(a, a), c -> Array(c))

scala> res0.mapValues(_.length)
res1: Map[String,Int] = Map(b -> 3, a -> 2, c -> 1)
```

(b) The main code of `countWords` does not depend on the fact that words are of type `String`. It will work in the same way for any other type of keys for the dictionary. So we keep the same code (except for renaming `word` to `x`) and replace `String` by a type parameter `A` in the type signature:

```
def countValues[A](xs: Seq[A]): Map[A, Int] =
  xs.foldLeft(Map[A, Int]()) { (dict, x) =>
    val newCount = dict.getOrElse(x, 0) + 1
    dict.updated(x, newCount)
  }

scala> countValues(Seq(100, 100, 200, 100, 200, 200, 100))
res0: Map[Int,Int] = Map(100 -> 4, 200 -> 3)
```

Example 2.5.1.5 **(a)** Implement the binary search algorithm[3] for a sorted sequence `xs: Seq[Int]` as a function returning the index of the requested value `goal` (assume that `xs` always contains `goal`):

```
@tailrec def binSearch(xs: Seq[Int], goal: Int): Int = ???

scala> binSearch(Seq(1, 3, 5, 7), 5)
res0: Int = 2
```

(b) Re-implement `binSearch` using `Stream.iterate` without writing explicitly recursive code.

Solution **(a)** The binary search algorithm splits the array into two halves and may continue the search recursively in one of the halves. We need to write the solution as a tail-recursive function with an additional accumulator argument. So we expect that the code should look like this:

```
@tailrec def binSearch(xs: Seq[Int], goal: Int, acc: _ = ???): Int = {
  if (???) acc  // This condition must decide whether we are finished.
  else {
    // Determine which half of the sequence contains ‘goal‘.
    // Then update the accumulator accordingly.
    val newAcc = ???
    binSearch(xs, goal, newAcc) // Tail-recursive call.
  }
}
```

We will first decide the type and the initial value of the accumulator, then implement the updater.

The information required for the recursive call must show the segment of the sequence where the target number is present. That segment is defined by two indices i, j representing the left and the right bounds of the sub-sequence, such that the target element is x_n with $x_i \leq x_n \leq x_{j-1}$. It follows that the accumulator should be a pair of two integers (i, j). The initial value of the accumulator is the pair $(0, N)$, where N is the length of the entire sequence. The search is finished when $i + 1 = j$. For convenience, we introduce *two* accumulator values (`left` and `right`) representing i and j:

```
@tailrec def binSearch(xs: Seq[Int], goal: Int)(left: Int = 0, right: Int = xs.length): Int = {
```

[3] https://en.wikipedia.org/wiki/Binary_search_algorithm

```
  // Check whether ‘goal‘ is at one of the boundaries.
  if (right - left <= 1 || xs(left) == goal) left
  else {
    val middle = (left + right) / 2
    // Determine which half of the array contains ‘target‘.
    // Update the accumulator accordingly.
    val (newLeft, newRight) =
      if (goal < xs(middle)) (left, middle)
      else (middle, right)
    binSearch(xs, goal)(newLeft, newRight) // Tail-recursive call.
  }
}

scala> binSearch(0 to 10, 3)() // Default accumulator values.
res0: Int = 3
```

Here we used a feature of Scala that allows us to set `xs.length` as a default value for the argument `right` of `binSearch`. This works because `right` is in a different **argument list** from `xs`. Default values in an argument list may depend on arguments in a *previous* argument list. However, the code

```
def binSearch(xs: Seq[Int], goal: Int, left: Int = 0, right: Int = xs.length)
```

will generate an error: the arguments in the same argument list cannot depend on each other. (The error will say `not found: value xs`.)

(b) We can visualize the binary search as a procedure that generates a stream of progressively tighter bounds for the location of `goal`. The initial bounds are `(0, xs.length)`, and the final bounds are `(k, k+1)` for some `k`. We can generate the sequence of bounds using `Stream.iterate` and stop the sequence when the bounds become sufficiently tight. To detect that, we use the `find` method:

```
def binSearch(xs: Seq[Int], goal: Int): Int = {
  type Acc = (Int, Int)
  val init: Acc = (0, xs.length)
  val updater: Acc => Acc = { case (left, right) =>
    if (right - left <= 1 || xs(left) == goal) (left, left + 1)
    else {
      val middle = (left + right) / 2
      // Determine which half of the array contains ‘target‘.
      // Update the accumulator accordingly.
      if (goal < xs(middle)) (left, middle)
      else (middle, right)
    }
  }

  Stream.iterate(init)(updater)
    .find { case (x, y) => y - x <= 1 }  // Find the element with tight enough bounds.
    .get._1                              // Take the ‘left‘ bound from that element.
}
```

In this code, recursion is delegated to `Stream.iterate` and is cleanly separated from the “business logic” (i.e., implementing the base case, the inductive step, and the post-processing).

Example 2.5.1.6 For a given positive `n:Int`, compute the sequence $[s_0, s_1, s_2, ...]$ defined by $s_0 = SD(n)$ and $s_k = SD(s_{k-1})$ for $k > 0$, where $SD(x)$ is the sum of the decimal digits of the integer x, e.g., $SD(123) = 6$. Stop the sequence s_i when the numbers begin repeating. For example, $SD(99) = 18$, $SD(18) = 9$, $SD(9) = 9$. So, for $n = 99$, the sequence s_i must be computed as $[99, 18, 9]$.

Hint: use `Stream.iterate` and `scanLeft`.

Solution We need to implement a function `sdSeq` having the type signature

```
def sdSeq(n: Int): Seq[Int]
```

First, we need to implement $SD(x)$. The sum of digits is obtained similarly to Section 2.3:

```
def SD(n: Int): Int = Stream.iterate(n)(_ / 10).takeWhile(_ != 0).map(_ % 10).sum
```

Let us compute the sequence $[s_0, s_1, s_2, ...]$ by repeatedly applying `SD` to some number, say, `99`:

```
scala> Stream.iterate(99)(SD).take(10).toList
res1: List[Int] = List(99, 18, 9, 9, 9, 9, 9, 9, 9, 9)
```

We need to stop the stream when the values start to repeat, keeping the first repeated value. In the example above, we need to stop the stream after the value `9` (but include that value). One solution is to transform the stream via `scanLeft` into a stream of *pairs* of consecutive values, so that it becomes easier to detect repetition:

```
scala> Stream.iterate(99)(SD).scanLeft((0,0)) { case ((prev, x), next) => (x, next) }.take(8).toList
res2: List[(Int, Int)] = List((0,0), (0,99), (99,18), (18,9), (9,9), (9,9), (9,9), (9,9))

scala> res2.drop(1).takeWhile { case (x, y) => x  != y }
res3: List[(Int, Int)] = List((0,99), (99,18), (18,9))
```

This looks right; it remains to remove the first parts of the tuples:

```
def sdSeq(n: Int): Seq[Int] = Stream.iterate(n)(SD)            // Stream[Int]
    .scanLeft((0,0)) { case ((prev, x), next) => (x, next) }   // Stream[(Int, Int)]
    .drop(1).takeWhile { case (x, y) => x  != y }              // Stream[(Int, Int)]
    .map(_._2)                                                 // Stream[Int]
    .toList                                                    // List[Int]

scala> sdSeq(99)
res3: Seq[Int] = List(99, 18, 9)
```

Example 2.5.1.7 Implement a function `unfold` with the type signature

```
def unfold[A](init: A)(next: A => Option[A]): Stream[A]
```

The function should create a stream of values of type `A` with the initial value `init`. Next elements are computed from previous ones via the function `next` until it returns `None`. An example test:

```
scala> unfold(0) { x => if (x > 5) None else Some(x + 2) }
res0: Stream[Int] = Stream(0, ?)

scala> res0.toList
res1: List[Int] = List(0, 2, 4, 6)
```

Solution We can formulate the task as an inductive definition of a stream. If `next(init) == None`, the stream must stop at `init`. (This is the base case of the induction). Otherwise, `next(init) == Some(x)` yields a new value `x` and indicates that we need to continue to "unfold" the stream with `x` instead of `init`. (This is the inductive step.) Streams can be created from individual values via the Scala standard library method `Stream.cons` that constructs a stream from a single value and a tail:

```
def unfold[A](init: A)(next: A => Option[A]): Stream[A] = next(init) match {
  case None      => Stream(init)                  // A stream containing a single value ‘init‘.
  case Some(x)   => Stream.cons(init, unfold(x)(next)) // ‘init‘ followed by the tail of stream.
}
```

Example 2.5.1.8 For a given stream $[s_0, s_1, s_2, ...]$ of type `Stream[T]`, compute the "half-speed" stream $h = [s_0, s_0, s_1, s_1, s_2, s_2, ...]$. The half-speed sequence h is defined as $h_{2k} = h_{2k+1} = s_k$ for $k = 0, 1, 2, ...$

Solution We use `map` to replace each element s_i by a sequence containing two copies of s_i. Let us try this on a sample sequence:

```
scala> Seq(1,2,3).map( x => Seq(x, x))
res0: Seq[Seq[Int]] = List(List(1, 1), List(2, 2), List(3, 3))
```

The result is almost what we need, except we need to `flatten` the nested list:

```
scala> Seq(1,2,3).map( x => Seq(x, x)).flatten
res1: Seq[Seq[Int]] = List(1, 1, 2, 2, 3, 3)
```

The composition of `map` and `flatten` is `flatMap`, so the final code is

```
def halfSpeed[T](str: Stream[T]): Stream[T] = str.flatMap(x => Seq(x, x))
```

```
scala> halfSpeed(Seq(1,2,3).toStream)
res2: Stream[Int] = Stream(1, ?)

scala> halfSpeed(Seq(1,2,3).toStream).toList
res3: List[Int] = List(1, 1, 2, 2, 3, 3)
```

Example 2.5.1.9 (The **loop detection** problem.) Stop a given stream $[s_0, s_1, s_2, ...]$ at a place k where the sequence repeats itself; that is, an element s_k equals some earlier element s_i with $i < k$.

Solution The trick is to create a half-speed sequence h_i out of s_i and then find an index $k > 0$ such that $h_k = s_k$. (The condition $k > 0$ is needed because we will always have $h_0 = s_0$.) If we find such an index k, it would mean that either $s_k = s_{k/2}$ or $s_k = s_{(k-1)/2}$; in either case, we will have found an element s_k that equals an earlier element.

As an example, for an input sequence $s = [1,3,5,7,9,3,5,7,9,...]$ we obtain the half-speed sequence $h = [1,1,3,3,5,5,7,7,9,9,3,3,...]$. Looking for an index $k > 0$ such that $h_k = s_k$, we find that $s_7 = h_7 = 7$. The element s_7 indeed repeats an earlier element (although s_7 is not the first such repetition).

There are in principle two ways of finding an index $k > 0$ such that $h_k = s_k$: First, to iterate over a list of indices $k = 1, 2, ...$ and evaluate the condition $h_k = s_k$ as a function of k. Second, to build a sequence of pairs (h_i, s_i) and use `takeWhile` to stop at the required index. In the present case, we cannot use the first way because we do not have a fixed set of indices to iterate over. Also, the condition $h_k = s_k$ cannot be directly evaluated as a function of k because s and h are streams that compute elements on demand, not lists whose elements are computed in advance and ready for use.

So the code must iterate over a stream of pairs (h_i, s_i):

```
def stopRepeats[T](str: Stream[T]): Stream[T] = {
  val halfSpeed = str.flatMap(x => Seq(x, x))
  val result = halfSpeed.zip(str) // Stream[(T, T)]
  .drop(1) // Enforce the condition k > 0.
  .takeWhile { case (h, s) => h != s } // Stream[(T, T)]
  .map(_._2) // Stream[T]
  str.head +: result // Prepend the first element that was dropped.
}

scala> stopRepeats(Seq(1, 3, 5, 7, 9, 3, 5, 7, 9).toStream).toList
res0: List[Int] = List(1, 3, 5, 7, 9, 3, 5)
```

Example 2.5.1.10 Reverse each word in a string but keep the order of words:

```
def revWords(s: String): String = ???

scala> revWords("A quick brown fox")
res0: String = A kciuq nworb xof
```

Solution The standard method `split` converts a string into an array of words:

```
scala> "pa re ci vo mu".split(" ")
res0: Array[String] = Array(pa, re, ci, vo, mu)
```

Each word is reversed with `reverse`; the resulting array is concatenated into a string with `mkString`:

```
def revWords(s: String): String = s.split(" ").map(_.reverse).mkString(" ")
```

Example 2.5.1.11 Remove adjacent repeated characters from a string:

```
def noDups(s: String): String = ???

scala> noDups("abbcdeeeeefddgggggh")
res0: String = abcdefdgh
```

Solution A string is automatically converted into a sequence of characters when we use methods such as `map` or `zip` on it. So, we can use `s.zip(s.tail)` to get a sequence of pairs (s_k, s_{k+1}) where c_k is

the k-th character of the string s. A `filter` will then remove elements s_k for which $s_{k+1} = s_k$:

```
scala> val s = "abbcd"
s: String = abbcd

scala> s.zip(s.tail).filter { case (sk, skPlus1) => sk != skPlus1 }
res0: IndexedSeq[(Char, Char)] = Vector((a,b), (b,c), (c,d))
```

It remains to convert this sequence of pairs into the string `"abcd"`. One way of doing this is to project the sequence of pairs onto the second parts of the pairs:

```
scala> res0.map(_._2).mkString
res1: String = bcd
```

We just need to add the first character, `'a'`. The resulting code is

```
def noDups(s: String): String = if (s == "") "" else {
  val pairs = s.zip(s.tail).filter { case (x, y) => x != y }
  pairs.head._1 +: pairs.map(_._2).mkString
}
```

The method `+:` prepends an element to a sequence, so `x +: xs` is equivalent to `Seq(x) ++ xs`.

Example 2.5.1.12 For a given sequence of type `Seq[A]`, find the longest subsequence that does not contain any adjacent duplicate values.

```
def longestNoDups[A](xs: Seq[A]): Seq[A] = ???

scala> longestNoDups(Seq(1, 2, 2, 5, 4, 4, 4, 8, 2, 3, 3))
res0: Seq[Int] = List(4, 8, 2, 3)
```

Solution This is a dynamic programming[4] problem. Many such problems are solved with a single `foldLeft`. The accumulator represents the current "state" of the dynamic programming solution, and the "state" is updated with each new element of the input sequence.

We first need to determine the type of the accumulator value, or the "state". The task is to find the longest subsequence without adjacent duplicates. So the accumulator should represent the longest subsequence found so far, as well as any required extra information about other subsequences that might grow as we iterate over the elements of `xs`. What is that extra information in our case?

Imagine creating the set of *all* subsequences that have no adjacent duplicates. For the input sequence $[1,2,2,5,4,4,4,8,2,3,3]$, this set of all subsequences will be $\{[1,2],[2,5,4],[4,8,2,3]\}$. We can build this set incrementally in the accumulator value of a `foldLeft`. To visualize how this set would be built, consider the partial result after seeing the first 8 elements of the input sequence, $[1,2,2,5,4,4,4,8]$. The partial set of non-repeating subsequences is $\{[1,2],[2,5,4],[4,8]\}$. When we see the next element, 2, we will update that partial set to $\{[1,2],[2,5,4],[4,8,2]\}$.

It is now clear that the subsequence $[1,2]$ has no chance of being the longest subsequence, since $[2,5,4]$ is already longer. However, we do not yet know whether $[2,5,4]$ or $[4,8,2]$ is the winner, because the subsequence $[4,8,2]$ could still grow and become the longest one (and it does become $[4,8,2,3]$ later). At this point, we need to keep both of these two subsequences in the accumulator, but we may already discard $[1,2]$.

We have deduced that the accumulator needs to keep only *two* sequences: the first sequence is already terminated and will not grow, the second sequence ends with the current element and may yet grow. The initial value of the accumulator is empty. The first subsequence is discarded when it becomes shorter than the second. The code can be written now:

```
def longestNoDups[A](xs: Seq[A]): Seq[A] = {
  val init: (Seq[A], Seq[A]) = (Seq(), Seq())
  val (first, last) = xs.foldLeft(init) { case ((first, current), x) =>
                    // If ‘current‘ is empty, ‘x‘ is not considered to be repeated.
    val xWasRepeated = current != Seq() && current.last == x
    val firstIsLongerThanCurrent = first.length > current.length
```

[4] https://en.wikipedia.org/wiki/Dynamic_programming

```
                    // Compute the new pair ‘(first, current)‘.
                    // Keep ‘first‘ only if it is longer; otherwise replace it by ‘current‘.
    val newFirst = if (firstIsLongerThanCurrent) first else current
                    // Append ‘x‘ to ‘current‘ if ‘x‘ is not repeated.
    val newCurrent = if (xWasRepeated) Seq(x) else current :+ x
    (newFirst, newCurrent)
  }
                    // Return the longer of the two subsequences; prefer ‘first‘.
  if (first.length >= last.length) first else last
}
```

2.5.2 Exercises

Exercise 2.5.2.1 Compute the sum of squared digits of a given integer; e.g., `dsq(123) = 14` (see Example 2.5.1.6). Generalize the solution to take as an argument an function `f: Int => Int` replacing the squaring operation. The required type signature and a sample test:

```
def digitsFSum(x: Int)(f: Int => Int): Int = ???

scala> digitsFSum(123){ x => x * x }
res0: Int = 14

scala> digitsFSum(123){ x => x * x * x }
res1: Int = 36
```

Exercise 2.5.2.2 Compute the **Collatz sequence** c_i as a stream defined by

$$c_0 = n \quad ; \qquad c_{k+1} = \begin{cases} c_k/2 & \text{if } c_k \text{ is even,} \\ 3 * c_k + 1 & \text{if } c_k \text{ is odd.} \end{cases}$$

Stop the stream when it reaches 1 (as one would expect[5] it will).

Exercise 2.5.2.3 For a given integer n, compute the sum of cubed digits, then the sum of cubed digits of the result, etc.; stop the resulting sequence when it repeats itself, and so determine whether it ever reaches 1. (Use Exercise 2.5.2.1.)

```
def cubes(n: Int): Stream[Int] = ???

scala> cubes(123).take(10).toList
res0: List[Int] = List(123, 36, 243, 99, 1458, 702, 351, 153, 153, 153)

scala> cubes(2).take(10).toList
res1: List[Int] = List(2, 8, 512, 134, 92, 737, 713, 371, 371, 371)

scala> cubes(4).take(10).toList
res2: List[Int] = List(4, 64, 280, 520, 133, 55, 250, 133, 55, 250)

def cubesReach1(n: Int): Boolean = ???

scala> cubesReach1(10)
res3: Boolean = true

scala> cubesReach1(4)
res4: Boolean = false
```

Exercise 2.5.2.4 For `a`, `b`, `c` of type `Set[Int]`, compute the set of all sets of the form `Set(x, y, z)` where `x` is from `a`, `y` from `b`, and `z` from `c`. The required type signature and a sample test:

```
def prod3(a: Set[Int], b: Set[Int], c: Set[Int]): Set[Set[Int]] = ???

scala> prod3(Set(1,2), Set(3), Set(4,5))
```

[5] https://en.wikipedia.org/wiki/Collatz_conjecture

```
res0: Set[Set[Int]] = Set(Set(1,3,4), Set(1,3,5), Set(2,3,4), Set(2,3,5))
```

Hint: use `flatMap`.

Exercise 2.5.2.5* Same task as in Exercise 2.5.2.4 for a set of sets: instead of just three sets `a`, `b`, `c`, a `Set[Set[Int]]` is given. The required type signature and a sample test:

```
def prodSet(si: Set[Set[Int]]): Set[Set[Int]] = ???

scala> prodSet(Set(Set(1,2), Set(3), Set(4,5), Set(6)))
res0: Set[Set[Int]] = Set(Set(1,3,4,6),Set(1,3,5,6),Set(2,3,4,6),Set(2,3,5,6))
```

Hint: use `foldLeft` and `flatMap`.

Exercise 2.5.2.6* In a sorted array `xs:Array[Int]` where no values are repeated, find all pairs of values whose sum equals a given number n. Use tail recursion. A type signature and a sample test:

```
def pairs(goal: Int, xs: Array[Int]): Set[(Int, Int)] = ???

scala> pairs(10, Array(1, 2, 3, 4, 5, 6, 7, 8))()
res0: Set[(Int, Int)] = Set((2,8), (3,7), (4,6), (5,5))
```

Exercise 2.5.2.7 Reverse a sentence's word order, but keep the words unchanged:

```
def revSentence(s: String): String = ???

scala> revSentence("A quick brown fox") // Words are separated by a single space.
res0: String = "fox brown quick A"
```

Exercise 2.5.2.8 **(a)** Reverse an integer's digits (see Example 2.5.1.6) as shown:

```
def revDigits(n: Int): Int = ???

scala> revDigits(12345)
res0: Int = 54321
```

(b) A **palindrome integer** is an integer number `n` such that `revDigits(n) == n`. Write a predicate function of type `Int => Boolean` that checks whether a given positive integer is a palindrome.

Exercise 2.5.2.9 Define a function `findPalindrome: Long => Long` performing the following computation: First define `f(n) = revDigits(n) + n` for a given integer `n`, where the function `revDigits` was defined in Exercise 2.5.2.8. If `f(n)` is a palindrome integer, `findPalindrome` returns that integer. Otherwise, it keeps applying the same transformation and computes `f(n)`, `f(f(n))`, ..., until a palindrome integer is eventually found (this is mathematically guaranteed). A sample test:

```
scala> findPalindrome(10101)
res0: Long = 10101

scala> findPalindrome(123)
res0: Long = 444

scala> findPalindrome(83951)
res1: Long = 869363968
```

Exercise 2.5.2.10 Transform a given sequence `xs: Seq[Int]` into a sequence `Seq[(Int, Int)]` of pairs that skip one neighbor. Implement this transformation as a function `skip1` with a type parameter `A` instead of the type `Int`. The required type signature and a sample test:

```
def skip1[A](xs: Seq[A]): Seq[(A, A)] = ???

scala> skip1(List(1,2,3,4,5))
res0: List[Int] = List((1,3), (2,4), (3,5))
```

Exercise 2.5.2.11 **(a)** For a given integer interval $[n_1, n_2]$, find the largest integer $k \in [n_1, n_2]$ such that the decimal representation of k does *not* contain any of the digits 3, 5, or 7. **(b)** For a given integer interval $[n_1, n_2]$, find the integer $k \in [n_1, n_2]$ with the largest sum of decimal digits. **(c)** A

positive integer n is called a **perfect number** if it is equal to the sum of its divisors (other integers k such that $k < n$ and n/k is an integer). For example, 6 is a perfect number because its divisors are 1, 2, and 3, and $1+2+3 = 6$, while 8 is not a perfect number because its divisors are 1, 2, and 4, and $1+2+4 = 7 \neq 8$. Write a function that determines whether a given number n is perfect. Determine all perfect numbers up to one million.

Exercise 2.5.2.12 Remove adjacent repeated elements from a sequence of type `Seq[A]` when they are repeated more than k times. Repetitions up to k times should remain unchanged. The required type signature and a sample test:

```
def removeDups[A](s: Seq[A], k: Int): Seq[A] = ???

scala> removeDups(Seq(1, 1, 1, 1, 5, 2, 2, 5, 5, 5, 5, 5, 1), 3)
res0: Seq[Int] = List(1, 1, 1, 5, 2, 2, 5, 5, 5, 1)
```

Exercise 2.5.2.13 Implement a function `unfold2` with the type signature

```
def unfold2[A,B](init: A)(next: A => Option[(A,B)]): Stream[B]
```

The function should create a stream of values of type `B` by repeatedly applying the given function `next` until it returns `None`. At each iteration, `next` should be applied to the value of type `A` returned by the previous call to `next`. An example test:

```
scala> unfold2(0) { x => if (x > 5) None else Some((x + 2, s"had $x")) }
res0: Stream[String] = Stream(had 0, ?)

scala> res0.toList
res1: List[String] = List(had 0, had 2, had 4)
```

Exercise 2.5.2.14* **(a)** Remove repeated elements (whether adjacent or not) from a sequence of type `Seq[A]`. (This re-implements the standard library's method `distinct`.)

(b) For a sequence of type `Seq[A]`, remove all elements that are repeated (whether adjacent or not) more than k times:

```
def removeK[A](k: Int, xs: Seq[A]): Seq[A] = ???

scala> removeK(2, Seq("a", "b", "a", "b", "b", "c", "b", "a"))
res0: Seq[String] = List(a, b, a, b, c)
```

Exercise 2.5.2.15* For a given sequence `xs:Seq[Double]`, find a subsequence that has the largest sum of values. The sequence `xs` is not sorted, and its values may be positive or negative. The required type signature and a sample test:

```
def maxsub(xs: Seq[Double]): Seq[Double] = ???

scala> maxsub(Seq(1.0, -1.5, 2.0, 3.0, -0.5, 2.0, 1.0, -10.0, 2.0))
res0: Seq[Double] = List(2.0, 3.0, -0.5, 2.0, 1.0)
```

Hint: use dynamic programming and `foldLeft`.

Exercise 2.5.2.16* Using tail recursion, find all common integers between two *sorted* sequences:

```
@tailrec def commonInt(xs: Seq[Int], ys: Seq[Int]): Seq[Int] = ???

scala> commonInt(Seq(1, 3, 5, 7), Seq(2, 3, 4, 6, 7, 8))
res0: Seq[Int] = List(3, 7)
```

2.6 Discussion and further developments

2.6.1 Total and partial functions

In Scala, functions can be total or partial. A **total** function will always compute a result value, while a **partial** function may fail to compute its result for certain values of its arguments.

A simple example of a partial function in Scala is the `max` method: it only works for non-empty sequences. Trying to evaluate it on an empty sequence generates an error called an "exception":

```
scala> Seq(1).tail
res0: Seq[Int] = List()
scala> res0.max
java.lang.UnsupportedOperationException: empty.max
  at scala.collection.TraversableOnce$class.max(TraversableOnce.scala:229)
  at scala.collection.AbstractTraversable.max(Traversable.scala:104)
  ... 32 elided
```

This kind of error may crash the entire program at run time. Unlike the type errors we saw before, which occur at compilation time (i.e., before the program can start), **run-time errors** occur while the program is running, and only when some partial function happens to get an incorrect input. The incorrect input may occur at any point after the program started running, which may crash the entire program in the middle of a long computation.

So, it seems clear that we should write code that does not generate such errors. For instance, it is safe to apply `max` to a sequence if we know that it is non-empty.

Sometimes, a function that uses pattern matching turns out to be a partial function because its pattern matching code fails on certain input data.

If a pattern matching expression fails, the code will throw an exception and stop running. In functional programming, we usually want to avoid this situation because it makes it much harder to reason about program correctness. In most cases, programs can be written to avoid the possibility of match errors. An example of an unsafe pattern matching expression is

```
def h(p: (Int, Int)): Int = p match { case (x, 0) => x }

scala> h( (1,0) )
res0: Int = 1

scala> h( (1,2) )
 scala.MatchError: (1,2) (of class scala.Tuple2$mcII$sp)
  at .h(<console>:12)
  ... 32 elided
```

Here the pattern contains a pattern variable `x` and a constant `0`. This pattern only matches tuples whose second part is equal to `0`. If the second argument is nonzero, a match error occurs and the program crashes. So, `h` is a partial function.

Pattern matching failures never happen if we match a tuple of correct size with a pattern such as `(x, y, z)`, because a pattern variable will always match a value. So, pattern matching with a pattern such as `(x, y, z)` is **infallible** (never fails at run time) when applied to a tuple with 3 elements.

Another way in which pattern matching can be made infallible is by including a pattern that matches everything:

```
p match {
  case (x, 0)   => ...   // This only matches some tuples.
  case _        => ...   // This matches everything.
}
```

If the first pattern `(x, 0)` fails to match the value `p`, the second pattern will be tried (and will always succeed). The `case` patterns in a `match` expression are tried in the order they are written. So, a `match` expression may be made infallible by adding a "match-all" underscore pattern.

2.6.2 Scope and shadowing of pattern matching variables

Pattern matching introduces **locally scoped** variables — that is, variables defined only on the right-hand side of the pattern match expression. As an example, consider this code:

```
def f(x: (Int, Int)): Int = x match { case (x, y) => x + y }

scala> f( (2,4) )
res0: Int = 6
```

The argument of `f` is the variable `x` of a tuple type `(Int,Int)`, but there is also a pattern variable `x` in the case expression. The pattern variable `x` matches the first part of the tuple and has type `Int`. Because variables are locally scoped, the pattern variable `x` is only defined within the expression `x + y`. The argument `x:(Int,Int)` is a completely different variable whose value has a different type.

The code works correctly but is confusing to read because of the name clash between the two quite different variables, both named `x`. Another negative consequence of the name clash is that the argument `x:(Int,Int)` *is invisible* within the case expression: if we write "`x`" in that expression, we will get the pattern variable `x:Int`. One says that the argument `x:(Int,Int)` has been **shadowed** by the pattern variable `x` (which is a "bound variable" inside the case expression).

The problem is easy to avoid: we can give the pattern variable another name. Since the pattern variable is locally scoped, it can be renamed within its scope without affecting any other code:

```
def f(x: (Int, Int)): Int = x match { case (a, b) => a + b }

scala> f( (2,4) )
res0: Int = 6
```

2.6.3 Lazy values and sequences. Iterators and streams

We have used streams to create sequences whose length is not known in advance. An example is a stream containing a sequence of increasing positive integers:

```
scala> val p = Stream.iterate(1)(_ + 1)
p: Stream[Int] = Stream(1, ?)
```

At this point, we have not defined a stopping condition for this stream. In some sense, streams may be seen as "infinite" sequences, although in practice a stream is always finite because computers cannot run infinitely long. Also, computers cannot store infinitely many values in memory.

More precisely, streams are "partially computed" rather than "infinite". The main difference between arrays and streams is that a stream's elements are computed on demand and not all initially available, while an array's elements are all computed in advance and are immediately available.

Generally, there are four possible ways a value could be available:

Availability	Explanation	Example Scala code
"eager"	computed immediately	`val z = f(123)`
"lazy"	computed upon first use	`lazy val z = f(123)`
"on-call"	computed each time it is used	`def z = f(123)`
"never"	cannot be computed due to errors	`val (x, y) = "abc"`

A **lazy value** (declared as `lazy val` in Scala) is computed only when it is needed in some other expression. Once computed, a lazy value stays in memory and will not be re-computed.

An "on-call" value is re-computed every time it is used. In Scala, a `def` declaration does that.

Most collection types in Scala (such as `List`, `Array`, `Set`, and `Map`) are **eager**. All elements of an eager collection are already evaluated.

A stream is a **lazy collection**. Elements of a stream are computed when first needed. After that, they remain in memory and will not be computed again:

```
scala> val str = Stream.iterate(1)(_ + 1)
str: Stream[Int] = Stream(1, ?)

scala> str.take(10).toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> str
res1: Stream[Int] = Stream(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ?)
```

In many cases, it is not necessary to keep previous values of a sequence in memory. For example:

```
scala> (1L to 1000000000L).sum        // Compute the sum of integers from 1 to 1 billion.
res0: Long = 500000000500000000
```

We do not actually need to put a billion numbers in memory if we only want to compute their sum. Indeed, the computation just shown does *not* put all the numbers in memory. The computation will fail if we use a list or a stream:

```
scala> (1L to 1000000000L).toStream.sum
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

The code `(1L to 1000000000L).sum` works because `(1 to n)` produces a sequence whose elements are computed whenever needed but do not remain in memory. This can be seen as a sequence with the "on-call" availability of elements. Sequences of this sort are called **iterators**:

```
scala> 1 to 5
res0: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> 1 until 5
res1: scala.collection.immutable.Range = Range(1, 2, 3, 4)
```

The types `Range` and `Range.Inclusive` are defined in the Scala standard library and are iterators. They behave as collections and support the usual methods (`map`, `filter`, etc.), but they do not store previously computed values in memory.

The `view` method Eager collections such as `List` or `Array` can be converted to iterators by using the `view` method. This is necessary when intermediate collections consume too much memory when fully evaluated. For example, consider the computation of Example 2.1.5.7 where we used `flatMap` to replace each element of an initial sequence by three new numbers before computing `max` of the resulting collection. If instead of three new numbers we wanted to compute *three million* new numbers each time, the intermediate collection created by `flatMap` would require too much memory, and the computation would crash:

```
scala> (1 to 10).flatMap(x => 1 to 3000000).max
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

Even though the range `(1 to 10)` is an iterator, a subsequent `flatMap` operation creates an intermediate collection that is too large for our computer's memory. We can use `view` to avoid this:

```
scala> (1 to 10).view.flatMap(x => 1 to 3000000).max
res0: Int = 3000000
```

The choice between using streams and using iterators is dictated by memory constraints. Except for that, streams and iterators behave similarly to other sequences. We may write programs in the map/reduce style, applying standard methods such as `map`, `filter`, etc., to streams and iterators. Mathematical reasoning about transforming a sequence is the same, whether the sequence is eager, lazy, or on-call.

The `Iterator` class The Scala library class `Iterator` has methods such as `Iterator.iterate` and others, similarly to `Stream`. However, `Iterator` does not behave as a *value* in the mathematical sense:

```
scala> val iter = (1 until 10).toIterator
iter: Iterator[Int] = non-empty iterator

scala> iter.toList // Look at the elements of ‘iter‘.
```

```
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> iter.toList // Look at those elements again...??
res1: List[Int] = List()

scala> iter
res2: Iterator[Int] = empty iterator
```

Evaluating the expression `iter.toList` two times produces a different result the second time. As we see from the Scala output, the value `iter` has become "empty" after the first use.

This situation is impossible in mathematics: if x is a value, such as 100, and f is a function, such as $f(x) = \sqrt{x}$, then $f(x)$ will be the same, $f(100) = \sqrt{100} = 10$, no matter how many times we compute $f(x)$. For instance, we can compute $f(x) + f(x) = 20$ and obtain the correct result. We could also set $y = f(x)$ and compute $y + y = 20$, with the same result. This property is called **referential transparency** or **functional purity** of the function f. After applying a pure function, we can be sure that, for instance, no hidden values in memory have been modified.

When we set $x = 100$ and compute $f(x) + f(x)$, the number 100 does not "become empty" after the first use; its value remains the same. This behavior is what we expect values to have. So, we say that integers "are values" in the mathematical sense. Alternatively, one says that numbers are **immutable,** i.e., cannot be changed. (What would it mean to "modify" the number 10?)

In programming, a type **has value-like behavior** if a computation applied to it always gives the same result. Usually, this means that the type contains immutable data, and the computation is referentially transparent. We can see that Scala's `Range` is immutable and behaves as a value:

```
scala> val x = 1 until 10
x: scala.collection.immutable.Range = Range(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> x.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> x.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Collections such as `List`, `Map`, or `Stream` are immutable. Some elements of a `Stream` may not be evaluated yet, but this does not affect its value-like behavior:

```
scala> val str = (1 until 10).toStream
str: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> str.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> str.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Iterators produced by applying the `view` method to collections will have value-like behavior:

```
scala> val v = (1 until 10).view
v: scala.collection.SeqView[Int,IndexedSeq[Int]] = SeqView(...)

scala> v.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> v.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Due to the lack of value-like behavior, programs written using `Iterator` may not obey the usual rules of mathematical reasoning. This makes it easy to write wrong code that looks correct.

To illustrate the problem, let us re-implement Example 2.5.1.9 by keeping the same code but using `Iterator` instead of `Stream`:

```
def stopRepeatsBad[T](iter: Iterator[T]): Iterator[T] = {
  val halfSpeed = iter.flatMap(x => Seq(x, x))
```

```
  halfSpeed.zip(iter) // Do not prepend the first element. It won't help.
  .drop(1).takeWhile { case (h, s) => h != s }
  .map(_._2)
}

scala> stopRepeatsBad(Seq(1, 3, 5, 7, 9, 3, 5, 7, 9).toIterator).toList
res0: List[Int] = List(5, 9, 3, 7, 9)
```

The result $[5, 9, 3, 7, 9]$ is incorrect, but not in an obvious way: the sequence *was* stopped at a repetition, as we wanted, but some of the elements of the given sequence are missing (while other elements are present). It is difficult to debug a program that produces *partially* correct numbers.

The error in this code occurs in the expression `halfSpeed.zip(iter)` due to the fact that `halfSpeed` was itself defined via `iter`. The result is that `iter` is used twice in this code, which leads to errors because `iter` is mutable and does not behave as a value. Creating an `Iterator` and using it twice in the same expression can give wrong results or even fail with an exception:

```
scala> val s = (1 until 10).toIterator
s: Iterator[Int] = non-empty iterator

scala> val t = s.zip(s).toList
java.util.NoSuchElementException: next on empty iterator
```

It is surprising and counter-intuitive that a variable should not be used twice in an expression. Intuitively, we expect the code `s.zip(s)` to work correctly even though the variable `s` is used twice. When we read the expression `s.zip(s)`, we imagine a given sequence `s` being "zipped" with itself. So, we reason that `s.zip(s)` should produce a sequence of pairs. But Scala's `Iterator` class is **mutable**: it gets modified during its use. This breaks the value-based reasoning about code.

An `Iterator` can be converted to a `Stream` using the `toStream` method. This restores the value-based reasoning because streams behave as values:

```
scala> val iter = (1 until 10).toIterator
iter: Iterator[Int] = non-empty iterator

scala> val str = iter.toStream
str: Stream[Int] = Stream(1, ?)

scala> str.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> str.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> str.zip(str).toList
res2: List[(Int, Int)] = List((1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7), (8,8), (9,9))
```

Instead of `Iterator`, we can use `Stream` and `view` when lazy or on-call collections are required. Newer versions of Scala replace `Stream` with `LazyList`, which is a lazy (and possibly infinite) stream. Libraries such as `scalaz` and `fs2` also provide lazy and on-call streams with correct value-like behavior.

The self-modifying behavior of `Iterator` is an example of a "side effect". A function has a **side effect** if the function's code performs some external action in addition to computing a result value. Examples of side effects are: modifying a value stored in memory; starting and stopping processes or threads; reading or writing files; printing; sending or receiving data over a network; showing images on a display; playing or recording sounds; getting photo or video from a digital camera.

Code that performs side effects does not behave as a value. Evaluating such code twice will perform the side effect twice, which is not the same as just re-using the result value twice. A function with a side effect may return different values each time it is called, even when the same arguments are given to the function. (For example, a digital camera will typically give a different image each time.) **Pure functions** are those that contain no code with side effects. A pure function will always return the same result value when applied to the same arguments. So, pure functions behave similarly to functions that are used in mathematics.

3 The logic of types. I. Disjunctive types

Disjunctive types describe values that belong to a disjoint set of alternatives.

To see how Scala implements disjunctive types, we need to begin by looking at "case classes".

3.1 Scala's case classes

3.1.1 Tuple types with names

It is often helpful to use names for the different parts of a tuple. Suppose that some program represents the size and the color of socks with the tuple type `(Double, String)`. What if the same tuple type `(Double, String)` is used in another place in the program to mean the amount paid and the payee? A programmer could mix the two values by mistake, and it would be hard to find out why the program incorrectly computes, say, the total amount paid:

```
def totalAmountPaid(ps: Seq[(Double, String)]): Double = ps.map(_._1).sum
val x = (10.5, "white")       // Sock size and color.
val y = (25.0, "restaurant")  // Payment amount and payee.

scala> totalAmountPaid(Seq(x, y)) // Nonsense.
res0: Double = 35.5
```

We would prevent this kind of mistake if we could use two *different* types, with names such as `MySock` and `Payment`, for the two kinds of data. There are three basic ways of defining a new named type in Scala: using a type alias, using a class (or "trait"), and using an opaque type.

Opaque types (hiding a type under a new name) is a feature of a future version of Scala 3; so we focus on type aliases and case classes.

A **type alias** is an alternative name for an existing (already defined) type. We could use type aliases in our example to add clarity to the code:

```
type MySockTuple = (Double, String)
type PaymentTuple = (Double, String)

scala> val s: MySockTuple = (10.5, "white")
s: MySockTuple = (10.5,white)

scala> val p: PaymentTuple = (25.0, "restaurant")
p: PaymentTuple = (25.0,restaurant)
```

But type aliases do not prevent mix-up errors:

```
scala> totalAmountPaid(Seq(s, p)) // Nonsense again.
res1: Double = 35.5
```

Scala's **case classes** can be seen as "tuples with names". A case class is equivalent to a tuple type that has a name chosen when we define the case class. Also, each part of the case class will have a separate name that we must choose. This is how to define case classes for the example with socks and payments:

```
case class MySock(size: Double, color: String)
case class Payment(amount: Double, name: String)

scala> val sock = MySock(10.5, "white")
sock: MySock = MySock(10.5,white)
```

```
scala> val paid = Payment(25.0, "restaurant")
paid: Payment = Payment(25.0,restaurant)
```

This code defines new types named `MySock` and `Payment`. Values of type `MySock` are written as `MySock(10.5, "white")`, which is similar to writing the tuple `(10.5, "white")` except for adding the name `MySock` in front of the tuple.

To access the parts of a case class, we use the part names:

```
scala> sock.size
res2: Double = 10.5

scala> paid.amount
res3: Double = 25.0
```

The mix-up error is now a type error detected by the compiler:

```
def totalAmountPaid(ps: Seq[Payment]): Double = ps.map(_.amount).sum

scala> totalAmountPaid(Seq(paid, paid))
res4: Double = 50.0

scala> totalAmountPaid(Seq(sock, paid))
<console>:19: error: type mismatch;
 found   : MySock
 required: Payment
       totalAmountPaid(Seq(sock, paid))
                           ^
```

A function whose argument is of type `MySock` cannot be applied to an argument of type `Payment`. Case classes with different names are *different types*, even if they contain the same parts.

Just as tuples can have any number of parts, case classes can have any number of parts, but the part names must be distinct, for example:

```
case class Person(firstName: String, lastName: String, age: Int)

scala> val noether = Person("Emmy", "Noether", 137)
noether: Person = Person(Emmy,Noether,137)

scala> noether.firstName
res5: String = Emmy

scala> noether.age
res6: Int = 137
```

This data type carries the same information as a tuple `(String, String, Int)`. However, the declaration of a `case class Person` gives the programmer several features that make working with the tuple's data more convenient and less error-prone.

Some (or all) part names may be specified when creating a case class value:

```
scala> val poincaré = Person(firstName = "Henri", lastName = "Poincaré", 165)
poincaré: Person = Person(Henri,Poincaré,165)
```

It is a type error to use wrong types with a case class:

```
scala> val p = Person(140, "Einstein", "Albert")
<console>:13: error: type mismatch;
 found   : Int(140)
 required: String
       val p = Person(140, "Einstein", "Albert")
                      ^
<console>:13: error: type mismatch;
 found   : String("Albert")
 required: Int
       val p = Person(140, "Einstein", "Albert")
                                       ^
```

This error is due to an incorrect order of parts when creating a case class value. However, parts can be specified in any order when using part names:

```
scala> val p = Person(age = 137, lastName = "Noether", firstName = "Emmy")
p: Person = Person(Emmy,Noether,137)
```

A part of a case class can have the type of another case class, creating a type similar to a nested tuple:

```
case class BagOfSocks(sock: MySock, count: Int)
val bag = BagOfSocks(MySock(10.5, "white"), 6)

scala> bag.sock.size
res7: Double = 10.5
```

3.1.2 Case classes with type parameters

Type classes can be defined with type parameters. As an example, consider an extension of `MySock` where, in addition to the size and color, an "extended sock" holds another value. We could define several specialized case classes:

```
case class MySock_Int(size: Double, color: String, value: Int)
case class MySock_Boolean(size: Double, color: String, value: Boolean)
```

but it is better to define a single parameterized case class:

```
case class MySockX[A](size: Double, color: String, value: A)
```

This case class can accommodate every type `A`. We may now create values of `MySockX` containing a `value` of any given type, say `Int`:

```
scala> val s = MySockX(10.5, "white", 123)
s: MySockX[Int] = MySockX(10.5,white,123)
```

Because the value `123` has type `Int`, the type parameter `A` in `MySockX[A]` was automatically set to the type `Int`. The result has type `MySockX[Int]`. The programmer does not need to specify that type explicitly.

Each time we create a value of type `MySockX`, a specific type will have to be used instead of the type parameter `A`. If we want to be explicit, we may write the type parameter like this:

```
scala> val s = MySockX[String](10.5, "white", "last pair")
s: MySockX[String] = MySockX(10.5,white,last pair)
```

However, we can write **parametric code** working with `MySockX[A]`, that is, keeping the type parameter `A` in the code. For example, a function that checks whether a sock of type `MySockX[A]` fits the author's foot can be written as:

```
def fits[A](sock: MySockX[A]): Boolean = (sock.size >= 10.5 && sock.size <= 11.0)
```

This function is defined for all types `A` at once, because its code works in the same way regardless of what `A` is. Scala will set the type parameter `A` automatically when we apply `fits` to an argument:

```
scala> fits(MySockX(10.5, "blue", List(1,2,3))) // Type parameter A = List[Int].
res0: Boolean = true
```

This code forces the type parameter `A` to be `List[Int]`, and so we may omit the type parameter of `fits`. When types become more complicated, it may be helpful to write out some type parameters. The compiler can detect a mismatch between the type parameter `A = List[Int]` used in the "sock" value and the type parameter `A = Int` in the function `fits`:

```
scala> fits[Int](MySockX(10.5, "blue", List(1,2,3)))
<console>:15: error: type mismatch;
 found   : List[Int]
 required: Int
       fits[Int](MySockX(10.5, "blue", List(1,2,3)))
                                       ^
```

Case classes may have several type parameters, and the types of the parts may use these type parameters. Here is an artificial example of a case class using type parameters in different ways,

```
case class Complicated[A,B,C,D](x: (A, A), y: (B, Int) => A, z: C => C)
```

This case class contains parts of different types that use the type parameters `A`, `B`, `C` in tuples and functions. The type parameter `D` is not used at all; this is allowed (and occasionally useful).

A type with type parameters, such as `MySockX` or `Complicated`, is called a **type constructor**. A type constructor "constructs" a new type, such as `MySockX[Int]`, from a given type parameter `Int`. Values of type `MySockX` cannot be created without setting the type parameter. So, it is important to distinguish the type constructor, such as `MySockX`, from a type that can have values, such as `MySockX[Int]`.

3.1.3 Tuples with one part and with zero parts

Let us compare tuples and case classes more systematically.

Parts of a case class are accessed by name with a dot syntax, for example `sock.color`. Parts of a tuple are accessed with the accessors such as `x._1`. This syntax is the same as that for a case class whose parts have names `_1`, `_2`, etc. So, it appears that tuple parts *do* have names in Scala, although those names are always automatically chosen as `_1`, `_2`, etc. Tuple types are also automatically named in Scala as `Tuple2`, `Tuple3`, etc., and they are parameterized, since each part of the tuple may be of any chosen type. A tuple type expression such as `(Int, String)` is just a special syntax for the parameterized type `Tuple2[Int, String]`. One could define the tuple types as case classes like this:

```
case class Tuple2[A, B](_1: A, _2: B)
case class Tuple3[A, B, C](_1: A, _2: B, _3: C)  // And so on with Tuple4, Tuple5...
```

However, these types are already defined in the Scala library.

Proceeding systematically, we ask whether tuple types can have just one part or even no parts. Indeed, Scala defines `Tuple1[A]` (which is rarely used in practice) as a tuple with a single part.

The tuple with zero parts also exists and is called `Unit` (instead of "`Tuple0`"). The syntax for the value of the `Unit` type is an empty tuple, denoted by `()` in Scala. It is clear that the value `()` is the *only* possible value of the `Unit` type. The name "unit" reminds us of that.

At first sight, the `Unit` type — an empty tuple that carries no data — may appear to be useless. It turns out, however, that the `Unit` type is important in functional programming. It is used as a type *guaranteed* to have only a single distinct value. This book will show many examples of using `Unit`.

Case classes may have one part or zero parts, similarly to the one-part and zero-part tuples:

```
case class B(z: Int)     // Tuple with one part.
case class C()           // Tuple with no parts.
```

The following table summarizes the correspondence between tuples and case classes:

Tuples	Case classes
`(123, "xyz"): Tuple2[Int, String]`	`case class A(x: Int, y: String)`
`(123,): Tuple1[Int]`	`case class B(z: Int)`
`(): Unit`	`case class C()`

Scala has an alternative syntax for empty case classes:

```
case object C   // Similar to ‘case class C()‘.
```

There are two main differences between `case class C()` and `case object C`:

- A `case object` cannot have type parameters, while we may define a `case class C[X, Y, Z]()` with type parameters `X`, `Y`, `Z` if needed.
- A `case object` is allocated in memory only once, while new values of a `case class C()` will be allocated in memory each time `C()` is evaluated.

Other than that, `case class C()` and `case object C` have the same meaning: a named tuple with zero parts, which we may also view as a "named `Unit`" type. This book will not use `case object`s because `case class`es are sufficient.

3.1.4 Pattern matching for case classes

Scala performs pattern matching in two situations:

- destructuring definition: `val` *pattern* `= ...`
- `case` expression: `case` *pattern* `=> ...`

Case classes can be used in both situations. A destructuring definition can be used in a function whose argument is of case class type `BagOfSocks`:

```
case class MySock(size: Double, color: String)
case class BagOfSocks(sock: MySock, count: Int)

def printBag(bag: BagOfSocks): String = {
  val BagOfSocks(MySock(size, color), count) = bag // Destructure the ‘bag‘.
  s"bag has $count $color socks of size $size"
}

val bag = BagOfSocks(MySock(10.5, "white"), 6)

scala> printBag(bag)
res0: String = bag has 6 white socks of size 10.5
```

A `case` expression can match a value, extract some pattern variables, and compute a result:

```
def fits(bag: BagOfSocks): Boolean = bag match {
  case BagOfSocks(MySock(size, _), _) => (size >= 10.5 && size <= 11.0)
}
```

In the code of this function, we match the `bag` value against the pattern `BagOfSocks(MySock(size, _), _)`. This pattern will always match and will define `size` as a pattern variable of type `Double`.

The syntax for pattern matching for case classes is similar to the syntax for pattern matching for tuples, except for the presence of the *names* of the case classes. For example, by removing the case class names from the pattern

```
case BagOfSocks(MySock(size, _), _) => ...
```

we obtain the nested tuple pattern

```
case ((size, _), _) => ...
```

that could be used for values of type `((Double, String), Int)`. We see that within pattern matching expressions, case classes behave as tuple types with added names.

Scala's "case classes" got their name from their use in `case` expressions. It is usually more convenient to use `case` expressions with case classes than to use destructuring definitions.

3.2 Disjunctive types

3.2.1 Motivation and first examples

In many situations, it is useful to have several different shapes of data within the same type. As a first example, suppose we are looking for real roots of a quadratic equation $x^2 + bx + c = 0$. There are three cases: no real roots, one real root, and two real roots. It is convenient to have a type that represents "real roots of a quadratic equation"; call it `RootsOfQ`. Inside that type, we distinguish between the three cases, but outside it looks like a single type.

Another example is the binary search algorithm that looks for an integer x in a sorted array. Either the algorithm finds the location of x in the array, or it determines that the array does not contain x. It is convenient if the algorithm could return a value of a single type (say, `SearchResult`) that represents *either* an index at which x is found, *or* the absence of an index.

More generally, we may have computations that *either* return a result *or* generate an error and fail to produce a result. It is then convenient to return a value of a single type (say, `Result`) that represents either a correct result or an error message.

In certain computer games, one has different types of "rooms", each room having certain properties depending on its type. Some rooms are dangerous because of monsters, other rooms contain useful objects, certain rooms allow you to finish the game, and so on. We want to represent all the different kinds of rooms uniformly, as a type `Room`, so that a value of type `Room` automatically stores the correct properties in each case.

In all these situations, data comes in several mutually exclusive shapes. This data can be represented by a single type if that type is able to describe a mutually exclusive set of cases:

- `RootsOfQ` must be either the empty tuple `()`, or `Double`, or a tuple `(Double, Double)`
- `SearchResult` must be either `Int` or the empty tuple `()`
- `Result` must be either an `Int` value or a `String` message

We see that the empty tuple, also known as the `Unit` type, is natural to use in these situations. It is also helpful to assign names to each of the cases:

- `RootsOfQ` is "no roots" with value `()`, or "one root" with value `Double`, or "two roots" with value `(Double, Double)`
- `SearchResult` is "index" with value `Int`, or "not found" with value `()`
- `Result` is "value" of type `Int` or "error message" of type `String`

Scala's case classes provides exactly what we need here — named tuples with zero, one, two or more parts. So, it is natural to use case classes instead of tuples:

- `RootsOfQ` is a value of type `case class NoRoots()`, or a value of type `case class OneRoot(x: Double)`, or a value of type `case class TwoRoots(x: Double, y: Double)`
- `SearchResult` is a value of type `case class Index(x: Int)` or a value of type `case class NotFound()`
- `Result` is a value of type `case class Value(x: Int)` or of type `case class Error(message: String)`

Our three examples are now described as types that select one case class out of a given set. It remains to see how Scala defines such types. For instance, the definition of `RootsOfQ` needs to indicate that the case classes `NoRoots`, `OneRoot`, and `TwoRoots` are the three alternatives allowed by the type `RootsOfQ`. The Scala syntax for that definition looks like this:

```
sealed trait RootsOfQ
final case class NoRoots()                      extends RootsOfQ
final case class OneRoot(x: Double)             extends RootsOfQ
final case class TwoRoots(x: Double, y: Double)  extends RootsOfQ
```

In the definition of `SearchResult`, we have two cases:

```
sealed trait SearchResult
final case class Index(i: Int)   extends SearchResult
final case class NotFound()      extends SearchResult
```

The definition of the `Result` type is parameterized, so that we can describe results of any type (while error messages are always of type `String`):

```
sealed trait Result[A]
final case class Value[A](x: A)              extends Result[A]
final case class Error[A](message: String)   extends Result[A]
```

The "`sealed trait` / `final case class`" syntax defines a type that represents a choice of one case class from a fixed set of case classes. This kind of type is called a **disjunctive** type (or a **co-product** type) in this book. The keywords `final` and `sealed` tell the Scala compiler that the set of case classes within a disjunctive type is fixed and unchangeable.

3.2.2 Solved examples: Pattern matching for disjunctive types

Our first examples of disjunctive types are `RootsOfQ`, `SearchResult`, and `Result[A]` defined in the previous section. We will now look at the Scala syntax for working with disjunctive types.

Consider the disjunctive type `RootsOfQ` with three parts (the case classes `NoRoots`, `OneRoot`, `TwoRoots`). The only way of creating a value of type `RootsOfQ` is to create a value of one of these case classes. This is done by writing expressions such as `NoRoots()`, `OneRoot(2.0)`, or `TwoRoots(1.0, -1.0)`. Scala will accept these expressions as having the type `RootsOfQ`:

```
scala> val x: RootsOfQ = OneRoot(2.0)
x: RootsOfQ = OneRoot(2.0)
```

How can we use a given value, say, `x:RootsOfQ`? The main tool for working with values of disjunctive types is pattern matching. In Chapter 2, we used pattern matching with syntax such as `{ case (x, y) => ... }`. To use pattern matching with disjunctive types, we write *several* `case` patterns because we need to match several possible cases of the disjunctive type:

```
def f(r: RootsOfQ): String = r match {
  case NoRoots()       => "no real roots"
  case OneRoot(r)      => s"one real root: $r"
  case TwoRoots(x, y)  => s"real roots: ($x, $y)"
}

scala> f(x)
res0: String = "one real root: 2.0"
```

Each `case` pattern will introduce its own pattern variables, such as `r`, `x`, `y` as in the code at left. Each pattern variable is defined only within the *local scope*, that is, within the scope of its `case` expression. It is impossible to make a mistake where we, say, refer to the variable `r` within the code that handles the case of two roots.

If the code only needs to work with a subset of cases, we can match all other cases with an underscore character (as in `case _`):

```
scala> x match {
  case OneRoot(r)   => s"one real root: $r"
  case _            => "have something else"
}
res1: String = one real root: 2.0
```

The `match`/`case` expression represents a choice over possible values of a given type. Note the similarity with this code:

```
def f(x: Int): Int = x match {
  case 0    => println(s"error: must be nonzero"); -1
  case 1    => println(s"error: must be greater than 1"); -1
  case _    => x
}
```

The values `0` and `1` are some possible values of type `Int`, just as `OneRoot(4.0)` is a possible value of type `RootsOfQ`. When used with disjunctive types, `match`/`case` expressions will usually cover the complete list of possibilities. If the list of cases is incomplete, the Scala compiler will print a warning:

```
scala> def g(x: RootsOfQ): String = x match {
         case OneRoot(r) => s"one real root: $r"
       }
<console>:14: warning: match may not be exhaustive.
It would fail on the following inputs: NoRoots(), TwoRoots(_, _)
```

This code defines a *partial* function `g` that can be applied only to values of the form `OneRoot(...)` and will fail (throwing an exception) for other values.

Let us look at more examples of using the disjunctive types we just defined.

Example 3.2.2.1 Given a sequence of quadratic equations, compute a sequence containing their real roots as values of type `RootsOfQ`.

Solution Define a case class representing a quadratic equation $x^2 + bx + c = 0$:

```
case class QEqu(b: Double, c: Double)
```

The following function determines how many real roots an equation has:

```
def solve(quadraticEqu: QEqu): RootsOfQ = {
   val QEqu(b, c) = quadraticEqu    // Destructure QEqu.
   val d = b * b / 4 - c
   if (d > 0) {
     val s = math.sqrt(d)
     TwoRoots(- b / 2 - s, - b / 2 + s)
   } else if (d == 0.0) OneRoot(- b / 2)
   else NoRoots()
}
```

Test the `solve` function:

```
scala> solve(QEqu(1,1))
res1: RootsOfQ = NoRoots()

scala> solve(QEqu(1,-1))
res2: RootsOfQ = TwoRoots(-1.618033988749895,0.6180339887498949)

scala> solve(QEqu(6,9))
res3: RootsOfQ = OneRoot(-3.0)
```

We can now implement the function `findRoots`:

```
def findRoots(equs: Seq[QEqu]): Seq[RootsOfQ] = equs.map(solve)
```

If the function `solve` will not be used often, we may want to write it inline as a nameless function:

```
def findRoots(equs: Seq[QEqu]): Seq[RootsOfQ] = equs.map { case QEqu(b, c) =>
  (b * b / 4 - c) match {
    case d if d > 0   =>
      val s = math.sqrt(d)
      TwoRoots(- b / 2 - s, - b / 2 + s)
    case 0.0          => OneRoot(- b / 2)
    case _            => NoRoots()
  }
}
```

This code depends on some features of Scala syntax. We can use the partial function `{ case QEqu(b, c) => ... }` directly as the argument of `map` instead of defining this function separately. This avoids having to destructure `QEqu` at a separate step. The `if`/`else` expression is replaced by an "embedded" `if` within the `case` expression, which is easier to read.

Test the final code:

```
scala> findRoots(Seq(QEqu(1,1), QEqu(2,1)))
res4: Seq[RootsOfQ] = List(NoRoots(), OneRoot(-1.0))
```

Example 3.2.2.2 Given a sequence of values of type `RootsOfQ`, compute a sequence containing only the single roots. Example test:

```
def singleRoots(rs: Seq[RootsOfQ]): Seq[Double] = ???

scala> singleRoots(Seq(TwoRoots(-1, 1), OneRoot(3.0), OneRoot(1.0), NoRoots()))
res5: Seq[Double] = List(3.0, 1.0)
```

Solution We apply `filter` and `map` to the sequence of roots:

```
def singleRoots(rs: Seq[RootsOfQ]): Seq[Double] = rs.filter {
  case OneRoot(x) => true
  case _          => false
}.map { case OneRoot(x) => x }
```

In the `map` operation, we need to cover only the one-root case because the two other possibilities have been excluded ("filtered out") by the preceding `filter` operation.

We can implement the same function by using the standard library's `collect` method that performs the filtering and mapping operation in one step:

```
def singleRoots(rs: Seq[RootsOfQ]): Seq[Double] = rs.collect { case OneRoot(x) => x }
```

Example 3.2.2.3 Implement binary search returning a `SearchResult`. Modify the binary search implementation from Example 2.5.1.5(b) so that it returns a `NotFound` value when appropriate.

Solution The code from Example 2.5.1.5(b) will return *some* index even if the given number is not present in the array:

```
scala> binSearch(Array(1, 3, 5, 7), goal = 5)
res6: Int = 2

scala> binSearch(Array(1, 3, 5, 7), goal = 4)
res7: Int = 1
```

In that case, the array's element at the computed index will not be equal to `goal`. We should return `NotFound()` in that case. The new code can be written as a `match`/`case` expression for clarity:

```
def safeBinSearch(xs: Seq[Int], goal: Int): SearchResult =
  binSearch(xs, goal) match {
    case n if xs(n) == goal   => Index(n)
    case _                    => NotFound()
  }

scala> safeBinSearch(Array(1, 3, 5, 7), 5)
res8: SearchResult = Index(2)

scala> safeBinSearch(Array(1, 3, 5, 7), 4)
res9: SearchResult = NotFound()
```

Example 3.2.2.4 Use the disjunctive type `Result[Int]` to implement "safe arithmetic", where a division by zero or a square root of a negative number gives an error message. Define arithmetic operations directly for values of type `Result[Int]`. Abandon further computations on any error.

Solution Begin by implementing the square root:

```
def sqrt(r: Result[Int]): Result[Int] = r match {
  case Value(x) if x >= 0  => Value(math.sqrt(x).toInt)
  case Value(x)            => Error(s"error: sqrt($x)")
  case Error(m)            => Error(m) // Keep the error message.
}
```

The square root is computed only if we have the `Value(x)` case, and only if $x \geq 0$. If the argument `r` was already an `Error` case, we keep the error message and perform no further computations.

To implement the addition operation, we need a bit more work:

```
def add(rx: Result[Int], ry: Result[Int]): Result[Int] = (rx, ry) match {
  case (Value(x), Value(y)) => Value(x + y)
  case (Error(m), _)        => Error(m)  // Keep the first error message.
  case (_, Error(m))        => Error(m)  // Keep the second error message.
}
```

This code illustrates nested patterns that match the tuple `(rx, ry)` against various possibilities. In this way, the code is clearer than code written with nested `if`/`else` expressions.

Implementing the multiplication operation results in almost the same code:

```
def mul(rx: Result[Int], ry: Result[Int]): Result[Int] = (rx, ry) match {
  case (Value(x), Value(y)) => Value(x * y)
  case (Error(m), _)        => Error(m)
  case (_, Error(m))        => Error(m)
}
```

To avoid repetition, we may define a general function that "maps" operations on integers to operations on `Result[Int]` types:

```
def map2(rx: Result[Int], ry: Result[Int])(op: (Int, Int) => Int): Result[Int] =
  (rx, ry) match {
    case (Value(x), Value(y)) => Value(op(x, y))
    case (Error(m), _)        => Error(m)
    case (_, Error(m))        => Error(m)
  }
```

Now we can easily "map" any binary operation on integers to a binary operation on `Result[Int]`, assuming that the operation never generates an error:

```
def sub(rx: Result[Int], ry: Result[Int]): Result[Int] = map2(rx, ry){ (x, y) => x - y }
```

Custom code is still needed for operations that *may* generate errors:

```
def div(rx: Result[Int], ry: Result[Int]): Result[Int] = (rx, ry) match {
  case (Value(x), Value(y)) if y != 0  => Value(x / y)
  case (Value(x), Value(y))            => Error(s"error: $x / $y")
  case (Error(m), _)                   => Error(m)
  case (_, Error(m))                   => Error(m)
}
```

We can now test the "safe arithmetic" on simple calculations:

```
scala> add(Value(1), Value(2))
res10: Result[Int] = Value(3)

scala> div(add(Value(1), Value(2)), Value(0))
res11: Result[Int] = Error(error: 3 / 0)
```

We see that indeed all further computations are abandoned once an error occurs. An error message shows only the immediate calculation that generated the error. For instance, the error message for $20 + 1/0$ never mentions 20:

```
scala> add(Value(20), div(Value(1), Value(0)))
res12: Result[Int] = Error(error: 1 / 0)

scala> add(sqrt(Value(-1)), Value(10))
res13: Result[Int] = Error(error: sqrt(-1))
```

3.2.3 Standard disjunctive types: `Option`, `Either`, `Try`

The Scala library defines the disjunctive types `Option`, `Either`, and `Try` because they are often useful. **The `Option` type** is a disjunctive type with two cases: the empty tuple and a one-element tuple. The names of the two case classes are `None` and `Some`. If the `Option` type were not already defined in the standard library, one could define it with the code:

```
sealed trait Option[+T]  // The annotation '+T' will be explained in Chapter 6.
final case object None           extends Option[Nothing]
final case class Some[T](t: T)   extends Option[T]
```

This code is similar to the type `SearchResult` defined in Section 3.2.1, except that `Option` has a type parameter instead of a fixed type `Int`. Another difference is the use of a `case object` instead of an empty case class, such as `None()`. Since Scala's `case object`s cannot have type parameters, the type parameter in the definition of `None` must be set to the special type `Nothing`, which is a type with *no*

values (also called the **void type**, unlike Java or C's `void` keyword). The special type annotation `+T` makes `None` usable as a value of type `Option[T]` for any type `T`; see Section 6.1.8 for more details.

An alternative (implemented, e.g., in the `scalaz` library) is to define the empty option value as:

```
final case class None[T]() extends Option[T]
```

In that implementation, the empty option `None[T]()` has a type parameter.

Several consequences follow from the Scala library's decision to define `None` without a type parameter. One consequence is that `None` can be reused as a value of type `Option[A]` for any type `A`:

```
scala> val y: Option[Int] = None
y: Option[Int] = None

scala> val z: Option[String] = None
z: Option[String] = None
```

Typically, `Option` is used in situations where a value may be either present or missing, especially when a missing value is *not an error*.

Example 3.2.3.1 Information about "subscribers" must include a name and an email address, but a telephone number is optional. To represent this information, we define a case class like this:

```
case class Subscriber(name: String, email: String, phone: Option[Long])
```

What if we represent the missing telephone number by a special value such as `-1` and use the simpler type `Long` instead of `Option[Long]`? The disadvantage is that we would need to *remember* to check for the special value `-1` in all functions that take the telephone number as an argument. Looking at a function such as `sendSMS(phone: Long)` at a different place in the code, a programmer might forget that the telephone number is actually optional. In contrast, the type signature `sendSMS(phone: Option[Long])` unambiguously indicates that the telephone number might be missing and helps the programmer to remember to handle both cases.

Pattern-matching code involving `Option` can handle the two cases like this:

```
def getDigits(phone: Option[Long]): Option[Seq[Long]] = phone match {
  case None              => None       // Cannot obtain digits, so return ‘None‘.
  case Some(number)      => Some(digitsOf(number))
}  // The function ‘digitsOf‘ was defined in Section 2.3.
```

At the two sides of "`case None => None`", the value `None` has different types, namely `Option[Long]` and `Option[Seq[Long]]`. Since these types are declared in the type signature of the function `getDigits`, the Scala compiler is able to figure out the types of all expressions in the `match`/`case` construction. So, pattern-matching code can be written without explicit type annotations such as `(None: Option[Long])`.

If we now need to compute the number of digits, we can write:

```
def numberOfDigits(phone: Option[Long]): Option[Long] = getDigits(phone) match {
  case None              => None
  case Some(digits)      => Some(digits.length)
}
```

These examples perform a computation when an `Option` value is non-empty, and leave it empty otherwise. This code pattern is used often. To avoid repeating the code, we can implement this code pattern as a function that takes the computation as an argument `f`:

```
def doComputation(x: Option[Long], f: Long => Long): Option[Long] = x match {
  case None              => None
  case Some(i)           => Some(f(i))
}
```

It is then natural to generalize this function to arbitrary types using type parameters instead of a fixed type `Long`. The resulting function is usually called `fmap` in functional programming libraries:

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  case None              => None
  case Some(a)           => Some(f(a))
```

```
}

scala> fmap(digitsOf)(Some(4096))
res0: Option[Seq[Long]] = Some(List(4, 0, 9, 6))

scala> fmap(digitsOf)(None)
res1: Option[Seq[Long]] = None
```

We say that the `fmap` operation **lifts** a given function of type `A => B` to the type `Option[A] => Option[B]`.

It is important to keep in mind that the code `case Some(a) => Some(f(a))` changes the type of the option value. On the left side of the arrow, the type is `Option[A]`, while on the right side it is `Option[B]`. The Scala compiler knows this from the given type signature of `fmap`, so an explicit type parameter, which we could write as `Some[B](f(a))`, is not needed.

The Scala library implements an equivalent function as a method of the `Option` class, with the syntax `x.map(f)` rather than `fmap(f)(x)`. We can concisely rewrite the previous code using these methods:

```
def getDigits(phone: Option[Long]): Option[Seq[Long]] = phone.map(digitsOf)
def numberOfDigits(phone: Option[Long]): Option[Long] = phone.map(digitsOf).map(_.length)
```

We see that the `map` operation for the `Option` type is analogous to the `map` operation for sequences.

The similarity between `Option[A]` and `Seq[A]` is clearer if we view `Option[A]` as a special kind of "sequence" whose length is restricted to be either 0 or 1. So, `Option[A]` can have all the operations of `Seq[A]` except operations such as `concat` that may grow the sequence beyond length 1. The standard operations defined on `Option` include `map`, `filter`, `zip`, `forall`, `exists`, `flatMap`, and `foldLeft`.

Example 3.2.3.2 Given a phone number as `Option[Long]`, extract the country code if it is present. (Assume that the country code is any digits in front of the 10-digit number; for the phone number 18004151212, the country code is 1.) The result must be again of type `Option[Long]`.

Solution If the phone number is a positive integer n, we may compute the country code simply as `n/10000000000L`. However, if the result of that division is zero, we should return an empty `Option` (i.e. the value `None`) rather than `0`. To implement this logic, we may begin by writing this code:

```
def countryCode(phone: Option[Long]): Option[Long] = phone match {
  case None      => None
  case Some(n)   =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
}
```

We may notice that we have reimplemented the code pattern similar to `map` in this code, namely "if `None` then return `None`, else do a computation". So we may try to rewrite the code as:

```
def countryCode(phone: Option[Long]): Option[Long] = phone.map { n =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
} // Type error: the result is Option[Option[Long]], not Option[Long].
```

This code does not compile: we are returning an `Option[Long]` within a function lifted via `map`, so the resulting type is `Option[Option[Long]]`. We may use `flatten` to convert `Option[Option[Long]]` to the required type `Option[Long]`:

```
def countryCode(phone: Option[Long]): Option[Long] = phone.map { n =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
}.flatten // Types are correct now.
```

Since the `flatten` follows a `map`, we can rewrite the code using `flatMap`:

```
def countryCode(phone: Option[Long]): Option[Long] = phone.flatMap { n =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
} // Types are correct now.
```

Another way of implementing this example is to notice the code pattern "if condition does not hold, return `None`, otherwise keep the value". For an `Option` type, this is equivalent to the `filter` operation (recall that `filter` returns an empty sequence if the predicate never holds). The code is:

```
def countryCode(phone: Option[Long]): Option[Long] = phone.map(_ / 10000000000L).filter(_ != 0L)

scala> countryCode(Some(18004151212L))
res0: Option[Long] = Some(1)

scala> countryCode(Some(8004151212L))
res1: Option[Long] = None
```

Example 3.2.3.3 Add a new requirement to Example 3.2.3.2: if the country code is not present, we should return the default country code 1.

Solution This is an often used code pattern: "if empty, substitute a default value". The Scala library has the method `getOrElse` for this purpose:

```
scala> Some(100).getOrElse(1)
res2: Int = 100

scala> None.getOrElse(1)
res3: Int = 1
```

So we can implement the new requirement as:

```
scala> countryCode(Some(8004151212L)).getOrElse(1L)
res4: Long = 1
```

Using `Option` with collections Many Scala library methods return an `Option` as a result. The main examples are `find`, `headOption`, and `lift` for sequences, and `get` for dictionaries.

The `find` method returns the first element satisfying a predicate:

```
scala> (1 to 10).find(_ > 5)
res0: Option[Int] = Some(6)

scala> (1 to 10).find(_ > 10) // No element is > 10.
res1: Option[Int] = None
```

The `lift` method returns the element of a sequence at a given index:

```
scala> (10 to 100).lift(0)
res2: Option[Int] = Some(10)

scala> (10 to 100).lift(1000) // No element at index 1000.
res3: Option[Int] = None
```

The `headOption` method returns the first element of a sequence, unless the sequence is empty. This is equivalent to `lift(0)`:

```
scala> Seq(1,2,3).headOption
res4: Option[Int] = Some(1)

scala> Seq(1,2,3).filter(_ > 10).headOption
res5: Option[Int] = None
```

Applying `.find(p)` computes the same result as `.filter(p).headOption`, but `.find(p)` may be faster.

The `get` method for a dictionary checks whether the given key is present in the dictionary. If so, `get` returns the value wrapped in `Some()`. Otherwise, it returns `None`:

```
scala> Map(10 -> "a", 20 -> "b").get(10)
res6: Option[String] = Some(a)

scala> Map(10 -> "a", 20 -> "b").get(30)
res7: Option[String] = None
```

The `get` method is a safe by-key access to dictionaries, unlike the direct access that may fail:

```
scala> Map(10 -> "a", 20 -> "b")(10)
res8: String = a

scala> Map(10 -> "a", 20 -> "b")(30)
java.util.NoSuchElementException: key not found: 30
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  at scala.collection.AbstractMap.default(Map.scala:59)
  ... 32 elided
```

Similarly, `lift` is a safe by-index access to collections, unlike the direct access that may fail:

```
scala> Seq(10,20,30)(0)
res9: Int = 10

scala> Seq(10,20,30)(5)
java.lang.IndexOutOfBoundsException: 5
  at scala.collection.LinearSeqOptimized$class.apply(LinearSeqOptimized.scala:65)
  at scala.collection.immutable.List.apply(List.scala:84)
  ... 32 elided
```

The `Either` type The standard disjunctive type `Either[A, B]` has two type parameters and is often used for computations that report errors. By convention, the *first* type (`A`) is the type of error, and the *second* type (`B`) is the type of the (non-error) result. The names of the two cases are `Left` and `Right`. A possible definition of `Either` may be written as:

```
sealed trait Either[A, B]
final case class  Left[A, B](value: A) extends Either[A, B]
final case class Right[A, B](value: B) extends Either[A, B]
```

By convention, a value `Left(x)` represents an error, and a value `Right(y)` represents a valid result.

As an example, the following function substitutes a default value and logs the error information:

```
def logError(x: Either[String, Int], default: Int): Int = x match {
  case Left(error)  => println(s"Got error: $error"); default
  case Right(res)   => res
}
```

To test:

```
scala> logError(Right(123), -1)
res1: Int = 123

scala> logError(Left("bad result"), -1)
Got error: bad result
res2: Int = -1
```

Why use `Either` instead of `Option` for computations that may fail? A failing computation such as `"xyz".toInt` cannot return a result, and sometimes we might use `None` to indicate that a result is not available. However, when the result is a requirement for further calculations, we will usually need to know exactly *which* error prevented the result from being available. The `Either` type may provide detailed information about such errors, which `Option` cannot do.

The `Either` type generalizes the type `Result` defined in Section 3.2.1 to an arbitrary error type instead of `String`. We have seen its usage in Example 3.2.2.4, where the code pattern was "if value is present, do a computation, otherwise keep the error". This code pattern is implemented by the `map` method of `Either`:

```
1 scala> Right(1).map(_ + 1)
2 res0: Either[Nothing, Int] = Right(2)
3
4 scala> Left[String, Int]("error").map(_ + 1)
5 res1: Either[String, Int] = Left("error")
```

The type `Nothing` was filled in by the Scala compiler because we did not specify the first type parameter of `Right` in line 1.

The methods `flatMap`, `fold`, and `getOrElse` are also defined for `Either`, with the same convention that a `Left` value represents an error.[1]

Exceptions and the `Try` type When computations fail for any reason, Scala generates an **exception** instead of returning a value. An exception means that the evaluation of some expression was stopped without returning a result.

As an example, exceptions are generated when the available memory is too small to store the resulting data (as we saw in Section 2.6.3), or if a stack overflow occurs during the computation (as we saw in Section 2.2.3). Exceptions may also occur due to programmer's error: when a pattern matching operation fails, when a requested key does not exist in a dictionary, or when the `head` operation is applied to an empty list.

Motivated by these examples, we may distinguish "planned" and "unplanned" exceptions.

A **planned** exception is generated by programmer's code via the `throw` syntax:

```
scala> throw new Exception("This is a test... this is only a test.")
java.lang.Exception: This is a test... this is only a test.
```

The Scala library contains a `throw` operation in various places, such as in the code for applying the `head` method to an empty sequence, as well as in other situations where exceptions are generated due to programmer's errors. These exceptions are generated deliberately and in well-defined situations. Although these exceptions indicate errors, these errors are anticipated in advance and so may be handled by the programmer.

For example, many Java libraries will generate exceptions when function arguments have unexpected values, when a network operation takes too long or a network connection is unexpectedly broken, when a file is not found or cannot be read due to access permissions, and in many other situations. All these exceptions are "planned" because they are generated explicitly by library code such as `throw new FileNotFoundException(...)`. The programmer's code is expected to catch these exceptions, to handle the error, and to continue the evaluation of the program.

An **unplanned** exception is generated by the Java runtime system when critical errors occur, such as an out-of-memory error. It is rare that a programmer writes `val y = f(x)` while *expecting* that an out-of-memory exception will sometimes occur at that point.[2] An unplanned exception indicates a serious and unforeseen problem with memory or another critically important resource, such as the operating system's threads or file handles. Such problems usually cannot be fixed and will prevent the program from running any further. It is reasonable that the program should abruptly stop (or "crash" as programmers say) after such an error.

The use of planned exceptions assumes that the programmer will write code to handle each exception. This assumption makes it significantly harder to write programs correctly: it is hard to figure out and to keep in mind all the possible exceptions that a given library function may `throw` in its code (and in the code of all other libraries on which it depends). Instead of using exceptions for indicating errors, Scala programmers can write functions that return a disjunctive type, such as `Either`, describing both a correct result and an error condition. Users of these functions will *have* to do pattern matching on the result values. This helps programmers to remember and to handle all relevant error situations that the programmers anticipate to encounter.

However, programmers will often need to use Java or Scala libraries that `throw` exceptions. To help write code for these situations, the Scala library contains a helper function called `Try()` and a disjunctive type, also called `Try`. The type `Try[A]` is equivalent to `Either[Throwable, A]`, where `Throwable` is the general type of all exceptions (i.e. values to which a `throw` operation can be applied). The two parts of the disjunctive type `Try[A]` are called `Failure` and `Success[A]` (instead of `Left[Throwable, A]` and `Right[Throwable, A]` in the `Either` type). The function `Try(expr)` will catch all exceptions thrown while the expression `expr` is evaluated. If the evaluation of `expr` succeeds and returns a value `x: A`, the value of `Try(expr)` will be `Success(x)`. Otherwise it will be `Failure(t)`, where `t: Throwable` is the value associated with the generated exception. Here is an example of using `Try`:

[1] These methods are available in Scala 2.12 or a later version.

[2] Just once in the author's experience, an out-of-memory exception had to be anticipated in an Android app.

```
import scala.util.{Try, Success, Failure}

scala> val p = Try("xyz".toInt)
p: Try[Int] = Failure(java.lang.NumberFormatException: For input string: "xyz")

scala> val q = Try("0002".toInt)
q: Try[Int] = Success(2)
```

The code `Try("xyz".toInt)` does not generate any exceptions and will not crash the program. Any computation that may `throw` an exception can be enclosed in a `Try()`, and the exception will be caught and encapsulated within the disjunctive type as a `Failure(...)` value.

The methods `map`, `filter`, `flatMap`, `foldLeft` are defined for the `Try` class similarly to the `Either` type. One additional feature of `Try` is to catch exceptions generated by the function arguments of `map`, `filter`, `flatMap`, and other standard methods:

```
scala> val y = q.map(y => throw new Exception("ouch"))
y: Try[Int] = Failure(java.lang.Exception: ouch)

scala> val z = q.filter(y => throw new Exception("huh"))
z: Try[Int] = Failure(java.lang.Exception: huh)
```

In this example, the values `y` and `z` were computed *successfully* even though exceptions were thrown while the function arguments of `map` and `filter` were evaluated. Further code can use pattern matching on the values `y` and `z` and examine those exceptions. However, it is important that these exceptions were caught and the program did not crash, meaning that further code is *able* to run.

While the standard types `Try` and `Either` will cover many use cases, programmers can also define custom disjunctive types in order to represent all the anticipated failures or errors in the business logic of a particular application. Representing all errors in the types helps assure that the program will not crash because of an exception that we forgot to handle or did not even know about.

3.3 Lists and trees as recursive disjunctive types

Consider this code defining a disjunctive type `NInt`:

```
sealed trait NInt
final case class N1(x: Int)     extends NInt
final case class N2(n: NInt)    extends NInt
```

The type `NInt` has two disjunctive parts, `N1` and `N2`. But the case class `N2` contains a value of type `NInt` as if the type `NInt` were already defined.

A type whose definition uses that same type is called a **recursive type**. The type `NInt` is an example of a recursive disjunctive type.

We might imagine defining a disjunctive type `X` whose parts recursively refer to the same type `X` (and/or to each other) in complicated ways. What kind of data would be represented by such a type `X`, and in what situations would `X` be useful? For instance, the simple definition:

```
final case class Bad(x: Bad)
```

is useless since we cannot create a value of type `Bad` unless we *already have* a value `x` of type `Bad`. This is an example of an infinite loop in type recursion. We will never be able to create values of type `Bad`, which means that the type `Bad` is "void" (has no values, like the special Scala type `Nothing`).

Section 8.5.1 will derive precise conditions under which a recursive type is not void. For now, we will look at the recursive disjunctive types that are used most often: lists and trees.

3.3.1 The recursive type `List`

A list of values of type `A` is either empty, or has one value of type `A`, or two values of type `A`, etc. We can visualize the type `List[A]` as a disjunctive type defined by:

```
sealed trait List[A]
final case class List0[A]()                extends List[A]
final case class List1[A](x: A)            extends List[A]
final case class List2[A](x1: A, x2: A)    extends List[A]
```

```
???                // Need an infinitely long definition.
```

However, this definition is not practical: we cannot define a separate case class for *each* possible length. Instead, we define the type `List[A]` via mathematical induction on the length of the list:

- Base case: empty list, `case class List0[A]()`.
- Inductive step: given a list of a previously defined length, say $\texttt{List}_{n-1}$, define a new case class $\texttt{List}_n$ describing a list with one more element of type `A`. So we could define $\texttt{List}_n = \texttt{(A, List}_{n-1}\texttt{)}$.

Let us try to write this inductive definition as code:

```
sealed trait ListI[A]           // Inductive definition of a list.
final case class List0[A]()                        extends ListI[A]
final case class List1[A](x: A, next: List0[A])   extends ListI[A]
final case class List2[A](x: A, next: List1[A])   extends ListI[A]
???                   // Still need an infinitely long definition.
```

To avoid writing an infinitely long type definition, we use a trick. Note that the definitions of `List1`, `List2`, etc., have a similar form (while `List0` is not similar). To replace the definitions `List1`, `List2`, etc., by a single definition `ListN`, we write the type `ListI[A]` inside the case class `ListN`:

```
sealed trait ListI[A]           // Inductive definition of a list.
final case class List0[A]()                        extends ListI[A]
final case class ListN[A](x: A, next: ListI[A])   extends ListI[A]
```

The type definition has become recursive. For this trick to work, it is important to use `ListI[A]` and not `ListN[A]` inside the case class `ListN[A]`. Otherwise, we would get an infinite loop in type recursion (similarly to `case class Bad` shown before).

Since we obtained the definition of type `ListI[A]` via a trick, let us verify that the code actually defines the disjunctive type we wanted.

To create a value of type `ListI[A]`, we must use one of the two available case classes. Using the first case class, we may create a value `List0()`. Since this empty case class does not contain any values of type `A`, it effectively represents an empty list (the base case of the induction). Using the second case class, we may create a value `ListN(x, next)` where `x` is of type `A` and `next` is an already constructed value of type `ListI[A]`. This represents the inductive step because the case class `ListN` is a named tuple containing `A` and `ListI[A]`. Now, the same consideration recursively applies to constructing the value `next`, which must be either an empty list or a pair containing a value of type `A` and another list. The assumption that the value `next: ListI[A]` is already constructed is equivalent to the inductive assumption that we already have a list of a previously defined length. So, we have verified that `ListI[A]` implements the inductive definition shown above.

Examples of values of type `ListI` are the empty list `List0()`, a one-element list `ListN(x, List0())`, and a two-element list `ListN(x, ListN(y, List0()))`.

To illustrate writing pattern-matching code using this type, let us implement the method `headOption`:

```
def headOption[A]: ListI[A] => Option[A] = {
  case List0()          => None
  case ListN(x, next)   => Some(x)
}
```

The Scala library already defines the type `List[A]` in a different but equivalent way:

```
sealed trait List[A]
final case object Nil extends List[Nothing]
final case class ::[A](head: A, tail: List[A]) extends List[A]
```

Because "operator-like" case class names, such as `::`, support the infix syntax, we may write expressions such as `head :: tail` instead of `::(head, tail)`. This syntax can be also used in pattern matching on `List` values, with code that looks like this:

```
def headOption[A]: List[A] => Option[A] = {
  case Nil            => None
```

```
  case head :: tail   => Some(head)
}
```

Examples of values created using Scala's standard `List` type are the empty list `Nil`, a one-element list `x :: Nil`, and a two-element list `x :: y :: Nil`. The same syntax `x :: y :: Nil` is used both for creating values of type `List` and for pattern matching on such values.

The Scala library also defines the helper function `List()`, so that `List()` is the same as `Nil` and `List(1, 2, 3)` is the same as `1 :: 2 :: 3 :: Nil`. Lists are easier to read in the syntax `List(1, 2, 3)`. Pattern matching may also use that syntax:

```
val x: List[Int] = List(1, 2, 3)

x match {
  case List(a)          => ...
  case List(a, b, c)    => ...
  case _                => ...
}
```

3.3.2 Tail recursion with `List`

Because the `List` type is defined by induction, it is straightforward to implement iterative computations with the `List` type using recursion.

A first example is the `map` function. We use reasoning by induction in order to figure out the implementation of `map`. The required type signature is

```
def map[A, B](xs: List[A])(f: A => B): List[B] = ???
```

The base case is an empty list, and we return again an empty list:

```
def map[A, B](xs: List[A])(f: A => B): List[B] = xs match {
  case Nil => Nil
  ...
```

In the inductive step, we have a pair `(head, tail)` in the case class `::`, with `head:A` and `tail:List[A]`. The pair can be pattern-matched with the syntax `head :: tail`. The `map` function should apply the argument `f` to the head value, which will give the first element of the resulting list. The remaining elements are computed by the induction assumption, i.e. by a recursive call to `map`:

```
def map[A, B](xs: List[A])(f: A => B): List[B] = xs match {
  case Nil           => Nil
  case head :: tail  => f(head) :: map(tail)(f) // Not tail-recursive.
```

While this implementation is straightforward and concise, it is not tail-recursive. This will be a problem for large enough lists.

Instead of implementing the often-used methods such as `map` or `filter` one by one, let us implement `foldLeft`, because most of the other methods can be expressed via `foldLeft`.

The required type signature is:

```
def foldLeft[A, R](xs: List[A])(init: R)(f: (R, A) => R): R = ???
```

Reasoning by induction, we start with the base case `xs == Nil`, where the only possibility is to return the value `init`:

```
def foldLeft[A, R](xs: List[A])(init: R)(f: (R, A) => R): R = xs match {
    case Nil            => init
    ...
```

The inductive step for `foldLeft` says that, given the values `head:A` and `tail:List[A]`, we need to apply the updater function to the previous accumulator value. That value is `init`. So we apply `foldLeft` recursively to the tail of the list once we have the updated accumulator value:

```
@tailrec def foldLeft[A, R](xs: List[A])(init: R)(f: (R, A) => R): R =
```

```
  xs match {
    case Nil            => init
    case head :: tail   =>
      val newInit = f(init, head) // Update the accumulator.
      foldLeft(tail)(newInit)(f)  // Recursive call to ‘foldLeft‘.
  }
```

This implementation is tail-recursive because the recursive call to `foldLeft` is the last expression returned in a `case` branch.

Another example is a function for reversing a list. The Scala library defines the `reverse` method for this task, but we will show an implementation using `foldLeft`. The updater function *prepends* an element to a previous list:

```
def reverse[A](xs: List[A]): List[A] =
  xs.foldLeft(Nil: List[A])((prev, x) => x :: prev)

scala> reverse(List(1, 2, 3))
res0: List[Int] = List(3, 2, 1)
```

Without the explicit type annotation `Nil:List[A]`, the Scala compiler will decide that `Nil` has type `List[Nothing]`, and the types will not match later in the code. In Scala, one often finds that the initial value for `foldLeft` needs an explicit type annotation.

The `reverse` function can be used to obtain a tail-recursive implementation of `map` for `List`. The idea is to first use `foldLeft` to accumulate transformed elements:

```
scala> Seq(1, 2, 3).foldLeft(Nil:List[Int])((prev, x) => x*x :: prev)
res0: List[Int] = List(9, 4, 1)
```

To obtain the correct result (`List(1, 4, 9)`), it remains to apply `reverse`:

```
def map[A, B](xs: List[A])(f: A => B): List[B] =
  xs.foldLeft(Nil: List[B])((prev, x) => f(x) :: prev).reverse

scala> map(List(1, 2, 3))(x => x*x)
res2: List[Int] = List(1, 4, 9)
```

This achieves stack safety at the cost of traversing the list twice. (This code is shown only as an example. The Scala library implements `List`'s `map` using low-level tricks for better performance.)

Example 3.3.2.1 A definition of the **non-empty list** is similar to `List` except that the empty-list case is replaced by a 1-element case:

```
sealed trait NEL[A]
final case class Last[A](head: A)                 extends NEL[A]
final case class More[A](head: A, tail: NEL[A])   extends NEL[A]
```

Values of a non-empty list look like this:

```
scala> val xs: NEL[Int] = More(1, More(2, Last(3)))  // [1, 2, 3]
xs: NEL[Int] = More(1,More(2,Last(3)))

scala> val ys: NEL[String] = Last("abc") // One element, ["abc"].
ys: NEL[String] = Last(abc)
```

To create non-empty lists more easily, we implement a conversion function `toNEL` from an ordinary list. To guarantee that a non-empty list can be created, we give `toNEL` *two* arguments:

```
def toNEL[A](x: A, rest: List[A]): NEL[A] = rest match {
  case Nil         => Last(x)
  case y :: tail   => More(x, toNEL(y, tail))
}   // Not tail-recursive: ‘toNEL()‘ is used inside ‘More(...)‘.
```

To test:

```
scala> toNEL(1, List()) // Result = [1].
res0: NEL[Int] = Last(1)
```

```
scala> toNEL(1, List(2, 3)) // Result = [1, 2, 3].
res1: NEL[Int] = More(1,More(2,Last(3)))
```

The `head` method is safe for non-empty lists, unlike `head` for an ordinary `List`:

```
def head[A]: NEL[A] => A = {
    case Last(x)        => x
    case More(x, _)     => x
}
```

We can also implement a tail-recursive `foldLeft` function for non-empty lists:

```
@tailrec def foldLeft[A, R](n: NEL[A])(init: R)(f: (R, A) => R): R = n match {
    case Last(x)        => f(init, x)
    case More(x, tail)  => foldLeft(tail)(f(init, x))(f)
  }

scala> foldLeft(More(1, More(2, Last(3))))(0)(_ + _)
res2: Int = 6
```

Example 3.3.2.2 Use `foldLeft` to implement a `reverse` function for the type `NEL`. The required type signature and a sample test:

```
def reverse[A]: NEL[A] => NEL[A] = ???

scala> reverse(toNEL(10, List(20, 30))) // Result must be [30, 20, 10].
res3: NEL[Int] = More(30,More(20,Last(10)))
```

Solution We will use `foldLeft` to build up the reversed list as the accumulator value. It remains to choose the initial value of the accumulator and the updater function. We have already seen the code for reversing the ordinary list via the `foldLeft` method (Section 3.3.2):

```
def reverse[A](xs: List[A]): List[A] = xs.foldLeft(Nil: List[A])((prev,x) => x :: prev)
```

However, we cannot reuse the same code for non-empty lists by writing `More(x, prev)` instead of `x :: prev`, because the `foldLeft` operation works with non-empty lists differently. Since lists are always non-empty, the updater function is always applied to an initial value, and the code works incorrectly:

```
def reverse[A](xs: NEL[A]): NEL[A] =
  foldLeft(xs)(Last(head(xs)):NEL[A])((prev,x) => More(x, prev))

scala> reverse(toNEL(10,List(20,30))) // Result = [30, 20, 10, 10].
res4: NEL[Int] = More(30,More(20,More(10,Last(10))))
```

The last element, `10`, should not have been repeated. It was repeated because the initial accumulator value already contained the head element `10` of the original list. However, we cannot set the initial accumulator value to an empty list, since a value of type `NEL[A]` must be non-empty. It seems that we need to handle the case of a one-element list separately. So we begin by matching on the argument of `reverse`, and apply `foldLeft` only when the list is longer than 1 element:

```
def reverse[A]: NEL[A] => NEL[A] = {
    case Last(x)         => Last(x)    // ‘reverse‘ is trivial.
    case More(x, tail)   =>            // Use foldLeft on ‘tail‘.
      foldLeft(tail)(Last(x):NEL[A])((prev,x) => More(x, prev))
  }

scala> reverse(toNEL(10, List(20, 30))) // Result = [30, 20, 10].
res5: NEL[Int] = More(30,More(20,Last(10)))
```

Exercise 3.3.2.3 Implement a function `toList` that converts a non-empty list into an ordinary Scala `List`. The required type signature and a sample test:

```
def toList[A](nel: NEL[A]): List[A] = ???

scala> toList(More(1, More(2, Last(3)))) // This is [1, 2, 3].
```

```
res6: List[Int] = List(1, 2, 3)
```

Exercise 3.3.2.4 Implement a `map` function for the type `NEL`. Type signature and a sample test:

```
def map[A,B](xs: NEL[A])(f: A => B): NEL[B] = ???

scala> map[Int, Int](toNEL(10, List(20, 30)))(_ + 5) // Result = [15, 25, 35].
res7: NEL[Int] = More(15,More(25,Last(35)))
```

Exercise 3.3.2.5 Implement a function `concat` that concatenates two non-empty lists:

```
def concat[A](xs: NEL[A], ys: NEL[A]): NEL[A] = ???

scala> concat(More(1, More(2, Last(3))), More(4, Last(5))) // Result is [1, 2, 3, 4, 5].
res8: NEL[Int] = More(1,More(2,More(3,More(4,Last(5)))))
```

3.3.3 Binary trees

We will consider four kinds of trees defined as recursive disjunctive types: binary trees, rose trees, perfect-shaped trees, and abstract syntax trees.

Examples of a **binary tree** with leaves of type `A` can be drawn as

or as

,

where a_i are some values of type `A`.

An inductive definition says that a binary tree is either a leaf with a value of type `A` or a branch containing *two* previously defined binary trees. Translating this definition into code, we get:

```
sealed trait Tree2[A]
final case class Leaf[A](a: A)                          extends Tree2[A]
final case class Branch[A](x: Tree2[A], y: Tree2[A])    extends Tree2[A]
```

Here are some examples of code expressions and the corresponding trees that use this definition:

```
Branch(Branch(Leaf("a1"), Leaf("a2")), Leaf("a3"))
```

```
Branch(Branch(Leaf("a1"), Branch(Leaf("a2"), Leaf("a3"))),
  Branch(Leaf("a4"), Leaf("a5")))
```

Recursive functions on trees are translated into concise code. For instance, the function `foldLeft` for trees of type `Tree2` is implemented as:

```
def foldLeft[A, R](t: Tree2[A])(init: R)(f: (R, A) => R): R = t match {
  case Leaf(a)          => f(init, a)
  case Branch(t1, t2)   =>
    val r1 = foldLeft(t1)(init)(f)   // Fold the left branch and obtain the result ‘r1‘.
    foldLeft(t2)(r1)(f)              // Using ‘r1‘ as the ‘init‘ value, fold the right branch.
}
```

Note that this function *cannot* be made tail-recursive using the accumulator trick, because `foldLeft` needs to call itself twice in the `Branch` case.

To verify that `foldLeft` works as intended, let us run a simple test:

```
val t: Tree2[String] = Branch(Branch(Leaf("a1"), Leaf("a2")), Leaf("a3"))

scala> foldLeft(t)("")(_ + " " + _)
res0: String = " a1 a2 a3"
```

3.3.4 Rose trees

A **rose tree** is similar to the binary tree except the branches contain a non-empty list of trees. Because of that, a rose tree can fork into arbitrarily many branches at each node, rather than always into two branches as the binary tree does. For example,

and

are rose trees.

A possible definition of a data type for the rose tree is:

```
sealed trait TreeN[A]
final case class Leaf[A](a: A)                  extends TreeN[A]
final case class Branch[A](ts: NEL[TreeN[A]])   extends TreeN[A]
```

Since we used a non-empty list `NEL`, a `Branch()` value is guaranteed to have at least one branch. If we used an ordinary `List` instead, we could (by mistake) create a tree with no leaves and no branches.

Exercise 3.3.4.1 Define the function `foldLeft` for a rose tree, using `foldLeft` for the type `NEL`. Type signature and a test:

```
def foldLeft[A, R](t: TreeN[A])(init: R)(f: (R, A) => R): R = ???

scala> foldLeft(Branch(More(Leaf(1), More(Leaf(2), Last(Leaf(3))))))(0)(_ + _)
res0: Int = 6
```

3.3.5 Perfect-shaped trees

Binary trees and rose trees may choose to branch or not to branch at any given node, resulting in structures that may have different branching depths at different nodes, such as

. A **perfect-shaped tree** always branches in the same way at every node until a chosen total depth, e.g.,

, where all nodes at depth 0 and 1 always branch into two, while nodes at depth 2 do not branch. The branching number is fixed for a given type of a perfect-shaped tree; in this example, the branching number is 2, so it is a perfect-shaped *binary* tree.

How can we define a data type representing a perfect-shaped binary tree? We need a type that is either a single value, or a pair of values, or a pair of pairs, etc. Begin with the non-recursive (but, of course, impractical) definition:

```
sealed trait PTree[A]
final case class Leaf[A](x: A)                      extends PTree[A]
final case class Branch1[A](xs: (A, A))             extends PTree[A]
final case class Branch2[A](xs: ((A, A),(A, A)))    extends PTree[A]
???                          // Need an infinitely long definition.
```

The case `Branch1` describes a perfect-shaped tree with total depth 1, the case `Branch2` has total depth 2, and so on. Now, we cannot rewrite this definition as a recursive type because the case classes do not have the same structure. The non-trivial trick is to notice that each case class $\texttt{Branch}_n$ uses the previous case class's data structure with the *type parameter* set to `(A, A)` instead of `A`. So we can rewrite this definition as:

```
sealed trait PTree[A]
final case class Leaf[A](x: A)                      extends PTree[A]
final case class Branch1[A](xs: Leaf[(A, A)])       extends PTree[A]
final case class Branch2[A](xs: Branch1[(A, A)])    extends PTree[A]
???                          // Need an infinitely long definition.
```

We can now apply the type recursion trick: replace the type $\texttt{Branch}_{n-1}\texttt{[(A, A)]}$ in the definition of $\texttt{Branch}_n$ by the recursively used type `PTree[(A, A)]`. Now we can define a perfect-shaped binary tree:

```
sealed trait PTree[A]
final case class Leaf[A](x: A)                    extends PTree[A]
final case class Branch[A](xs: PTree[(A, A)])    extends PTree[A]
```

Since we used some tricks to figure out the definition of `PTree[A]`, let us verify that this definition actually describes the recursive disjunctive type we wanted. The only way to create a structure of type `PTree[A]` is to create a `Leaf[A]` or a `Branch[A]`. A value of type `Leaf[A]` is a correct regularly-shaped tree. It remains to consider the case of `Branch[A]`. Creating a `Branch[A]` requires a previously created `PTree` with values of type `(A, A)` instead of `A`. By the inductive assumption, the previously created `PTree[A]` would have the correct shape. Now, it is clear that if we replace the type parameter `A` by the pair `(A, A)`, a perfect-shaped tree such as

remains perfect-shaped but becomes one level deeper, which can be drawn (replacing each a_i by a pair $a_i^{'}, a_i^{''}$) as

.

We see that `PTree[A]` is a correct definition of a perfect-shaped binary tree.

Example 3.3.5.1 Define a (non-tail-recursive) `map` function for a perfect-shaped binary tree. The required type signature and a test:

```
def map[A, B](t: PTree[A])(f: A => B): PTree[B] = ???

scala> map(Branch(Branch(Leaf(((1,2),(3,4))))))(_ * 10)
res0: PTree[Int] = Branch(Branch(Leaf(((10,20),(30,40)))))
```

Solution Begin by pattern matching on the tree:

```
def map[A, B](t: PTree[A])(f: A => B): PTree[B] = t match {
  case Leaf(x)      => ???
  case Branch(xs)   => ???
}
```

In the base case, we have no choice but to return `Leaf(f(x))`:

```
def map[A, B](t: PTree[A])(f: A => B): PTree[B] = t match {
  case Leaf(x)      => Leaf(f(x))
  case Branch(xs)   => ???
}
```

In the inductive step, we are given a previous tree value `xs:PTree[(A, A)]`. It is clear that we need to apply `map` recursively to `xs`. Let us try:

```
def map[A, B](t: PTree[A])(f: A => B): PTree[B] = t match {
  case Leaf(x)      => Leaf(f(x))
  case Branch(xs)   => Branch(map(xs)(f))    // Type error!
}
```

Here, `map(xs)(f)` has an incorrect type of the function `f`. Since `xs` has type `PTree[(A, A)]`, the recursive call `map(xs)(f)` requires `f` to be of type `((A, A)) => (B, B)` instead of `A => B`. So, we need to provide a function of the correct type instead of `f`. A function of type `((A, A)) => (B, B)` will be obtained out of `f: A => B` if we apply `f` to each part of the tuple `(A, A)`. The code for that function is `{ case (x, y) => (f(x), f(y)) }`. Therefore, we can implement `map` as:

```
def map[A, B](t: PTree[A])(f: A => B): PTree[B] = t match {
  case Leaf(x)      => Leaf(f(x))
  case Branch(xs)   => Branch(map(xs){ case (x, y) => (f(x), f(y)) })
}
```

This code is not tail-recursive since it calls `map` inside an expression.

Exercise 3.3.5.2 Using tail recursion, compute the depth of a perfect-shaped binary tree of type `PTree`. (An `PTree` of depth n has 2^n leaf values.) The required type signature and a test:

```
@tailrec def depth[A](t: PTree[A]): Int = ???

scala> depth(Branch(Branch(Leaf((("a","b"),("c","d"))))))
res2: Int = 2
```

Exercise 3.3.5.3* Define a tail-recursive function `foldLeft` for a perfect-shaped binary tree. The required type signature and a test:

```
@tailrec def foldLeft[A, R](t: PTree[A])(init: R)(f: (R, A) => R): R = ???

scala> foldLeft(Branch(Branch(Leaf(((1,2),(3,4))))))(0)(_ + _)
res0: Int = 10

scala> foldLeft(Branch(Branch(Leaf((("a","b"),("c","d"))))))("")(_ + _)
res1: String = abcd
```

3.3.6 Abstract syntax trees

Expressions in formal languages are represented by abstract syntax trees. An **abstract syntax tree** (or **AST** for short) is defined as either a leaf of one of the available leaf types, or a branch of one of the available branch types. All the available leaf and branch types must be specified as part of the definition of an AST. In other words, one must specify the data carried by leaves and branches, as well as the branching numbers.

To illustrate how ASTs are used, let us rewrite Example 3.2.2.4 via an AST. We view Example 3.2.2.4 as a small sub-language that deals with "safe integers" and supports the "safe arithmetic" operations `Sqrt`, `Add`, `Mul`, and `Div`. Example calculations in this sub-language are $\sqrt{16} * (1+2) = 12$; $20 + 1/0 =$ error; and $10 + \sqrt{-1} =$ error.

We can implement this sub-language in two stages. The first stage will create a data structure (an AST) that represents an unevaluated expression in the sub-language. The second stage will evaluate that AST to obtain either a number or an error message.

A straightforward way of defining the data structure for the AST is to use a disjunctive type whose parts describe all the possible operations of the sub-language. We will need one case class for each of `Sqrt`, `Add`, `Mul`, and `Div`. An additional operation, `Num`, will lift ordinary integers into "safe integers". So, we define the disjunctive type (`Arith`) for the "safe arithmetic" sub-language as:

```
sealed trait Arith
final case class Num(x: Int)                extends Arith
final case class Sqrt(x: Arith)             extends Arith
final case class Add(x: Arith, y: Arith)    extends Arith
final case class Mul(x: Arith, y: Arith)    extends Arith
final case class Div(x: Arith, y: Arith)    extends Arith
```

A value of type `Arith` is either a `Num(x)` for some integer `x`, or an `Add(x, y)` where `x` and `y` are previously defined `Arith` expressions, or another operation.

This type definition is similar to the binary tree type if we rename `Leaf` to `Num` and `Branch` to `Add`:

```
sealed trait Tree
final case class Leaf(x: Int)               extends Tree
final case class Branch(x: Tree, y: Tree)   extends Tree
```

However, the `Arith` type is a tree that supports four different types of branches, some with branching number 1 and others with branching number 2.

This example illustrates the structure of an AST: it is a tree of a specific shape, with leaves and branches chosen from a specified set of allowed possibilities. In the "safe arithmetic" example, we have a single allowed type of leaf (`Num`) and four allowed types of branches (`Sqrt`, `Add`, `Mul`, and `Div`).

This completes the first stage of implementing the sub-language. We may now use the disjunctive type `Arith` to create expressions in the sub-language. For example, $\sqrt{16} * (1+2)$ is represented by:

```
scala> val x: Arith = Mul(Sqrt(Num(16)), Add(Num(1), Num(2)))
x: Arith = Mul(Sqrt(Num(16)),Add(Num(1),Num(2)))
```

We can visualize `x` as the abstract syntax tree

.

The expressions $20 + 1/0$ and $10 * \sqrt{-1}$ are represented by:

```
scala> val y: Arith = Add(Num(20), Div(Num(1), Num(0)))
y: Arith = Add(Num(20),Div(Num(1),Num(0)))

scala> val z: Arith = Add(Num(10), Sqrt(Num(-1)))
z: Arith = Add(Num(10),Sqrt(Num(-1)))
```

As we see, the expressions `x`, `y`, and `z` *remain unevaluated*; each of them is a data structure that encodes a tree of operations of the sub-language. These operations will be evaluated at the second stage of implementing the sub-language.

To evaluate expressions in the "safe arithmetic", we can implement a function with type signature `run: Arith => Either[String, Int]`. That function plays the role of an **interpreter** or "**runner**" for programs written in the sub-language. The runner will walk through the expression tree and execute all the operations, taking care of possible errors.

To implement `run`, we need to define required arithmetic operations on the type `Either[String, Int]`. For instance, we need to be able to add or multiply values of that type. Instead of custom code from Example 3.2.2.4, we can use the standard `map` and `flatMap` methods defined on `Either`. For example, addition and multiplication of two "safe integers" is implemented as:

```
def add(x: Either[String, Int], y: Either[String, Int]):
    Either[String, Int] = x.flatMap { r1 => y.map(r2 => r1 + r2) }
def mul(x: Either[String, Int], y: Either[String, Int]):
    Either[String, Int] = x.flatMap { r1 => y.map(r2 => r1 * r2) }
```

The code for the "safe division" is:

```
def div(x: Either[String, Int], y: Either[String, Int]):
    Either[String, Int] = x.flatMap { r1 => y.flatMap(r2 =>
  if (r2 == 0) Left(s"error: $r1 / $r2") else Right(r1 / r2) )
}
```

With this code, we can implement the runner as a recursive function:

```
def run: Arith => Either[String, Int] = {
  case Num(x)     => Right(x)
  case Sqrt(x)    => run(x).flatMap { r =>
    if (r < 0) Left(s"error: sqrt($r)") else Right(math.sqrt(r).toInt)
  }
  case Add(x, y)  => add(run(x), run(y))
  case Mul(x, y)  => mul(run(x), run(y))
  case Div(x, y)  => div(run(x), run(y))
}
```

Test it with the values `x`, `y`, `z` defined previously:

```
scala> run(x)
res0: Either[String, Int] = Right(12)

scala> run(y)
res1: Either[String, Int] = Left("error: 1 / 0")

scala> run(z)
res2: Either[String, Int] = Left("error: sqrt(-1)")
```

3.4 Summary

What problems can we solve now?

- Represent values from a disjoint domain by a custom-defined disjunctive type.
- Use disjunctive types instead of exceptions to indicate failures.
- Use standard disjunctive types `Option`, `Try`, `Either` and their standard methods.
- Define recursive disjunctive types (e.g., lists and trees) and implement recursive functions that work with them.

The following examples and exercises illustrate these tasks.

3.4.1 Solved examples

Example 3.4.1.1 Define a disjunctive type `DayOfWeek` representing the seven days of a week.

Solution Since each day carries no information except the day's name, we can use empty case classes and represent the day's name via the name of the case class:

```
sealed trait DayOfWeek
final case class Sunday()    extends DayOfWeek
final case class Monday()    extends DayOfWeek
final case class Tuesday()   extends DayOfWeek
final case class Wednesday() extends DayOfWeek
final case class Thursday()  extends DayOfWeek
final case class Friday()    extends DayOfWeek
final case class Saturday()  extends DayOfWeek
```

This data type is analogous to an enumeration type in C or C++:

```
typedef enum { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday } DayOfWeek;
```

Example 3.4.1.2 Modify `DayOfWeek` so that the values additionally represent a restaurant name and total amount for Fridays and a wake-up time on Saturdays.

Solution For the days where additional information is given, we use non-empty case classes:

```
sealed trait DayOfWeekX
final case class Sunday()                                extends DayOfWeekX
final case class Monday()                                extends DayOfWeekX
final case class Tuesday()                               extends DayOfWeekX
final case class Wednesday()                             extends DayOfWeekX
final case class Thursday()                              extends DayOfWeekX
final case class Friday(restaurant: String, amount: Int) extends DayOfWeekX
final case class Saturday(wakeUpAt: java.time.LocalTime) extends DayOfWeekX
```

This data type is no longer equivalent to an enumeration type.

Example 3.4.1.3 Define a disjunctive type that describes the real roots of the equation $ax^2 + bx + c = 0$, where a, b, c are arbitrary real numbers.

Solution Begin by solving the equation and enumerating all the possible cases. It may happen that $a = b = c = 0$, and then all x are roots. If $a = b = 0$ but $c \neq 0$, the equation is $c = 0$, which has no roots. If $a = 0$ but $b \neq 0$, the equation becomes $bx + c = 0$, having a single root. If $a \neq 0$ and $b^2 > 4ac$, we have two distinct real roots. If $a \neq 0$ and $b^2 = 4ac$, we have one real root. If $b^2 < 4ac$, we have no real roots. The resulting type definition can be written as:

```
sealed trait RootsOfQ2
final case class AllRoots()                    extends RootsOfQ2
final case class ConstNoRoots()                extends RootsOfQ2
final case class Linear(x: Double)             extends RootsOfQ2
final case class NoRealRoots()                 extends RootsOfQ2
final case class OneRootQ(x: Double)           extends RootsOfQ2
```

```
final case class TwoRootsQ(x: Double, y: Double) extends RootsOfQ2
```

This disjunctive type contains six parts, among which three parts are empty tuples and two parts are single-element tuples; but this is not a useless redundancy. We would lose information if we reused `Linear` for the two cases $a = 0$, $b \neq 0$ and $a \neq 0$, $b^2 = 4ac$, or if we reused `NoRoots()` for representing all three different no-roots cases.

Example 3.4.1.4 Define a function `rootAverage` that computes the average value of all real roots of a general quadratic equation, where the set of roots is represented by the type `RootsOfQ2` defined in Example 3.4.1.3. The required type signature is:

```
val rootAverage: RootsOfQ2 => Option[Double] = ???
```

The function should return `None` if the average is undefined.

Solution The average is defined only in cases `Linear`, `OneRootQ`, and `TwoRootsQ`. In all other cases, we must return `None`. We implement this via pattern matching:

```
val rootAverage: RootsOfQ2 => Option[Double] = roots => roots match {
    case Linear(x)       => Some(x)
    case OneRootQ(x)     => Some(x)
    case TwoRootsQ(x, y) => Some((x + y) * 0.5)
    case _               => None
  }
```

We do not need to enumerate all other cases since the underscore (_) matches everything that the previous cases did not match.

In Scala, the often-used code pattern `x => x match { case ... => ... }` can be shortened to just the nameless function `{ case ... => ... }`. The code then becomes:

```
val rootAverage: RootsOfQ2 => Option[Double] = {
  case Linear(x)       => Some(x)
  case OneRootQ(x)     => Some(x)
  case TwoRootsQ(x, y) => Some((x + y) * 0.5)
  case _               => None
}
```

Test it:

```
scala> Seq(NoRealRoots(),OneRootQ(1.0), TwoRootsQ(1.0, 2.0), AllRoots()).map(rootAverage)
res0: Seq[Option[Double]] = List(None, Some(1.0), Some(1.5), None)
```

Example 3.4.1.5 Generate 100 quadratic equations $x^2 + bx + c = 0$ with random coefficients b, c (uniformly distributed between -1 and 1) and compute the mean of the largest real roots from all these equations.

Solution Use the type `QEqu` and the `solve` function from Example 3.2.2.1. A sequence of equations with random coefficients is created by applying the method `Seq.fill`:

```
def random(): Double = scala.util.Random.nextDouble() * 2 - 1
val coeffs: Seq[QEqu] = Seq.fill(100)(QEqu(random(), random()))
```

Now we can use the `solve` function to compute all roots:

```
val solutions: Seq[RootsOfQ] = coeffs.map(solve)
```

For each set of roots, compute the largest root:

```
scala> val largest: Seq[Option[Double]] = solutions.map {
  case OneRoot(x)     => Some(x)
  case TwoRoots(x, y) => Some(math.max(x, y))
  case _              => None
}
largest: Seq[Option[Double]] = List(None, Some(0.9346072365885472), Some(1.1356234869160806),
     Some(0.9453181931646322), Some(1.1595052441078866), None, Some(0.5762252742788)...
```

It remains to remove the `None` values and to compute the mean of the resulting sequence. The Scala library defines the `flatten` method that removes `None`s and transforms `Seq[Option[A]]` into `Seq[A]`:

```
scala> largest.flatten
res0: Seq[Double] = List(0.9346072365885472, 1.1356234869160806, 0.9453181931646322,
     1.1595052441078866, 0.5762252742788...
```

Now compute the mean of the last sequence. Since the `flatten` operation is preceded by `map`, we can replace it by a `flatMap`. The final code is:

```
val largest = Seq.fill(100)(QEqu(random(), random()))
  .map(solve)
  .flatMap {
    case OneRoot(x)      => Some(x)
    case TwoRoots(x, y) => Some(math.max(x, y))
    case _               => None
  }

scala> largest.sum / largest.size
res1: Double = 0.7682649774589514
```

Example 3.4.1.6 Implement a function with type signature:

```
def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = ???
```

The function should preserve information as much as possible.

Solution Begin by pattern matching on the argument:

```
1 def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = {
2   case None                      => ???
3   case Some(eab: Either[A, B])  => ???
4 }
```

In line 3, we wrote the **type annotation** `eab: Either[A, B]` only for clarity. It is not required here since the Scala compiler can deduce the type of the pattern variable `eab` from the fact that we are matching a value of type `Option[Either[A, B]]`.

In the scope of line 2, we need to return a value of type `Either[A, Option[B]]`. A value of that type must be either a `Left(x)` for some `x:A`, or a `Right(y)` for some `y:Option[B]`, where `y` must be either `None` or `Some(z)` with a `z:B`. However, in our case the code is of the form `case None => ???`, and we cannot produce any values `x:A` or `z:B` since `A` and `B` are arbitrary, unknown types. The only remaining possibility is to return `Right(y)` with `y = None`, and so the code must be:

```
  ...
  case None => Right(None) // No other choice here.
```

In the next scope, we can perform pattern matching on the value `eab`:

```
  ...
  case Some(eab: Either[A, B]) = eab match {
    case Left(a)   => ???
    case Right(b)  => ???
  }
```

It remains to figure out what expressions to write in each case. In the case `Left(a) => ???`, we have a value of type `A`, and we need to compute a value of type `Either[A, Option[B]]`. We execute the same argument as before: The return value must be `Left(x)` for some `x:A`, or `Right(y)` for some `y:Option[B]`. At this point, we have a value of type `A` but no values of type `B`. So we have two possibilities: to return `Left(a)` or to return `Right(None)`. If we decide to return `Left(a)`, the code is:

```
1 def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = {
2   case None       => Right(None) // No other choice here.
3   case Some(eab)  => eab match {
4     case Left(a)  => Left(a)   // Could return Right(None) here.
5     case Right(b) => ???
6   }
7 }
```

Let us decide whether to return `Left(a)` or `Right(None)` in line 4. Both choices will satisfy the required return type `Either[A, Option[B]]`. However, if we return `Right(None)` in that line, we will ignore the given value `a:A`, i.e., we will lose information. So we return `Left(a)` in line 4.

Reasoning similarly for line 5, we find that we may return `Right(None)` or `Right(Some(b))`. The first

choice ignores the given value of `b:B`. To preserve information, we make the second choice:

```
def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = {
  case None        => Right(None)
  case Some(eab)   => eab match {
      case Left(a)     => Left(a)
      case Right(b)    => Right(Some(b))
  }
}
```

We can now refactor this code into a somewhat more readable form by using nested patterns:

```
def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = {
  case None              => Right(None)
  case Some(Left(a))     => Left(a)
  case Some(Right(b))    => Right(Some(b))
}
```

Example 3.4.1.7 Implement a function with the type signature:

```
def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = ???
```

The function should preserve information as much as possible.

Solution Begin by pattern matching on the argument:

```
1 def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = {
2   case (Some(a), Some(b)) => ???
3   ...
```

In line 2, we have values `a:A` and `b:B`, and we need to compute a value of type `Option[(A, B)]`. A value of that type is either `None` or `Some((x, y))` where we would need to obtain values `x:A` and `y:B`. Since `A` and `B` are arbitrary types, we cannot produce new values `x` and `y` from scratch. The only way of obtaining `x:A` and `y:B` is to set `x = a` and `y = b`. So, our choices are to return `Some((a, b))` or `None`. We reject returning `None` since that would unnecessarily lose information. Thus, we continue writing code as:

```
1 def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = {
2   case (Some(a), Some(b)) => Some((a, b))
3   case (Some(a), None)    => ???
4   ...
```

In line 3, we have a value `a:A` but no values of type `B`. Since the type `B` is arbitrary, we cannot produce any values of type `B` to return a value of the form `Some((x, y))`. So, `None` is the only computable value of type `Option[(A, B)]` in line 3. We continue to write the code:

```
1 def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = {
2   case (Some(a), Some(b)) => Some((a, b))
3   case (Some(a), None)    => None // No other choice here.
4   case (None, Some(b))    => ???
5   case (None, None)       => ???
6 }
```

In lines 4–5, we find that there is no choice other than returning `None`. So we can simplify the code:

```
def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = {
  case (Some(a), Some(b)) => Some((a, b))
  case _                  => None // No other choice here.
}
```

3.4.2 Exercises

Exercise 3.4.2.1 Define a disjunctive type `CellState` representing the visual state of one cell in the "Minesweeper"[3] game: A cell can be closed (showing nothing), or show a bomb, or be open and show the number of bombs in neighbor cells.

[3] https://en.wikipedia.org/wiki/Minesweeper_(video_game)

Exercise 3.4.2.2 In the "Minesweeper" game, count the total number of cells with zero neighbor bombs shown by implementing a function with type signature `Seq[Seq[CellState]] => Int`.

Exercise 3.4.2.3 Define a disjunctive type `RootOfLinear` representing all possibilities for the solution of the equation $ax + b = 0$ for arbitrary real a, b. (The possibilities are: no roots; one root; all x are roots.) Implement the solution as a function `solve1` with type signature

```
def solve1: ((Double, Double)) => RootOfLinear = ???
```

Exercise 3.4.2.4 Given a `Seq[(Double, Double)]` containing pairs (a, b) of the coefficients of $ax + b = 0$, compute a `Seq[Double]` containing the roots of that equation when a unique root exists. Use the type `RootOfLinear` and the function `solve1` defined in Exercise 3.4.2.3.

Exercise 3.4.2.5 The case class `Subscriber` was defined in Example 3.2.3.1. Given a `Seq[Subscriber]`, compute the sequence of email addresses for all subscribers that did *not* provide a phone number.

Exercise 3.4.2.6* In this exercise, a "procedure" is a function of type `Unit => Unit`. An example of a procedure is `{ _ => println("hello") }`. Define a disjunctive type `Proc` for an abstract syntax tree representing three operations on procedures: 1) `Func[A](f)`, create a procedure from a function `f` of type `Unit => A`, where `A` is a type parameter. (Note that the type `Proc` does not have any type parameters.) 2) `Sequ(p1, p2)`, execute two procedures sequentially. 3) `Para(p1, p2)`, execute two procedures in parallel. Then implement a "runner" that converts a `Proc` into a `Future[Unit]`, running the computations either sequentially or in parallel as appropriate. Test with this code:

```
sealed trait Proc; final case class Func[A](???)    // And so on.
def runner: Proc => Future[Unit] = ???
val proc1: Proc = Func{_ => Thread.sleep(200); println("hello1")}
val proc2: Proc = Func{_ => Thread.sleep(400); println("hello2")}

scala> runner(Sequ(Para(proc2, proc1), proc2))
hello1
hello2
hello2
```

Exercise 3.4.2.7 Use pattern matching to implement functions with given type signatures, preserving information as much as possible:

```
def f1[A, B]: Option[(A, B)] => (Option[A], Option[B]) = ???
def f2[A, B]: Either[A, B] => (Option[A], Option[B]) = ???
def f3[A,B,C]: Either[A, Either[B,C]] => Either[Either[A,B], C] = ???
```

Exercise 3.4.2.8 Define a parameterized type `EvenList[A]` representing a list of values of type `A` that must have an even length (zero, two, four, etc.). Implement `foldLeft` and `map` for `EvenList`.

Exercise 3.4.2.9* The standard type `List[A]` requires all its values to have the same type `A`. Define a parameterized type `ListX[A]` representing a data structure in the form of a list with one or more values where the first value has type `A`, the second value has type `Option[A]`, the third — `Option[Option[A]]`, and so on. Using a wrong type at a given place (say, `Option[Option[A]]` instead of `Option[A]` for the second value in the list) should cause a type error. Implement functions `map` and `foldLeft` for `ListX`.

3.5 Discussion and further developments

3.5.1 Disjunctive types as mathematical sets

To understand the properties of disjunctive types from the mathematical point of view, consider a function whose argument is a disjunctive type, such as:

```
def isDoubleRoot(r: RootsOfQ) = ...
```

The type of the argument `r:RootsOfQ` represents the mathematical domain of the function, that is, the set of admissible values of the argument `r`. What kind of domain is that? The set of real roots of a

Figure 3.1: The disjoint domain represented by the type `RootsOfQ`.

quadratic equation $x^2 + bx + c = 0$ can be empty, or it can contain a single real number x, or a pair of real numbers (x, y). Geometrically, a number x is pictured as a point on a line (a one-dimensional space), and pair of numbers (x, y) is pictured as a point on a Cartesian plane (a two-dimensional space). The no-roots case corresponds to a zero-dimensional space, which is pictured as a single point in Figure 3.1. The point, the line, and the plane do not intersect (i.e., have no common points. Together, they form the set of the possible roots of the quadratic equation $x^2 + bx + c = 0$.

In the mathematical notation, a one-dimensional real space is denoted by $\mathbb{R}$, a two-dimensional space by $\mathbb{R}^2$, and a zero-dimensional space by $\mathbb{R}^0$. At first, we may think that the mathematical representation of the type `RootsOfQ` is a union of the three sets, $\mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2$. But an ordinary union of sets would not always work correctly because we need to distinguish the parts of the union unambiguously, even if some parts have the same type. For instance, the disjunctive type shown in Example 3.4.1.3 cannot be correctly represented by the mathematical union

$$\mathbb{R}^0 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2$$

because $\mathbb{R}^0 \cup \mathbb{R}^0 = \mathbb{R}^0$ and $\mathbb{R}^1 \cup \mathbb{R}^1 = \mathbb{R}^1$, so:

$$\mathbb{R}^0 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2 = \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2 \quad .$$

This representation has lost the distinction between e.g., `Linear(x)` and `OneRootQ(x)`.

In the Scala code, each part of a disjunctive type must be distinguished by a unique name such as `NoRoots`, `OneRoot`, and `TwoRoots`. To represent this mathematically, we can attach a distinct label to each part of the union. Labels are symbols without any special meaning, and we can just assume that labels are names of Scala case classes. Parts of the union are then represented by sets of pairs such as $(\texttt{OneRoot}, x)_{x \in \mathbb{R}^1}$. Then the domain `RootsOfQ` is expressed as:

$$\texttt{RootsOfQ} = (\texttt{NoRoots}, u)_{u \in \mathbb{R}^0} \cup (\texttt{OneRoot}, x)_{x \in \mathbb{R}^1} \cup (\texttt{TwoRoots}, (x, y))_{(x,y) \in \mathbb{R}^2} \quad .$$

This is an ordinary union of mathematical sets, but each of the sets has a unique label, so no two values from different parts of the union could possibly be equal. This kind of set is called a **labeled union** (it is also called tagged union or disjoint union). Each element of a labeled union is a pair of the form `(label, data)`, where the label uniquely identifies the part of the union, and the data can have any chosen type such as $\mathbb{R}^1$. If we use labeled unions, we cannot confuse different parts of the union even if their data have the same type, because labels are required to be distinct.

Labeled unions are not often used in mathematics, but they are needed in software engineering because real-life data is often described by sets comprised of several disjoint parts.

Named `Unit` types At first sight, it may seem strange that the zero-dimensional space is represented by a set containing *one* point. Why should we not use an empty set (rather than a set with one point) to represent the case where the equation has no real roots? The reason is that we are required to represent not only the values of the roots but also the information *about* the existence of the roots. The case with no real roots needs to be represented by some *value* of type `RootsOfQ`. This value cannot be missing, which would happen if we used an empty set to represent the no-roots case. It is natural to use the named empty tuple `NoRoots()` to represent this case, just as we used a named 2-tuple `TwoRoots(x, y)` to represent the case of two roots.

Consider the value u used by the mathematical set $(\texttt{NoRoots}, u)_{u\in\mathbb{R}^0}$. Since $\mathbb{R}^0$ consists of a single point, there is only *one* possible value of u. Similarly, the `Unit` type in Scala has only one distinct value, written as `()`. A case class with no parts, such as `NoRoots`, has only one distinct value, written as `NoRoots()`. This Scala value is fully analogous to the mathematical notation $(\texttt{NoRoots}, u)_{u\in\mathbb{R}^0}$.

So, case classes with no parts are quite similar to `Unit` except for an added name, e.g., `NoRoots()` is the `Unit` value `()` with name `NoRoots`. For this reason, they can be viewed as "named unit" types.

3.5.2 Disjunctive types in other programming languages

Disjunctive types and pattern matching turns out to be one of the defining features of FP languages. Languages that were not designed for functional programming do not support these features, while ML, OCaml, Haskell, F#, Scala, Swift, Elm, and PureScript support disjunctive types and pattern matching as part of the language design.

It is remarkable that named tuple types (also called "structs" or "records") are provided in almost every programming language, while disjunctive types are almost never present except in languages designed for the FP paradigm. (Ada and Pascal are the only languages that support disjunctive types without other FP features.[4])

The `union` types in C and C++ are not disjunctive types because it is not possible to determine which part of the union is represented by a given value. A `union` declaration in C looks like this:

```
union { int x; double y; long z; } i_d_l;
```

Without a label, we (and the compiler) will not know whether a given value of type `i_d_l` represents an `int`, a `double`, or a `long`. This will lead to errors that are hard to detect.

Programming languages of the C family (C, C++, Objective C, Java) support **enumeration** (`enum`) types, which are a limited form of disjunctive types, and a `switch` operation, which is a limited form of pattern matching. An `enum` type declaration in Java looks like this:

```
enum Color { RED, GREEN, BLUE; }
```

In Scala, this is equivalent to a disjunctive type containing three *empty* tuples:

```
sealed trait Color
final case class RED()   extends Color
final case class GREEN() extends Color
final case class BLUE()  extends Color
```

If we "enrich" the `enum` types with extra data, so that the tuples could be non-empty, and extend the `switch` expression to be able to handle the extra data, we will obtain the full functionality of disjunctive types. A definition of `RootsOfQ` could then look like this:

```
enum RootsOfQ {                  // This is not valid in Java!
  NoRoots(), OneRoot(Double x), TwoRoots(Double x, Double y);
}
```

A future version of Scala 3 will have a syntax for disjunctive types[5] that resembles "enriched `enum`":

```
enum RootsOfQ {  case NoRoots;  case OneRoot(x: Double);  case TwoRoots(x: Double, y: Double)  }
```

[4] https://en.wikipedia.org/wiki/Comparison_of_programming_languages_(basic_instructions)#Other_types
[5] https://dotty.epfl.ch/docs/reference/enums/adts.html

For comparison, the syntax for a disjunctive type equivalent to `RootsOfQ` in OCaml and Haskell is:

```
(* OCaml *)
type RootsOfQ = NoRoots | OneRoot of float | TwoRoots of float*float
```

```
-- Haskell
data RootsOfQ = NoRoots | OneRoot Double | TwoRoots (Double, Double)
```

This is more concise than the Scala syntax. When reasoning about disjunctive types, it is inconvenient to write out long type definitions. Chapter 5 will introduce a mathematical notation designed for efficient reasoning about types. That notation is still more concise than Haskell or OCaml.

3.5.3 Disjunctions and conjunctions in formal logic

In logic, a **proposition** is a logical formula that could be true or false. A **disjunction** of propositions A, B, C is denoted by $A \vee B \vee C$ and is true if and only if *at least one* of A, B, C is true. A **conjunction** of A, B, C is denoted by $A \wedge B \wedge C$ and is true if and only if *all* of the propositions A, B, C are true.

There is a similarity between a disjunctive data type and a logical disjunction of propositions. A value of the disjunctive data type `RootsOfQ` can be constructed only if we have one of the values `NoRoots()`, `OneRoot(x)`, or `TwoRoots(x, y)`. Let us now rewrite the previous sentence as a logical formula. Denote by $\mathcal{CH}(A)$ the logical proposition "this $\mathcal{C}$ode $\mathcal{H}$as a value of type `A`", where "this code" refers to a particular function in a program. So, the proposition "the function *can* compute a value of type `RootsOfQ`" is denoted by $\mathcal{CH}(\texttt{RootsOfQ})$. We can then write the above sentence about `RootsOfQ` as a logical formula:

$$\mathcal{CH}(\texttt{RootsOfQ}) = \mathcal{CH}(\texttt{NoRoots}) \vee \mathcal{CH}(\texttt{OneRoot}) \vee \mathcal{CH}(\texttt{TwoRoots}) \quad . \tag{3.1}$$

There is also a similarity between logical *conjunctions* and tuple types. Consider the named tuple (i.e., a case class) `TwoRoots(x: Double, y: Double)`. When can we have a value of type `TwoRoots`? Only if we have two values of type `Double`. Rewriting this sentence as a logical formula, we get:

$$\mathcal{CH}(\texttt{TwoRoots}) = \mathcal{CH}(\texttt{Double}) \wedge \mathcal{CH}(\texttt{Double}) \quad .$$

Formal logic admits the simplification:

$$\mathcal{CH}(\texttt{Double}) \wedge \mathcal{CH}(\texttt{Double}) = \mathcal{CH}(\texttt{Double}) \quad .$$

However, no such simplification will be available in the general case, e.g.:

```
case class Data3(x: Int, y: String, z: Double)
```

For this type, we will have the formula:

$$\mathcal{CH}(\texttt{Data3}) = \mathcal{CH}(\texttt{Int}) \wedge \mathcal{CH}(\texttt{String}) \wedge \mathcal{CH}(\texttt{Double}) \quad . \tag{3.2}$$

We find that tuples are related to logical conjunctions in the same way as disjunctive types are related to logical disjunctions. This is the main reason for choosing the name "disjunctive types".[6]

The correspondence between disjunctions, conjunctions, and data types is explained in more detail in Chapter 5. For now, we note that the operations of conjunction and disjunction are not sufficient to produce all possible logical expressions. To obtain a complete logic, it is also necessary to have the logical implication $A \to B$ ("if A is true than B is true"). It turns out that the implication $A \to B$ is related to the function type `A => B` in the same way as the disjunction operation is related to disjunctive types and the conjunction to tuples. In Chapter 4, we will study function types in depth.

[6] Disjunctive types are also known as "variants", "sum types", "co-product types", and "tagged union types".

Part II

Intermediate level

4 The logic of types. II. Curried functions

4.1 Functions that return functions

4.1.1 Motivation and first examples

Consider the task of preparing a logger function that prints messages with a configurable prefix.

A simple logger function can be a value of type `String => Unit`, such as:

```
val logger: String => Unit = { message => println(s"INFO: $message") }

scala> logger("hello world")
INFO: hello world
```

This function prints any given message with the logging prefix `"INFO"`.

The standard library function `println(...)` always returns a `Unit` value after printing its arguments. As we already know, there is only a single value of type `Unit`, and that value is denoted by `()`. To see that `println` returns `Unit`, run this code:

```
scala> val x = println(123)
123
x: Unit = ()
```

The task is to make the logging prefix configurable. A simple solution is to implement a function `logWith` that takes a prefix as an argument and returns a new logger containing that prefix. Note that the function `logWith` returns a new *function*, i.e., a new value of type `String => Unit`:

```
def logWith(prefix: String): (String => Unit) = {
    message => println(s"$prefix: $message")
}
```

The body of `logWith` consists of a nameless function `message => println(...)`, which is a value of type `String => Unit`. This value will be returned when we evaluate `logWith("...")`.

We can now use `logWith` to create some logger functions:

```
scala> val info = logWith("INFO")
info: String => Unit = <function1>

scala> val warn = logWith("WARN")
warn: String => Unit = <function1>
```

The created loggers are then usable as ordinary functions:

```
scala> info("hello")
INFO: hello

scala> warn("goodbye")
WARN: goodbye
```

The values `info` and `warn` can be used by any code that needs a logging function.

It is important that the prefix is "baked into" functions created by `logWith`. A logger such as `warn` will always print messages with the prefix `"WARN"`, and the prefix cannot be changed any more. This is because the value `prefix` is treated as a local constant within the body of the nameless function computed and returned by `logWith`. For instance, the body of the function `warn` is equivalent to:

```
{ val prefix = "WARN"; (message => s"$prefix: $message") }
```

Whenever a new function is created using `logWith(prefix)`, the (immutable) value of `prefix` is stored within the body of the newly created function. This is a general feature of nameless functions: the function's body keeps copies of all the outer-scope values it uses. One sometimes says that the function's body "closes over" those values; for this reason, nameless functions are also called "**closures**". It would be clearer to say that nameless functions "capture" values from outer scopes.

As another example of the capture of values, consider this code:

```
val f: (Int => Int) = {
  val p = 10
  val q = 20
  x => p + q * x
}
```

The body of the function `f` is equivalent to `{ x => 10 + 20 * x }` because the values `p = 10` and `q = 20` are captured.

The parentheses around a function's type, such as `(Int => Int)`, are optional in Scala.

4.1.2 Curried and uncurried functions

Reasoning mathematically about the following code:

```
val info = logWith("INFO")
info("hello")
```

we would expect that `info` is *the same value* as `logWith("INFO")`, and so the code `info("hello")` should have the same effect as the code `logWith("INFO")("hello")`. This is indeed so:

```
scala> logWith("INFO")("hello")
INFO: hello
```

The syntax `logWith("INFO")("hello")` looks like the function `logWith` applied to *two* arguments. Yet, `logWith` was defined as a function with a single argument of type `String`. This is not a contradiction because `logWith("INFO")` returns a function that accepts an additional argument. So, expressions `logWith("INFO")` and `logWith("INFO")("hello")` are both valid. In this sense, we are allowed to apply `logWith` to one argument at a time.

A function that can be applied to arguments in this way is called a **curried** function.

While a curried function can be applied to one argument at a time, an **uncurried** function must be applied to all arguments at once, e.g.:

```
def prefixLog(prefix: String, message: String): Unit = println(s"$prefix: $message")
```

The type of the curried function `logWith` is `String => (String => Unit)`. By Scala's syntax conventions, the function arrow (`=>`) groups to the *right*. So, the parentheses in the type expression `String => (String => Unit)` are not needed. The function's type can be written as `String => String => Unit`.

The type `String => String => Unit` is different from `(String => String) => Unit`, which is the type of a function returning `Unit` and having a function of type `String => String` as its argument. When an argument's type is a function type, e.g., `String => String`, it *must* be enclosed in parentheses.

In general, a curried function takes an argument and returns another function that again takes an argument and returns another function, and so on, until finally a non-function type is returned. So, the type signature of a curried function generally looks like `A => B => C => ... => R => S`, where `A`, `B`, ..., `R` are the **curried arguments** and `S` is the "final" result type.

For example, in the type expression `A => B => C => D` the types `A`, `B`, `C` are the types of curried arguments, and `D` is the final result type. It takes time to get used to reading this kind of syntax.

In Scala, functions defined with multiple argument groups (enclosed in multiple pairs of parentheses) are curried functions. We have seen examples of curried functions before:

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B]
def fmap[A, B](f: A => B)(xs: Option[A]): Option[B]
def foldLeft[A, R](xs: Seq[A])(init: R)(update: (R, A) => R): R
```

The type signatures of these functions can be also written equivalently as:

```
def map[A, B]: Seq[A] => (A => B) => Seq[B]
def fmap[A, B]: (A => B) => Option[A] => Option[B]
def foldLeft[A, R]: Seq[A] => R => ((R, A) => R) => R
```

Curried arguments of a *function type*, such as `(A => B)`, need parentheses.

To summarize, a curried function such as `logWith` can be defined in three equivalent ways in Scala:

```
1 def logWith1(prefix: String)(message: String): Unit = println(s"$prefix: $message")
2 def logWith2(prefix: String): String => Unit = { message => println(s"$prefix: $message") }
3 def logWith3: String => String => Unit = { prefix => message => println(s"$prefix: $message") }
```

For clarity, we will sometimes enclose nameless functions in parentheses or curly braces.

Line 3 above shows that the arrow symbols `=>` group to the right within the *code* of nameless functions. So, `x => y => expr` means `{x => {y => expr}}`, a nameless function taking an argument `x` and returning a nameless function that takes an argument `y` and returns an expression `expr`. This syntax convention is helpful since the code `x => y => z` visually corresponds to the curried function's type signature `A => B => C`, which uses the same syntax convention. Also, the syntax `(x => y) => z` could not possibly work for a nameless function because matching a function with the pattern `x => y` makes no sense. If we matched a function such as `{ t => t + 20 }` against the pattern `x => y` by setting `x = t` and `y = t + 20`, we would have no value for the bound variable `t`. (What would be the integer value of `y`?) So, `x => (y => z)` is the only sensible way of inserting parentheses into `x => y => z`.

Although the code `(x => y) => z` is invalid, the type expression `(A => B) => C` is valid. We may write a nameless function of type `(A=>B)=>C` as `f => expr` where `f: A => B` is the argument and `expr` the body.

4.1.3 Equivalence of curried and uncurried functions

We defined the curried function `logWith` in order to be able to create logger functions such as `info` and `warn`. However, some curried functions, such as `foldLeft`, are almost always applied to all possible arguments. A curried function applied to all its possible arguments is equivalent to an uncurried function that takes all those arguments at once. Let us look at this equivalence in more detail.

Consider a curried function with type signature `Int => Int => Int`. This function takes an integer and returns an (uncurried) function taking an integer and returning an integer. An example of such a curried function is:

```
def f1(x: Int): Int => Int = { y => x - y }
```

The function takes an integer `x` and returns the expression `y => x - y`, which is a function of type `Int => Int`. The code of `f1` can be written equivalently as:

```
val f1: Int => Int => Int = { x => y => x - y }
```

Let us compare the function `f1` with a function that takes its two arguments at once, e.g.:

```
def f2(x: Int, y: Int): Int = x - y
```

The function `f2` has type signature `(Int, Int) => Int`. The syntax for using `f1` and `f2` is different:

```
scala> f1(20)(4)
res0: Int = 16

scala> f2(20, 4)
res1: Int = 16
```

The main difference is that `f2` must be applied at once to both arguments, while `f1` could be applied to just the first argument (`20`). Applying a curried function to some but not all possible arguments is called a **partial application**. The result of evaluating `f1(20)` is a function that can be later applied to another argument:

```
scala> val r1 = f1(20)
r1: Int => Int = <function1>
```

```
scala> r1(4)
res2: Int = 16
```

Applying a curried function to all possible arguments is called a **saturated** application. A saturated application returns a value that is not a function and cannot be applied to more arguments.

If we need to partially apply an *uncurried* function, we can use the underscore (_) symbol:

```
1 scala> val r2: Int => Int = f2(20, _)
2 r2: Int => Int = <function1>
3
4 scala> r2(4)
5 res3: Int = 16
```

(The type annotation `Int => Int` is required in line 1.) This code creates a function `r2` by partially applying `f2` to the first argument but not to the second. Other than that, `r2` is the same function as `r1` defined above; i.e., `r2` returns the same values for the same arguments as `r1`. A more straightforward syntax for a partial application is:

```
scala> val r3: Int => Int = { x => f2(20, x) }    // Same as r2 above.
r3: Int => Int = <function1>

scala> r3(4)
res4: Int = 16
```

We can see that a curried function, such as `f1`, is better adapted for partial application than `f2`, because the syntax is shorter. However, the types of functions `f1` and `f2` are **equivalent**: for any `f1` we can reconstruct `f2` and vice versa, without loss of information:

```
def f2new(x: Int, y: Int): Int = f1(x)(y)             // f2new is equal to f2
def f1new: Int => Int => Int = { x => y => f2(x, y) } // f1new is equal to f1
```

It is clear that the function `f1new` computes the same results as `f1`, and that the function `f2new` computes the same results as `f2`. The equivalence of the functions `f1` and `f2` is not *equality* — these functions are *different*; but each of them can be reconstructed from the other.

More generally, a curried function has a type signature of the form `A => B => C => ... => R => S`, where `A`, `B`, `C`, ..., `S` are some types. A function with this type signature is equivalent to an uncurried function with type signature `(A,B,C,...,R) => S`. The uncurried function takes all arguments at once, while the curried function takes one argument at a time. Other than that, these two functions compute the same results given the same arguments.

We have seen how a curried function can be converted to an equivalent uncurried one, and vice versa. The Scala library defines the methods `curried` and `uncurried` that convert between these forms of functions. To convert between `f2` and `f1`:

```
scala> val f1c = (f2 _).curried
f1c: Int => (Int => Int) = <function1>

scala> val f2u = Function.uncurried(f1c)
f2u: (Int, Int) => Int = <function2>
```

The syntax `(f2 _)` is needed in Scala to convert methods to function values. Recall that Scala has two ways of defining a function: one as a method (defined using `def`), another as a function value (defined using `val`). The extra underscore will become unnecessary in Scala 3.

The methods `curried` and `uncurried` are quick to implement, as we will show in Section 4.2.1. These functions are called the **currying** and **uncurrying** transformations.

4.2 Fully parametric functions

We have seen that some functions are declared with type parameters, which are set only when the function is applied to specific arguments. Examples of such functions are `map` and `filter`, written as:

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B]
def filter[A](xs: Seq[A])(p: A => Boolean): Seq[A]
```

Such functions can be applied to arguments of different types without changing the function's code. It is clearly better to implement a single function with type parameters instead of writing several functions with repeated code but working with different types.

When we apply the function `map` as `map(xs)(f)` to a specific value `xs` of type, e.g., `Seq[Int]`, and a specific function `f` of type, say, `Int => String`, the Scala compiler will automatically set the type parameters `A = Int` and `B = String` in the code of `map`. We may also set type parameters explicitly and write, for example, `map[Int, String](xs)(f)`. This syntax shows a certain similarity between type parameters such as `Int`, `String` and "value parameters" (arguments) `xs` and `f`. Setting type parameters, e.g., `map[Int, String]`, means substituting `A = Int`, `B = String` into the type signature of the function, similarly to how setting value parameters means substituting specific values into the function body.

In the functions `map` and `filter` as just shown, some types are parameters while others are specific types, such as `Seq` and `Boolean`. It is sometimes possible to replace *all* specific types in the type signature of a function by type parameters. The result is a "fully parametric" function.

We call a function **fully parametric** if its arguments have types described by type parameters, and the code of the function treats all types as type parameters. In other words, fully parametric functions do not use any values of specific types, such as `Int` or `String`, in their code. A fully parametric function does not use any information about its argument types, other than assuming that those types correctly match the type signature.

What kind of functions are fully parametric? To build up intuition, let us compare the following two functions that have the same type signature:

```
def cos_sin(p: (Double, Double)): (Double, Double) = p match {
  case (x, y) =>
    val r = math.sqrt(x * x + y * y)
    (x / r, y / r)   // Return cos and sin of the angle, or ‘NaN‘ when undefined.
}

def swap(p: (Double, Double)): (Double, Double) = p match {
  case (x, y) => (y, x)
}
```

We can introduce type parameters into the type signature of `swap` to make it fully parametric:

```
def swap[A, B](p: (A, B)): (B, A) = p match {
  case (x, y) => (y, x)
}
```

Converting `swap` into a fully parametric function is possible because the operation of swapping the parts of a tuple `(A, B)` works in the same way for all types `A`, `B`. No changes were made in the body of the function. The specialized version of `swap` working on `(Double, Double)` can be obtained from the fully parametric version of `swap` if we set the type parameters as `A = Double`, `B = Double`.

In contrast, the function `cos_sin` performs a computation that is specific to the type `Double`. That computation cannot be generalized to an arbitrary type parameter `A` instead of the type `Double`. So, `cos_sin` cannot be generalized to a fully parametric function.

Typically, a fully parametric function has all its arguments typed with type parameters or with some combinations of type parameters, i.e., **type expressions** such as `(A, B)` or `X => Either[X, Y]`.

The `swap` operation for pairs is already defined in the Scala library:

```
scala> (1, "abc").swap
res0: (String, Int) = (abc,1)
```

If needed, other swapping functions can be implemented for tuples with more elements, e.g.:

```
def swapAC[A,B,C]: ((A, B, C)) => (C, B, A) = { case (x, y, z) => (z, y, x) }
```

The Scala syntax requires *double* parentheses around tuple types of arguments but not around the tuple type of a function's result. So, the function `cos_sin` may be written as a value like this:

```
val cos_sin: ((Double, Double)) => (Double, Double) = ...
```

4.2.1 Examples. Function composition

Further examples of fully parametric functions are the identity function, the constant function, the function composition methods, and the currying / uncurrying transformations.

The identity function is available in the Scala library as `identity[T]`:

```
def identity[T]: T => T = (t => t)
```

In the mathematical notation, we write the identity function as "id" for brevity.

The constant function (available in the Scala library as `Function.const[C, X]`) takes an argument `c` of type `C` and returns *a new function* that always returns `c`:

```
def const[C, X](c: C): X => C = (_ => c)
```

The syntax `_ => c` is used to emphasize that the new returned function ignores its argument.

Function composition Consider two functions `f: Int => Double` and `g: Double => String`. We can apply `f` to an integer argument `x` and get a result `f(x)` of type `Double`. Applying `g` to that result gives a `String` value `g(f(x))`. The transformation from an `x` of type `Int` to a final `String` value `g(f(x))` can be viewed as a new function of type `Int => String`. That new function is called the **forward composition** of the two functions `f` and `g`. In Scala, the forward composition of `f` and `g` is written as `f andThen g`:

```
val f: Int => Double = (x => 5.67 + x)
val g: Double => String = (x => f"Result x = $x%3.2f")

scala> val h = f andThen g       // h(x) is defined as g(f(x)).
h: Int => String = <function1>

scala> h(40)
res36: String = Result x = 45.67
```

The Scala compiler derives the type of `h` automatically as `Int => String`.

This book denotes the forward composition by the symbol $\fatsemi$ (pronounced "before"). We write:

$$f \fatsemi g \stackrel{\text{def}}{=} x \to g(f(x)) \quad . \tag{4.1}$$

The symbol $\stackrel{\text{def}}{=}$ means "is defined as" or "is equal by definition".

We could implement the operation of forward composition as a fully parametric function:

```
def andThen[X, Y, Z](f: X => Y)(g: Y => Z): X => Z = { x => g(f(x)) }
```

This type signature requires the types of the function arguments to match in a certain way, or else the composition is undefined (and the code would produce a type error).

The method `andThen` is an example of a function that *both* returns a new function *and* takes other functions as arguments.

The **backward composition** of two functions f and g works in the opposite order: first g is applied and then f. Using the symbol $\circ$ (pronounced "after") for this operation, we can write:

$$f \circ g \stackrel{\text{def}}{=} x \to f(g(x)) \quad . \tag{4.2}$$

In Scala, the backward composition is called `compose` and used as `f compose g`. This method may be implemented as a fully parametric function:

```
def compose[X, Y, Z](f: Y => X)(g: Z => Y): Z => X = { z => f(g(z)) }
```

We have already seen the methods `curried` and `uncurried` defined by the Scala library. As an illustration, here is the code for the **uncurrying** transformation (converting curried functions to uncurried):

```
def uncurry[A, B, R](f: A => B => R): ((A, B)) => R = { case (a, b) => f(a)(b) }
```

These examples show that fully parametric functions perform operations so general that they work in the same way for all types. Some arguments of fully parametric functions may have complicated types such as `A => B => R`, which are type expressions built up from type parameters. But fully parametric functions do not use values of specific types such as `Int` or `String`.

Functions with type parameters are sometimes called "generic". This book uses the term "**fully parametric**" to designate a certain restricted kind of generic functions.

4.2.2 Laws of function composition

The operations of function composition, introduced in Section 4.2.1, have three important properties or "laws" that follow directly from the definitions. These laws are:

- The two **identity laws**: the composition of any function f with the identity function (`identity[A]`) will give again the function f.
- The **associativity law**: the consecutive composition of three functions f, g, h does not depend on the order in which the pairs are composed.

These laws hold equally for the forward and the backward composition, since those are just syntactic variants of the same operation. Let us write these laws rigorously as equations and prove them.

Proofs with forward composition The composition of the identity function with an arbitrary function f on the left is written as $f \fatsemi \text{id}$. The composition with the function f on the right is written as $\text{id} \fatsemi f$. In both cases, the result must be equal to the function f. The resulting two laws are:

$$\text{left identity law of composition}: \quad \text{id} \fatsemi f = f \quad ,$$
$$\text{right identity law of composition}: \quad f \fatsemi \text{id} = f \quad .$$

To prove that these laws hold, we need to show that both sides of the laws, which are functions, give the same result when applied to an arbitrary value x. Let us first clarify how the type parameters must be set for all types to match consistently.

The laws must hold for an arbitrary function f. Assume that f has the type signature $A \to B$, where A and B are arbitrary types (type parameters). Consider the left identity law. The function $(\text{id} \fatsemi f)$ is, by definition (4.1), a function that takes an argument x, applies id to that x, and then applies f to the result:

$$\text{id} \fatsemi f = (x \to f\,(\text{id}\,(x))) \quad .$$

If f has type $A \to B$, its argument must be of type A, or else the types will not match. Therefore, the identity function must have type $A \to A$, and the argument x must have type A. With these choices of the type parameters, the function $(x \to f(\text{id}(x)))$ will have type $A \to B$. This type matches the right-hand side of the law, which is just f. We add type annotations to the code as *superscripts*:

$$\text{id}^{:A\to A} \fatsemi f^{:A\to B} = \left(x^{:A} \to f\,(\text{id}\,(x))\right)^{:A\to B} \quad .$$

In the Scala syntax, this formula may be written as:

```
identity[A] andThen (f: A => B) == { x: A => f(identity(x)) }: A => B
```

We will follow the convention where type parameters are single uppercase letters, as is common in Scala code (although this convention is not enforced by the Scala compiler). The colon symbol (:) in the superscript $x^{:A}$ means a type annotation, as in Scala code `x:A`. Superscripts *without* a colon, such as id^A, denote type parameters, as in Scala code `identity[A]`. Since the function `identity[A]` has type `A => A`, we can write id^A or equivalently (but more verbosely) $\text{id}^{:A\to A}$ to denote that function.

Now we can prove the law. By definition of the identity function, we have $\text{id}\,(x) = x$, and so:

$$\text{id} \fatsemi f = (x \to f(\text{id}\,(x))) = (x \to f(x)) = f \quad .$$

The last step works since $x \to f(x)$ is a function taking an argument x and applying f to that argument; i.e., $x \to f(x)$ is an **expanded form** of the same function f.

We turn to the right identity law, $f \fatsemi \text{id} = f$. Write out the left-hand side:

$$f \fatsemi \text{id} = (x \to \text{id}\,(f(x))) \quad .$$

To check that the types match, assume that $f^{:A\to B}$. Then x must have type A, and the identity function must have type $B \to B$. The result of $\text{id}\,(f(x))$ will also have type B. With these choices of type parameters, all types match:

$$f^{:A\to B} \fatsemi \text{id}^{:B\to B} = \left(x^{:A} \to \text{id}\,(f(x))\right)^{:A\to B} \quad .$$

Since $\text{id}\,(f(x)) = f(x)$, we find:

$$f \fatsemi \text{id} = (x \to f(x)) = f \quad .$$

In this way, we have demonstrated that both identity laws hold.

The associativity law is written as an equation like this:

$$\text{associativity law of function composition}: \quad (f \fatsemi g) \fatsemi h = f \fatsemi (g \fatsemi h) \quad . \tag{4.3}$$

Let us verify that the types match here. The types of the functions f, g, and h must be such that all the function compositions match. If f has type $A \to B$ for some type parameters A and B, then the argument of g must be of type B; so we must have $g^{:B\to C}$, where C is another type parameter. The composition $f \fatsemi g$ has type $A \to C$, so h must have type $C \to D$ for some type D. Assuming the types as $f^{:A\to B}$, $g^{:B\to C}$, and $h^{:C\to D}$, we find that the types in all the compositions $f \fatsemi g$, $g \fatsemi h$, $(f \fatsemi g) \fatsemi h$, and $f \fatsemi (g \fatsemi h)$ match. We can rewrite Eq. (4.3) with type annotations:

$$(f^{:A\to B} \fatsemi g^{:B\to C}) \fatsemi h^{:C\to D} = f^{:A\to B} \fatsemi (g^{:B\to C} \fatsemi h^{:C\to D}) \quad . \tag{4.4}$$

After checking the types, we are ready to verify the associativity law. Note that both sides of the law (4.4) are functions of type $A \to D$. To prove that two functions are equal means to prove that they return the same results when applied to the same arguments. So, let us apply both sides of Eq. (4.4) to an arbitrary value $x^{:A}$. Using the definition (4.1) of the forward composition, we find:

$$((f \fatsemi g) \fatsemi h)\,(x) = h\,((f \fatsemi g)\,(x)) = h(g(f(x))) \quad ,$$
$$(f \fatsemi (g \fatsemi h))\,(x) = (g \fatsemi h)\,(f(x)) = h(g(f(x))) \quad .$$

Both sides of the law are now equal when applied to an arbitrary value x. This concludes the proof.

Because of the associativity law, we do not need parentheses when writing the expression $f \fatsemi g \fatsemi h$. The function $(f \fatsemi g) \fatsemi h$ is equal to the function $f \fatsemi (g \fatsemi h)$.

In the proof, we have omitted the type annotations since we already checked that all types match. Checking the types beforehand allows us to write shorter derivations.

Proofs with backward composition This book prefers to use the **forward composition** $f \fatsemi g$ rather than the backward composition $g \circ f$. If desired, all equations can be converted from one notation to the other by reversing the order of compositions:

$$f \fatsemi g \stackrel{\text{def}}{=} g \circ f$$

for any functions $f^{:A\to B}$ and $g^{:B\to C}$. Let us see how to prove the composition laws in the backward notation. We will just need to reverse the order of function compositions in the proofs above.

The left identity and right identity laws are:

$$f \circ \text{id} = f \quad , \quad \text{id} \circ f = f \quad .$$

To match the types, we need to choose the type parameters as:

$$f^{:A\to B} \circ \text{id}^{:A\to A} = f^{:A\to B} \quad , \quad \text{id}^{B\to B} \circ f^{:A\to B} = f^{:A\to B} \quad .$$

We now apply both sides of the laws to an arbitrary value $x^{:A}$. For the left identity law, we find:

$$\text{use definition (4.2)}: \quad f \circ \text{id} = (x \to f(\text{id}\,(x))) = (x \to f(x)) = f \quad .$$

Similarly for the right identity law:

$$\text{id} \circ f = (x \to \text{id}\,(f(x))) = (x \to f\,(x)) = f \quad .$$

The associativity law,

$$h \circ (g \circ f) = (h \circ g) \circ f \quad ,$$

is proved by applying both sides to an arbitrary value x of a suitable type:

$$(h \circ (g \circ f))(x) = h((g \circ f)(x)) = h(g(f(x))) \quad ,$$
$$((h \circ g) \circ f)(x) = (h \circ g)(f(x)) = h(g(f(x))) \quad .$$

The types are checked by assuming that f has the type $f^{:A \to B}$. The types in $g \circ f$ match only when $g^{:B \to C}$, and then $g \circ f$ is of type $A \to C$. The type of h must be $h^{:C \to D}$ for the types in $h \circ (g \circ f)$ to match. We can write the associativity law with type annotations as:

$$h^{:C \to D} \circ (g^{:B \to C} \circ f^{:A \to B}) = (h^{:C \to D} \circ g^{:B \to C}) \circ f^{:A \to B} \quad . \tag{4.5}$$

The associativity law allows us to omit parentheses in the expression $h \circ g \circ f$.

The length of calculations is the same in the forward and the backward notation. One difference is that types of function compositions are more visually clear in the forward notation: it is harder to check that types match in Eq. (4.5) than in Eq. (4.4). To make the backward notation easier to work with, one could write[1] the function types in reverse as, e.g., $g^{:C \leftarrow B} \circ f^{:B \leftarrow A}$.

4.2.3 Example: A function that is *not* fully parametric

Fully parametric functions do not make any decisions based on the actual types of arguments. As an example of code that is *not* fully parametric, consider the following "fake identity" function:

```
def fid[A]: A => A = {
  case x: Int   => (x - 1).asInstanceOf[A]     // Special code for A = Int.
  case x        =>  x                          // Standard code for all other types A.
}
```

This function's type signature is the same as that of `identity[A]`, and its behavior is the same for all types `A` except for `A = Int`:

```
scala> fid("abc")
res0: String = abc

scala> fid(true)
res1: Boolean = true

scala> fid(0)
res2: Int = -1
```

While Scala allows us to write this kind of code, the result is confusing: the type signature `A => A` does not indicate a special behavior with `A = Int`. In any case, `fid` is not a fully parametric function.

Let us see whether the identity laws of function composition hold when using `fid[A]` instead of the correct function `identity[A]`. To see that, we compose `fid` with a simple function `f_1` defined by:

```
def f_1: Int => Int = { x => x + 1 }
```

The composition (`f_1 andThen fid`) has type `Int => Int`. Since `f_1` has type `Int => Int`, Scala will automatically set the type parameter `A = Int` in `fid[A]`:

```
scala> def f_2 = f_1 andThen fid    // f2 = f1 ⨾ fid
f_2: Int => Int
```

By the identity law, we should have $f_2 = f_1 ⨾ \text{id} = f_1$. But we can check that `f_1` and `f_2` are not equal:

```
scala> f_1(0)
res3: Int = 1

scala> f_2(0)
res4: Int = 0
```

It is important that we are able to detect that `fid` is not a fully parametric function by checking whether some equation holds, without looking at the code of `fid`. In this book, we will always formulate any desired properties through equations or "laws". To verify that a law holds, we will perform symbolic calculations similar to the proofs in Section 4.2.2. These calculations are **symbolic** in the sense that

[1] This is done in the book "Program design by calculation", see http://www4.di.uminho.pt/~jno/ps/pdbc.pdf

we are manipulating symbols (such as x, f, g, h) without substituting any specific values for these symbols but only using some general rules and properties. This is similar to symbolic calculations in mathematics, such as $(x-y)\left(x^2+xy+y^2\right)=x^3-y^3$. In the next section, we will get more experience with symbolic calculations relevant to functional programming.

4.3 Symbolic calculations with nameless functions

4.3.1 Calculations with curried functions

In mathematics, functions are evaluated by substituting their argument values into their body. Each sub-expression is then evaluated and its result substituted into the larger expression.

Nameless functions are evaluated in the same way. For example, applying the nameless function $x \to x+10$ to an integer 2, we substitute 2 instead of x in "$x+10$" and get the sub-expression "$2+10$". Then we evaluate that sub-expression to 12. The computation is written like this:

$$(x \to x+10)(2) = 2+10 = 12 \quad .$$

To run this computation in Scala, we need to add a type annotation to the nameless function as in $(x^{:\text{Int}} \to x+10)(2)$. The code is:

```
scala> ((x: Int) => x + 10)(2)
res0: Int = 12
```

Curried function calls such as $f(x)(y)$ or $\left(x \to \text{expr}(x)\right)(y)(z)$ may look unfamiliar and confusing. We need to get some experience working with them.

Consider a curried nameless function being applied to arguments, such as `(x => y => x - y)(20)(4)`, and compute the result of this function application. Begin with the argument `20`. Applying a nameless function of the form `(x => ...)` to `20` means substituting `x = 20` into the body of the function. After that substitution, we obtain the expression `y => 20 - y`, which is again a nameless function. Applying that function to the remaining argument `(4)` means substituting `y = 4` into the body of `y => 20 - y`. We get the expression `20 - 4`, which equals `16`. Check the result in Scala:

```
scala> ((x: Int) => (y: Int) => x - y)(20)(4)
res1: Int = 16
```

Applying a curried function such as `x => y => z => expr(x,y,z)` to three curried arguments `10`, `20`, and `30` means substituting `x = 10`, `y = 20`, and `z = 30` into the expression `expr(x,y,z)`. In this way, we can easily apply a curried function to any number of curried arguments.

This calculation is helped by the convention that `f(g)(h)` means first applying `f` to `g` and then applying the result to `h`. In other words, function application groups to the *left*: `f(g)(h) = (f(g))(h)`. It would be confusing if function application grouped to the right and `f(g)(h)` meant first applying `g` to `h` and then applying `f` to the result. If *that* were the syntax convention, it would be harder to reason about applying a curried function to its arguments.

We see that the right grouping of the function arrow `=>` is well adapted to the left grouping of function applications. All functional languages follow these syntactic conventions.

To make calculations shorter, we will write code in a mathematical notation rather than in the Scala syntax. Type annotations are written with a colon in the superscript, for example: $x^{:\text{Int}} \to x+10$ instead of the Scala expression `((x: Int) => x + 10)`.

The symbolic evaluation of the Scala code `((x: Int) => (y: Int) => x - y)(20)(4)` can be written as:

$$\begin{aligned}
& (x^{:\text{Int}} \to y^{:\text{Int}} \to \underline{x} - y)\,(20)\,(4) \\
\text{apply function and substitute } x=20: \quad & = (y^{:\text{Int}} \to 20 - \underline{y})\,(4) \\
\text{apply function and substitute } y=4: \quad & = 20-4 = 16 \quad .
\end{aligned}$$

In the above step-by-step calculation, the colored underlines and comments at left are added for clarity. A colored underline indicates a sub-expression that is going to be rewritten at the *next* step.

Here we performed calculations by substituting an argument into a function at each step. A compiled Scala program is evaluated in a similar way at run time.

Nameless functions are *values* and can be used as part of larger expressions, just as any other values. For instance, nameless functions can be arguments of other functions (nameless or not). Here is an example of applying a nameless function $f \to f(9)$ to a nameless function $x \to x\%4$:

$$\begin{aligned} & (f \to \underline{f}(9))\,(x \to x\%4) \\ \text{substitute } f = (x \to x\%4): \quad & = (x \to \underline{x}\%4)(9) \\ \text{substitute } x = 9: \quad & = 9\%4 = 1 \quad . \end{aligned}$$

In the nameless function $f \to f(9)$, the argument f has to be itself a function, otherwise the expression $f(9)$ would make no sense. The argument x of $f(x)$ must be an integer, or else we would not be able to compute $x\%4$. The result of computing $f(9)$ is 1, an integer. We conclude that f must have type Int → Int, or else the types do not match.

To verify this result in Scala, we need to specify a type annotation for f:

```
scala> ((f: Int => Int) => f(9))(x => x % 4)
res2: Int = 1
```

No type annotation is needed for $x \to x\%4$ since the Scala compiler already knows the type of f and figures out that x in $x \to x\%4$ must have type `Int`.

Let us summarize the syntax conventions for curried nameless functions:

- Function expressions group everything to the right: $x \to y \to z \to e$ means $x \to (y \to (z \to e))$.
- Function calls group everything to the left. So, $f(x)(y)(z)$ means $\big((f(x))(y)\big)(z)$, i.e., $f(x)$ gives a new function that is applied to y, giving again a new function that is finally applied to z.
- Function applications group stronger than infix operations, so $f(x) + y$ means $(f(x)) + y$, as usual in mathematics, and not $f(x + y)$.

Here are some more examples of performing function applications symbolically. Types are omitted for brevity; every non-function value is of type `Int`:

$$\begin{aligned} (x \to x * 2)\,(10) &= 10 * 2 = 20 \quad . \\ (p \to z \to z * p)\,(t) &= (z \to z * t) \quad . \\ (p \to z \to z * p)\,(t)(4) &= (z \to z * t)(4) = 4 * t \quad . \end{aligned}$$

Some results of these computation are integer values such as 20; other results are nameless functions such as $z \to z * t$. Verify this in Scala:

```
scala> ((x:Int) => x*2)(10)
res3: Int = 20

scala> ((p:Int) => (z:Int) => z*p)(10)
res4: Int => Int = <function1>

scala> ((p:Int) => (z:Int) => z*p)(10)(4)
res5: Int = 40
```

In the following examples, some arguments are themselves functions. Consider an expression that uses the nameless function $(g \to g(2))$ as an argument:

$$(f \to p \to \underline{f}(p))\,(g \to g(2)) \tag{4.6}$$

$$\begin{aligned} \text{substitute } f = (g \to g(2)): \quad & = p \to (g \to \underline{g}(2))\,(p) \\ \text{substitute } g = p: \quad & = p \to p(2) \quad . \end{aligned} \tag{4.7}$$

The result of this expression is a function $p \to p(2)$ that will apply *its* argument p to the value 2. A possible value for p is the function $x \to x+4$. So, let us apply the expression in Eq. (4.6) to $p \to p(2)$:

$$\begin{aligned}
& \underline{(f \to p \to f(p))\,(g \to g(2))}\,(x \to x+4) \\
\text{use Eq. (4.7)}: \quad & = (p \to \underline{p}(2))\,(x \to x+4) \\
\text{substitute } p = (x \to x+4): \quad & = (x \to \underline{x}+4)\,(2) \\
\text{substitute } x = 2: \quad & = 2+4 = 6 \quad .
\end{aligned}$$

To verify this calculation in Scala, we need to add appropriate type annotations for f and p. To figure out the types, we reason like this:

We know that the function $f \to p \to f(p)$ is being applied to the arguments $f = (g \to g(2))$ and $p = (x \to x+4)$. So, the argument f in $f \to p \to f(p)$ must be a function that takes p as an argument.

The variable x in $x \to x+4$ must be of type `Int`. Thus, the type of the expression $x \to x+4$ is Int → Int, and so must be the type of the argument p. We write $p^{:\text{Int}\to\text{Int}}$.

Finally, we need to make sure that the types match in the function $f \to p \to f(p)$. Types match in $f(p)$ if the type of f's argument is the same as the type of p, which is Int → Int. So f's type must be (Int → Int) → A for some type A. Since in our example $f = (g \to g(2))$, types match only if g has type Int → Int. But then $g(2)$ has type Int, and so we must have A = Int. Thus, the type of f is (Int → Int) → Int. We know enough to write the Scala code now:

```
scala> ((f: (Int => Int) => Int) => p => f(p))(g => g(2))(x => x + 4)
res6: Int = 6
```

Type annotations for p, g, and x may be omitted because the Scala compiler will figure out the missing types from the given type of f. However, it is never an error to specify more type annotations when writing code; it makes the code longer but may be helpful.

4.3.2 Solved examples: Deriving a function's type from its code

Checking that the types match is an important part of the functional programming paradigm, both in the practice of writing code and in theoretical derivations of laws for various functions. For instance, in the derivations of the composition laws (Section 4.2.2), we were able to deduce the possible type parameters for f, g, and h in the expression $f \fatsemi g \fatsemi h$. This worked because the composition operation `andThen` (denoted by the symbol $\fatsemi$) is fully parametric. Given a fully parametric function, one can derive the most general type signature that matches the body of that function. The same type-deriving procedure may also help in converting a given function to a fully parametric form.

Let us look at some examples of doing this.

Example 4.3.2.1 The functions `const` and `id` were defined in Section 4.2.1. What is the value `const(id)` and what is its type? Determine the most general type parameters in the expression `const(id)`.

Solution We need to treat the functions `const` and `id` as values, since our goal is to apply `const` to `id`. Write the code of these functions in a short notation:

$$\begin{aligned}
\text{const}^{C,X} &\stackrel{\text{def}}{=} c^{:C} \to _^{:X} \to c \quad , \\
\text{id}^{A} &\stackrel{\text{def}}{=} a^{:A} \to a \quad .
\end{aligned}$$

The types will match in the expression `const(id)` only if the argument of the function `const` has the same type as the type of `id`. Since `const` is a curried function, we need to look at its *first* curried argument, which is of type C. The type of `id` is $A \to A$, where A is so far an arbitrary type. So, the type parameter C in $\text{const}^{C,X}$ must be equal to $A \to A$:

$$C = A \to A \quad .$$

The type parameter X in $\text{const}^{C,X}$ is not constrained, so we keep it as X. The result of applying `const` to `id` is of type $X \to C$, which equals $X \to A \to A$. In this way, we find:

$$\text{const}^{A\to A,X}(\text{id}^{A}) : X \to A \to A \quad .$$

The types A and X can be arbitrary. The type $X \to A \to A$ is the most general type for the expression `const(id)` because we have not made any assumptions about the types except requiring that all functions must be always applied to arguments of the correct types.

To compute the value of `const(id)`, it remains to substitute the code of `const` and `id`. Since we already checked the types, we may omit all type annotations:

$$\begin{aligned}
& \underline{\text{const}}\,(\text{id}) \\
\text{definition of const}: \quad & = (c \to x \to \underline{c})(\text{id}) \\
\text{apply function, substitute } c = \text{id}: \quad & = x \to \underline{\text{id}} \\
\text{definition of id}: \quad & = x \to a \to a \quad .
\end{aligned}$$

The function $(x \to a \to a)$ takes an argument $x^{:X}$ and returns the identity function $a^{:A} \to a$. It is clear that the argument x is ignored by this function, so we can rewrite it equivalently as:

$$\text{const}\,(\text{id}) = _^{:X} \to a^{:A} \to a \quad .$$

Example 4.3.2.2 Implement a function `twice` that takes a function `f: Int => Int` as its argument and returns a function that applies `f` twice. For instance, if the function `f` is `{ x => x + 3 }`, the result of `twice(f)` should be equal to the function `x => x + 6`. Test this with the expression `twice(x => x + 3)(10)`. After implementing the function `twice`, generalize it to a fully parametric function.

Solution According to the requirements, the function `twice` must return a new function of type `Int => Int`. So the type signature of `twice` is

```
def twice(f: Int => Int): Int => Int = ???
```

Since `twice(f)` must be a new function with an integer argument, we begin the code of `twice` by writing a new nameless function `{ (x: Int) => ... }`,

```
def twice(f: Int => Int): Int => Int = { (x: Int) => ??? }
```

The new function must apply `f` twice to its argument, that is, it must return `f(f(x))`. We can finish the implementation now:

```
def twice(f: Int => Int): Int => Int = { x => f(f(x)) }
```

The type annotation `(x: Int)` can be omitted. Let us verify that `twice(x => x+3)(10)` equals `10 + 6`:

```
scala> val g = twice(x => x + 3)  // Expect g to be equal to the function { x => x + 6 }.
g: Int => Int = <function1>

scala> g(10)                      // Expect twice(x => x + 3)(10) to be equal to (x => x+6)(10) = 16.
res0: Int = 16
```

To transform `twice` into a fully parametric function means replacing its type signature by a fully parameterized type signature while keeping the function body unchanged:

```
def twice[A, B, ...](f: ...): ... = { x => f(f(x)) }
```

To determine the type signature and the possible type parameters A, B, ..., we need to determine the most general type that matches the function body. The function body is the expression $x \to f(f(x))$. Assume that x has type A; for types to match in the sub-expression $f(x)$, we need f to have type $A \to B$ for some type B. The sub-expression $f(x)$ will then have type B. For types to match in $f(f(x))$, the argument of f must have type B; but we already assumed $f^{:A\to B}$. This is consistent only if $A = B$. In this way, $x^{:A}$ implies $f^{:A\to A}$, and the expression $x \to f(f(x))$ has type $A \to A$. We can now write the type signature of `twice`:

```
def twice[A](f: A => A): A => A = { x => f(f(x)) }
```

This fully parametric function has only one independent type parameter, A, and can be equivalently written in the code notation as:

$$\text{twice}^A \stackrel{\text{def}}{=} f^{:A\to A} \to x^{:A} \to f(f(x)) = f^{:A\to A} \to f \,\fatsemi\, f \quad . \tag{4.8}$$

The procedure of deriving the most general type for a given code is called **type inference**. In Example 4.3.2.2, the presence of the type parameter A and the type signature $(A \to A) \to A \to A$ have been "inferred" from the code $f \to x \to f(f(x))$.

Example 4.3.2.3 Consider the fully parametric function `twice` defined in Example 4.3.2.2. What is the most general type of `twice(twice)`, and what computation does it perform? Test your answer on the expression `twice(twice)(x => x+3)(10)`. What are the type parameters in that expression?

Solution Note that `twice(twice)` means that the function `twice` is used as *its own* argument, i.e., this is `twice(f)` with `f = twice`. We begin by assuming unknown type parameters as `twice[A](twice[B])`. The function `twice[A]` of type $(A \to A) \to A \to A$ can be applied to the argument `twice[B]` only if `twice[B]` has type $A \to A$. But `twice[B]` is of type $(B \to B) \to B \to B$. The symbol $\to$ groups to the right, so we have:

$$(B \to B) \to B \to B = (B \to B) \to (B \to B) \quad .$$

This can match with $A \to A$ only if we set $A = (B \to B)$. So the most general type of `twice(twice)` is:

$$\text{twice}^{B \to B}(\text{twice}^{B}) : (B \to B) \to B \to B \quad . \tag{4.9}$$

After checking that types match, we may omit types from further calculations.

Example 4.3.2.2 defined `twice` with the `def` syntax. To use `twice` as an argument in the expression `twice(twice)`, it is convenient to define `twice` as a value, `val twice = ...` However, the function `twice` needs type parameters, and Scala 2 does not directly support `val` definitions with type parameters. Scala 3 will support type parameters appearing together with arguments in a nameless function:

```
val twice = [A] => (f: A => A) => (x: A) => f(f(x)) // Valid only in Scala 3.
```

Keeping this in mind, we use the definition of `twice` from Eq. (4.8): $\text{twice}(f) = f \fatsemi f$, which omits the curried argument $x^{:A}$ and makes the calculation shorter. Substituting that into `twice(twice)`, we find:

$$\begin{aligned}
& \text{twice}(\text{twice}) = \text{twice} \fatsemi \text{twice} \\
\text{expand function composition}: \quad & = f \to \text{twice}(\underline{\text{twice}}(f)) \quad . \\
\text{definition of twice}(f): \quad & = f \to \underline{\text{twice}}(f \fatsemi f) \\
\text{definition of twice}: \quad & = f \to f \fatsemi f \fatsemi f \fatsemi f \quad .
\end{aligned}$$

This clearly shows that `twice(twice)` is a function applying its (function-typed) argument *four* times.

The types in `twice(twice)(x => x+3)` follow from Eq. (4.9): since `x => x+3` has type `Int => Int`, types will match only if we set $B = \text{Int}$. The result is `twice[Int => Int](twice[Int])`. To test, we need to write at least one type parameter in the code, or else Scala cannot correctly infer the types in `twice(twice)`:

```
scala> twice(twice[Int])(x => x + 3)(10) // Or write ‘twice[Int => Int](twice)(x => x + 3)(10)‘ .
res0: Int = 22
```

This confirms that `twice(twice)(x => x+3)` equals the function `x => x + 12`, unlike `twice(twice(x => x+3))`, which equals the function `x => x + 6` as shown in Example 4.3.2.2.

Example 4.3.2.4 **(a)** For the given function `p`, infer a general type signature with type parameter(s):

```
def p[...]:... = { f => f(2) }
```

(b) Could we choose the type parameters in the expression `p(p)` such that the types match?

Solution **(a)** In the nameless function $f \to f(2)$, the argument f must be itself a function with an argument of type `Int`, otherwise the sub-expression $f(2)$ is ill-typed. So, types will match if f has type $\text{Int} \to \text{Int}$ or $\text{Int} \to \text{String}$ or similar. The most general case is when f has type $\text{Int} \to A$, where A is an arbitrary type (i.e., a type parameter); then the value $f(2)$ has type A. Since the nameless function $f \to f(2)$ has an argument f of type $\text{Int} \to A$ and a result $f(2)$ of type A, we find that the type of p must be $(\text{Int} \to A) \to A$. With this type assignment, all types match. The type parameter A remains undetermined and is added to the type signature of the function `p`. The code is:

```
def p[A]: (Int => A) => A = { f => f(2) }
```

Notation	Scala syntax	Comments
$x^{:A}$	`x: A`	a value or an argument of type `A`
$f^{:A \to B}$	`f: A => B`	a function of type `A => B`
$x^{:\text{Int}} \to f(x)$	`{ x: Int => f(x) }`	a nameless function with argument `x`
$f^{A,B} \stackrel{\text{def}}{=} \ldots$	`def f[A, B] = ...`	define a function with type parameters
id^A, also $\text{id}^{:A \to A}$	`identity[A]`	the standard "identity" function
$A \to B \to C$	`A => B => C`	type of a curried function
$f \fatsemi g$	`f andThen g`	forward composition of functions
$g \circ f$	`g compose f`	backward composition of functions

Table 4.1: Some notation for symbolic reasoning about code.

(b) The expression `p(p)` applies `p` to itself, just as `twice(twice)` did in Example 4.3.2.3. Begin by writing `p(p)` with unknown type parameters: `p[A](p[B])`. Then try to choose `A` and `B` so that the types match in that expression. Does the type of `p[B]`, which is `(Int => B) => B`, match the type of the argument of `p[A]`, which is `Int => A`, with some choice of `A` and `B`? A function type `P => Q` matches `X => Y` only if `P = X` and `Q = Y`. So `(Int => B) => B` can match `Int => A` only if `Int => B` matches `Int` and if `B = A`. But it is impossible for `Int => B` to match `Int`, no matter how we choose `B`.

We conclude that the expression `p[A](p[B])` has a problem: for any choice of `A` and `B`, some type will be mismatched. One says that the expression `p(p)` is **not well-typed** and does not **typecheck**. Such expressions contain a type error and are rejected by the Scala compiler.

We have seen that, for a code fragment containing function expressions, one can infer the most general type that makes all functions match their arguments, unless the code does not typecheck. The Damas-Hindley-Milner algorithm[2] performs type inference (or determines that there is a type error) for a large class of expressions containing functions, tuples, and disjunctive types.

4.4 Summary

Table 4.1 shows the notations introduced in this chapter.

What can we do with this chapter's techniques?

- Implement functions that return new functions and/or take functions as arguments.
- Simplify function expressions symbolically, when functions are applied to arguments.
- Infer the most general type for a given code expression (perform type inference).
- Convert functions to a fully parametric form when possible.

The following examples and exercises illustrate these techniques.

4.4.1 Solved examples

Example 4.4.1.1 Implement a function that applies a given function f repeatedly to an initial value x_0, until a given function `cond` returns `true`:

```
def converge[X](f: X => X, x0: X, cond: X => Boolean): X = ???
```

[2] https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system#Algorithm_W

Solution We call `find` on an iterator that keeps applying `f`; this stops when the condition is `true`:

```
def converge[X](f: X => X, x0: X, cond: X => Boolean): X =
  Stream.iterate(x0)(f)   // Type is Stream[X].
  .find(cond)             // Type is Option[X].
  .get                    // Type is X.
```

The method `get` is a partial function that can be applied only to non-empty `Option` values. It is safe to call `get` here, because the stream is unbounded and, if the condition `cond` never becomes `true`, the program will run out of memory (since `Stream.iterate` keeps all computed values in memory) or the user will run out of patience. So `.find(cond)` can never return an empty `Option` value. Of course, it is not satisfactory that the program crashes when the sequence does not converge. Exercise 4.4.2.2 will implement a safer version of this function by limiting the allowed number of iterations.

A tail-recursive implementation that works in constant memory is:

```
@tailrec def converge[X](f: X => X, x0: X, cond: X => Boolean): X =
  if (cond(x0)) x0 else converge(f, f(x0), cond)
```

To test this code, compute an approximation to $\sqrt{q}$ by Newton's method with the iteration function $f(x) = \frac{1}{2}\left(x + \frac{q}{x}\right)$. We iterate $f(x)$ starting with $x_0 = q/2$ until a given precision is obtained:

```
def approx_sqrt(q: Double, precision: Double): Double = {
      def cond(x: Double): Boolean = math.abs(x * x - q) <= precision
      def iterate_sqrt(x: Double): Double = 0.5 * (x + q / x)
      converge(iterate_sqrt, q / 2, cond)
}
```

Newton's method for $\sqrt{q}$ is guaranteed to converge when $q \geq 0$. Test it:

```
scala> approx_sqrt(25, 1.0e-8)
res0: Double = 5.000000000016778
```

Example 4.4.1.2 Using both `def` and `val`, define a Scala function that takes an integer `x` and returns a function that adds `x` to *its* argument.

Solution Let us first write down the required type signature: the function must take an integer argument `x: Int`, and the return value must be a function of type `Int => Int`:

```
def add_x(x: Int): Int => Int = ???
```

We are required to return a function that adds `x` to its argument. Let us call that argument `z`, to avoid confusion with the `x`. So, we are required to return the function `{ z => z + x }`. Since functions are values, we return a new function by writing a nameless function expression:

```
def add_x(x: Int): Int => Int = { z => z + x }
```

To implement the same function by using a `val`, we first convert the type signature of `add_x` to the equivalent curried type Int → Int → Int. Now we can write the Scala code of a function `add_x_v`:

```
val add_x_v: Int => Int => Int = { x => z => z + x }
```

The function `add_x_v` is equal to `add_x` except for using the `val` syntax instead of `def`. We do not need to write the type of the arguments `x` and `z` since we already wrote the type Int → Int → Int of `add_x_v`.

Example 4.4.1.3 Using `def` and `val`, implement a curried function `prime_f` that takes a function f and an integer x, and returns `true` when $f(x)$ is prime. Use the function `isPrime` from Section 1.1.2.

Solution First, determine the required type signature of `prime_f`. The value $f(x)$ must have type `Int`, or else we cannot check whether it is prime. So, f must have type Int → Int. Since `prime_f` should be a curried function, we need to put each argument into its own set of parentheses:

```
def prime_f(f: Int => Int)(x: Int): Boolean = ???
```

To implement `prime_f`, we need to return the result of `isPrime` applied to `f(x)`. A simple solution is:

```
def prime_f(f: Int => Int)(x: Int): Boolean = isPrime(f(x))
```

To implement the same function using `val`, rewrite its type signature as:

```
val prime_f: (Int => Int) => Int => Boolean = ???
```

(The parentheses around `Int => Int` are mandatory since `Int => Int => Int => Boolean` would be a completely different type.) The implementation is:

```
val prime_f: (Int => Int) => Int => Boolean = { f => x => isPrime(f(x)) }
```

The code `isPrime(f(x))` is a forward composition of the functions `f` and `isPrime`, so we can write:

```
val prime_f: (Int => Int) => Int => Boolean = (f => f andThen isPrime)
```

A nameless function of the form `f => f.something` is equivalent to a shorter Scala syntax `(_.something)`. So we finally rewrite the code of `prime_f` as:

```
val prime_f: (Int => Int) => Int => Boolean = (_ andThen isPrime)
```

Example 4.4.1.4 Implement a function `choice(x,p,f,g)` that takes a value x, a predicate p, and two functions f and g. The return value must be $f(x)$ if $p(x)$ returns `true`; otherwise the return value must be $g(x)$. Infer the most general type for this function.

Solution The code of this function must be:

```
def choice[...](x,p,f,g) = if (p(x)) f(x) else g(x)
```

To infer the most general type for this code, begin by assuming that x has type A, where A is a type parameter. Then the predicate p must have type `A => Boolean`. Since p is an arbitrary predicate, the value $p(x)$ will be sometimes `true` and sometimes `false`. So, `choice(x,p,f,g)` will sometimes compute $f(x)$ and sometimes $g(x)$. It follows that type A must be the argument type of both f and g, which means that the most general types so far are $f^{:A\to B}$ and $g^{:A\to C}$, yielding the type signature:

$$\text{choice}(x^{:A}, p^{:A\to\text{Boolean}}, f^{:A\to B}, g^{:A\to C}) \quad .$$

What could be the return type of `choice(x,p,f,g)`? If $p(x)$ returns `true`, the function `choice` returns $f(x)$, which is of type B. Otherwise, `choice` returns $g(x)$, which is of type C. However, the type signature of `choice` must be fixed in advance (at compile time) and cannot depend on the value $p(x)$ computed at run time. So, the types of $f(x)$ and of $g(x)$ must be the same, $B = C$. The type signature of `choice` will thus have only two type parameters, A and B:

```
def choice[A, B](x: A, p: A => Boolean, f: A => B, g: A => B): B =  if (p(x)) f(x) else g(x)
```

Example 4.4.1.5 Infer the most general type for the fully parametric function:

```
def q[...]: ... = { f => g => g(f) }
```

What types are inferred for the expressions `q(q)` and `q(q(q))`?

Solution Begin by assuming $f^{:A}$ with a type parameter A. In the sub-expression $g \to g(f)$, the curried argument g must itself be a function, because it is being applied to f as $g(f)$. So we assign types as $f^{:A} \to g^{:A\to B} \to g(f)$, where A and B are type parameters. Then the final returned value $g(f)$ has type B. Since there are no other constraints on the types, the types A and B remain arbitrary, so we add them to the type signature:

```
def q[A, B]: A => (A => B) => B = { f => g => g(f) }
```

To match types in the expression `q(q)`, we first assume arbitrary type parameters and write `q[A, B](q[C, D])`. We need to introduce new type parameters C, D because these type parameters may need to be set differently from A, B when we try to match the types in the expression `q(q)`.

The type of the first curried argument of `q[A, B]`, which is A, must match the entire type of `q[C, D]`, which is $C \to (C \to D) \to D$. So we must set the type parameter A as:

$$A = C \to (C \to D) \to D \quad .$$

The type of `q(q)` becomes:

$$q^{A,B}(q^{C,D}) : ((C \to (C \to D) \to D) \to B) \to B \quad , \quad \text{where} \quad A = C \to (C \to D) \to D \quad .$$

There are no other constraints on the type parameters B, C, D.

We use this result to infer the most general type for `q(q(q))`. We may denote $r \stackrel{\text{def}}{=} q(q)$ for brevity; then, as we just found, r has type $((C \to (C \to D) \to D) \to B) \to B$. To infer types in the expression `q(r)`, we introduce new type parameters E, F and write `q[E, F](r)`. The type of the argument of `q[E, F]` is E, and this must be the same as the type of r. This gives the constraint:

$$E = ((C \to (C \to D) \to D) \to B) \to B \quad .$$

Other than that, the type parameters are arbitrary. The type of the expression `q(q(q))` is $(E \to F) \to F$. We conclude that the most general type of `q(q(q))` is:

$$\begin{aligned} q^{E,F}(q^{A,B}(q^{C,D})) : ((((C \to (C \to D) \to D) \to B) \to B) \to F) \to F \quad , \\ \text{where} \quad A = C \to (C \to D) \to D \\ \text{and} \quad E = ((C \to (C \to D) \to D) \to B) \to B \quad . \end{aligned}$$

It is clear from this derivation that expressions such as `q(q(q(q)))`, `q(q(q(q(q))))`, etc., are well-typed.

Let us test these results in Scala, renaming the type parameters for clarity to `A`, `B`, `C`, `D`:

```
scala> def qq[A, B, C]: ((A => (A => B) => B) => C) => C = q(q)
qq: [A, B, C]=> ((A => ((A => B) => B)) => C) => C

scala> def qqq[A, B, C, D]: ((((A => (A => B) => B) => C) => C) => D) => D = q(q(q))
qqq: [A, B, C, D]=> ((((A => ((A => B) => B)) => C) => C) => D) => D
```

We did not need to write any type parameters within the expressions `q(q)` and `q(q(q))` because the full type signature was declared for each of these expressions. Since the Scala compiler did not print any error messages, we are assured that the types match correctly.

Example 4.4.1.6 Infer types in the code expression:

$$(f \to g \to g(f))\,(f \to g \to g(f))\,(f \to f(10)) \quad ,$$

and simplify the code through symbolic calculations.

Solution The given expression is a curried function $f \to g \to g(f)$ applied to two curried arguments. The plan is to consider each of these sub-expressions in turn, assigning types for them using type parameters, and then to figure out how to set the type parameters so that all types match.

Begin by renaming the shadowed variables (f and g) to remove shadowing:

$$(f \to g \to g(f))\,(x \to y \to y(x))\,(h \to h(10)) \quad . \tag{4.10}$$

As we have seen in Example 4.4.1.5, the sub-expression $f \to g \to g(f)$ is typed as $f^{:A} \to g^{:A \to B} \to g(f)$, where A and B are some type parameters. The sub-expression $x \to y \to y(x)$ is the same function as $f \to g \to g(f)$ but with possibly different type parameters, say, $x^{:C} \to y^{:C \to D} \to y(x)$. The types A, B, C, D are so far unknown.

Finally, the variable h in the sub-expression $h \to h(10)$ must have type $\text{Int} \to E$, where E is another type parameter. So, the sub-expression $h \to h(10)$ is a function of type $(\text{Int} \to E) \to E$.

The types must match in the entire expression (4.10):

$$(f^{:A} \to g^{:A \to B} \to g(f))(x^{:C} \to y^{:C \to D} \to y(x))(h^{:\text{Int} \to E} \to h(10) \quad . \tag{4.11}$$

It follows that f must have the same type as $x \to y \to y(x)$, while g must have the same type as $h \to h(10)$. The type of g, which we know as $A \to B$, will match the type of $h \to h(10)$, which we know as $(\text{Int} \to E) \to E$, only if $A = (\text{Int} \to E)$ and $B = E$. It follows that f has type $\text{Int} \to E$. At the

same time, the type of f must match the type of $x \to y \to y(x)$, which is $C \to (C \to D) \to D$. This can work only if $C = \text{Int}$ and $E = (C \to D) \to D = (\text{Int} \to D) \to D$.

In this way, we have found all the relationships between the type parameters A, B, C, D, E in Eq. (4.11). The type D remains arbitrary, while the type parameters A, B, C, E are expressed as:

$$A = \text{Int} \to (\text{Int} \to D) \to D \quad , \tag{4.12}$$

$$B = E = (\text{Int} \to D) \to D \quad , \tag{4.13}$$

$$C = \text{Int} \quad .$$

The entire expression in Eq. (4.11) is a saturated application of a curried function, and thus has the same type as the "final" result expression $g(f)$, which has type B. So, the entire expression in Eq. (4.11) has type $B = (\text{Int} \to D) \to D$.

Having established that types match, we can now omit the type annotations and rewrite the code:

$$\begin{aligned}
& (f \to g \to \underline{g}(\underline{f}))\,(x \to y \to y(x))\,(h \to h(10)) \\
\text{substitute } f = x \to y \to y(x) \text{ and } g = h \to h(10): \quad & = (h \to \underline{h}(10))\,(x \to y \to y(x)) \\
\text{substitute } h: \quad & = (x \to y \to y(\underline{x}))(10) \\
\text{substitute } x: \quad & = y \to y(10) \quad .
\end{aligned}$$

The type of this expression is $(\text{Int} \to D) \to D$ with a type parameter D. Since the argument y is an arbitrary function, we cannot simplify $y(10)$ or $y \to y(10)$ any further. We conclude that the final simplified form of Eq. (4.10) is $y^{:\text{Int}\to D} \to y(10)$.

To test this, we first define the function $f \to g \to g(f)$ as in Example 4.4.1.5:

```
def q[A, B]: A => (A => B) => B = { f => g => g(f) }
```

We also define the function $h \to h(10)$ with a general type $(\text{Int} \to E) \to E$:

```
def r[E]: (Int => E) => E = { h => h(10) }
```

To help Scala evaluate Eq. (4.11), we need to set the type parameters for the first `q` function as `q[A, B]` where A and B are given by Eqs. (4.12)–(4.13):

```
scala> def s[D] = q[Int => (Int => D) => D, (Int => D) => D](q)(r)
s: [D]=> (Int => D) => D
```

To verify that the function s^D indeed equals $y^{:\text{Int}\to D} \to y(10)$, we apply s^D to some functions of type $\text{Int} \to D$, say, for $D = \text{Boolean}$ or $D = \text{Int}$:

```
scala> s(_ > 0)  // Set D = Boolean and evaluate (10 > 0).
res6: Boolean = true

scala> s(_ + 20) // Set D = Int and evaluate (10 + 20).
res7: Int = 30
```

Example 4.4.1.7 Compute $(x \to y \to x(x(y))) \fatsemi (p \to p(2)) \fatsemi (z \to z+3)$ symbolically and infer types.

Solution The forward composition $f \fatsemi g$ substitutes the *body* of f into the argument of g:

$$\text{substitute } y = f(x): \quad (x \to f(x)) \fatsemi (y \to \underline{g(y)}) = (x \to g(f(x))) \quad .$$

This allows us to compute the forward compositions left to right:

$$(x \to y \to x(x(y))) \fatsemi (p \to p(2)) = x \to (y \to x(x(y)))(2) = x \to x(x(2)) \quad .$$

$$(x \to x(x(2))) \fatsemi (z \to z+3) = x \to x(x(2)) + 3 \quad .$$

Computing the pairwise combinations in another order, we get:

$$(p \to p(2)) \fatsemi (z \to z+3) = p \to p(2) + 3 \quad .$$

$$(x \to y \to x(x(y))) \fatsemi (p \to p(2)+3) = x \to (y \to x(x(y)))\,(2) + 3 = x \to x(x(2)) + 3 \quad .$$

Types are inferred as: $(x \to y \to x(x(y)))^{:(\text{Int}\to\text{Int})\to(\text{Int}\to\text{Int})} \fatsemi (p \to p(2))^{:(\text{Int}\to\text{Int})\to\text{Int}} \fatsemi (z \to z+3)^{:\text{Int}\to\text{Int}}$.

Example 4.4.1.8 We are given a function $q^{:A\to A}$ and we only know that for any $f^{:A\to A}$ the law $f \fatsemi q = q \fatsemi f$ holds (i.e., q commutes with every function). Show that q must be an identity function.

Solution Since the law must hold for any f, we may choose f at will. Let us fix a value $z^{:A}$ and choose $f \stackrel{\text{def}}{=} _ \to z$, that is, a constant function returning z. Applying both sides of the law $f \fatsemi q = q \fatsemi f$ to an arbitrary $x^{:A}$, we get:

$$(f \fatsemi q)(x) = q(f(x)) = q(z) \quad , \qquad (q \fatsemi f)(x) = f(q(x)) = z \quad .$$

It follows that $q(z) = z$ for any chosen $z^{:A}$. In other words, q is an identity function ($\text{id}^{:A\to A}$).

4.4.2 Exercises

Exercise 4.4.2.1 Revise the function from Exercise 1.6.2.4, implementing it as a curried function and replacing the hard-coded number 100 by a *curried* first argument. The type signature should become `Int => List[List[Int]] => List[List[Int]]`.

Exercise 4.4.2.2 Implement the function `converge` from Example 4.4.1.1 as a curried function with an additional argument to set the maximum number of iterations, returning `Option[Double]` as the final result type. The new version of `converge` should return `None` if the convergence condition is not satisfied after the given maximum number of iterations. The type signature and an example test:

```
@tailrec def convergeN[X](cond: X => Boolean)(x0: X)(maxIter: Int)(f: X => X): Option[X] = ???

scala> convergeN[Int](_ < 0)(0)(10)(_ + 1) // This does not converge.
res0: Option[Int] = None

scala> convergeN[Double]{ x => math.abs(x * x - 25) < 1e-8 }(1.0)(10) { x => 0.5 * (x + 25 / x ) }
res1: Option[Double] = Some(5.000000000053722)
```

Exercise 4.4.2.3 Implement a fully parametric, information-preserving, curried function that recovers an error using a given function argument. The type signature and an example test:

```
def recover[E, A]: Option[Either[E, A]] => (E => A) => Option[A] = ???

scala> recover(Some(Left("error"))) { _ => 123 }
res0: Option[Int] = Some(123)
```

Exercise 4.4.2.4 For `id` and `const` as defined above, what are the types of `id(id)`, `id(id)(id)`, `id(id(id))`, `id(const)`, and `const(const)`? Simplify these code expressions by symbolic calculations.

Exercise 4.4.2.5 For the function `twice` from Example 4.3.2.2, infer the most general type for the function `twice(twice(twice)))`. What does that function do? Test your answer on an example.

Exercise 4.4.2.6 Define a function `thrice` similarly to `twice` except it should apply a given function 3 times. What does the function `thrice(thrice(thrice)))` do?

Exercise 4.4.2.7 Define a function `ence` similarly to `twice` except it should apply a given function n times, where n is an additional curried argument.

Exercise 4.4.2.8 Define a fully parametric function `flip(f)` that swaps arguments for any given function `f` having two arguments. To test:

```
def f(x: Int, y: Int) = x - y     // Expect f(10, 2) == 8.
val g = flip(f)                   // Now expect g(2, 10) == 8.

scala> assert( f (10, 2) == 8 && g(2, 10) == 8 )
```

Exercise 4.4.2.9 Write a function `curry2` converting an uncurried function of type `(Int, Int) => Int` into an equivalent curried function of type `Int => Int => Int`.

Exercise 4.4.2.10 Apply the function $(x \to _ \to x)$ to the value $(z \to z(q))$ where $q^{:Q}$ is a given value of type Q. Infer types in these expressions.

Exercise 4.4.2.11 Infer types in the following expressions and test in Scala:

(a) $p \to q \to p(t \to t(q))$. **(b)** $p \to q \to q(x \to x(p(q)))$.

Exercise 4.4.2.12 Show that the following expressions cannot be well-typed:

(a) $p \to p(q \to q(p))$. **(b)** $p \to q \to q(x \to p(q(x)))$.

Exercise 4.4.2.13 Infer types and simplify the following code expressions by symbolic calculations:

(a) $q \to (x \to y \to z \to x(z)(y(z)))\,(a \to a)\,(b \to b(q))$.

(b) $(f \to g \to h \to f(g(h)))\,(x \to x)$.

(c) $(x \to y \to x(y))\,(x \to y \to x)$.

(d) $(x \to y \to x(y))\,(x \to y \to y)$.

(e) $x \to (f \to y \to f(y)(x))\,(z \to _ \to z)$.

(f) $z \to (x \to y \to x)\,(x \to x(z))\,(y \to y(z))$.

Exercise 4.4.2.14 Infer types and simplify the following code expressions by symbolic calculations:

(a) $(z \to z+1) \fatsemi (x \to y \to x/y) \fatsemi (p \to p(2))$.

(b) $(p \to q \to p+q+1) \fatsemi (f \to f \fatsemi f) \fatsemi (x \to x(1))$.

Exercise 4.4.2.15* In the following statements, the types A and B are fixed, and functions are *not* assumed to be fully parametric in A or B.

(a) Given a function $h^{:A\to B}$ that satisfies the law $f^{:A\to A} \fatsemi h^{:A\to B} = h^{:A\to B}$ for any $f^{:A\to A}$, prove that the function h must ignore its argument and return a fixed value of type B.

(b) We are given two functions $g^{:A\to A}$ and $h^{:B\to B}$. We know only that g and h satisfy the law $f^{:A\to B} \fatsemi h^{:B\to B} = g^{:A\to A} \fatsemi f^{:A\to B}$ for any function $f^{:A\to B}$. Prove that both g and h must be equal to identity functions of suitable types: $g^{:A\to A} = \text{id}^A$ and $h^{:B\to B} = \text{id}^B$.

Hint: choose f to be a suitable *constant* function and substitute f into the given laws.

4.5 Discussion and further developments

4.5.1 Higher-order functions

The **order** of a function is the number of function arrows (`=>`) contained in the type signature of that function. If a function's type signature contains more than one arrow, the function is called a **higher-order** function. Higher-order functions take functions as arguments and/or return functions.

The methods `andThen`, `compose`, `curried`, and `uncurried` are examples of higher-order functions that take other functions as arguments *and* return new functions.

The following examples illustrate the concept of a function's order. Consider the code:

```
def f1(x: Int): Int = x + 10
```

The function `f1` has type signature `Int => Int` and order 1, so it is *not* a higher-order function.

```
def f2(x: Int): Int => Int = (z => z + x)
```

The function `f2` has type signature `Int => Int => Int` and is a higher-order function of order 2.

```
def f3(g: Int => Int): Int = g(123)
```

The function `f3` has type signature `(Int => Int) => Int` and is a higher-order function of order 2.

Note that `f2` is a higher-order function only because its return value is of a function type. An equivalent computation can be performed by an uncurried function that is *not* higher-order:

```
scala> def f2u(x: Int, z: Int): Int = z + x    // Type signature (Int, Int) => Int
```

Unlike `f2`, the function `f3` *cannot* be converted to a non-higher-order function because `f3` has an argument of a function type. Converting to an uncurried form cannot eliminate such arguments.

4.5.2 Name shadowing and the scope of bound variables

Bound variables are introduced in nameless functions whenever an argument is defined. For example, in the nameless function $x \to y \to x + y$, the bound variables are the curried arguments x and y. The variable y is only defined within the scope $(y \to x + y)$ of the inner function; the variable x is defined within the entire scope of $x \to y \to x + y$.

Another way of introducing bound variables in Scala is to write a `val` or a `def` within curly braces:

```
val x = {
  val y = 10          // Bound variable ‘y‘.
  y + y * y
}     // Same as ‘val x = 10 + 10 * 10‘.
```

A bound variable is invisible outside the scope that defines it. So, it is easy to rename a bound variable: no outside code could possibly use it and depend on its value.

However, outside code may define a variable that (by chance) has the same name as a bound variable inside the scope. Consider this example from calculus: In the integral:

$$f(x) = \int_0^x \frac{dx}{1+x} \quad ,$$

two bound variables named x are defined in two scopes: one in the scope of f, another in the scope of the nameless function $x \to \frac{1}{1+x}$. The convention in mathematics is to treat these two x's as two *completely different* variables that just happen to have the same name. In sub-expressions where both of these bound variables are visible, priority is given to the bound variable defined in the smaller inner scope. The outer definition of x is then **shadowed** (hidden) by the inner definition of x. For this reason, evaluating $f(10)$ will give:

$$f(10) = \int_0^{10} \frac{dx}{1+x} = \log_e(11) \approx 2.398 \quad ,$$

rather than $\int_0^{10} \frac{dx}{1+10} = \frac{10}{11}$. The outer definition $x = 10$ is shadowed within the expression $\frac{1}{1+x}$ by the definition of x in the smaller local scope of $x \to \frac{1}{1+x}$.

Since this is the standard mathematical convention, the same convention is adopted in functional programming. A variable defined in a function scope (i.e., a bound variable) will shadow any outside definitions of a variable with the same name.

Name shadowing is not advisable in practical programming, because it usually decreases the clarity of code and so invites errors. Consider the nameless function

$$x \to x \to x \quad ,$$

and let us decipher this confusing syntax. The symbol $\to$ groups to the right, so $x \to x \to x$ is the same as $x \to (x \to x)$. It is a function that takes x and returns $x \to x$. Since the argument x in $(x \to x)$ may be renamed to y without changing the function, we can rewrite the code to:

$$x \to (y \to y) \quad .$$

Having removed name shadowing, we can more easily understand this code and reason about it. For instance, it becomes clear that this function ignores its argument x and always returns the same value (the identity function $y \to y$). So we can rewrite $(x \to x \to x)$ as $(_ \to y \to y)$, which is clearer.

4.5.3 Operator syntax for function applications

In mathematics, function applications are sometimes written without parentheses, for instance $\cos x$ or $\sin z$. Formulas such as $2 \sin x \cos x$ imply parentheses as $2 \cdot \sin(x) \cdot \cos(x)$. Functions such as

$\cos x$ are viewed as "operators" that are applied to their arguments without parentheses, similar to the operators of summation $\sum_k f(k)$ and differentiation $\frac{d}{dx} f(x)$.

Many programming languages (such as ML, OCaml, F#, Haskell, Elm, PureScript) have adopted this "operator syntax", making parentheses optional for function arguments so that `f x` means the same as `f(x)`. Parentheses are still used where necessary to avoid ambiguity or for readability.[3]

The conventions for nameless functions in the operator syntax become:

- Function expressions group to the right, so $x \to y \to z \to e$ means $x \to (y \to (z \to e))$.
- Function applications group to the left, so $f\, x\, y\, z$ means $\big((f\, x)\, y\big)\, z$.
- Function applications group stronger than infix operations, so $f\, x + y$ means $(f\, x) + y$, just as in mathematics "$\cos x + y$" groups "$\cos x$" stronger than the infix "+" operation.

Thus, $x \to y \to a\, b\, c + p\, q$ means $x \to (y \to ((a\, b)\ c) + (p\, q))$. When this notation becomes hard to read correctly, one needs to add parentheses, e.g., writing $f(x \to g\, h)$ instead of $f\, x \to g\, h$.

This book will not use the "operator syntax" when reasoning about code. Scala does not support the parentheses-free operator syntax; parentheses are needed around each curried argument.

In programming language theory, curried functions are "simpler" because they always have a *single* argument (but may return a function that will consume further arguments). From the point of view of programming practice, curried functions are often harder to read and to write.

In the operator syntax, a curried function `f` is applied to curried arguments as, e.g., `f 20 4`. This departs further from the mathematical tradition and requires some getting used to. If the two arguments are more complicated than just 20 and 4, the resulting expression may become harder to read, compared with the syntax where commas are used to separate the arguments. (Consider, for instance, the expression `f (g 10) (h 20) + 30`.) To improve readability of code, programmers may prefer to define names for complicated expressions and then use those names as curried arguments.

In Scala, the choice of whether to use curried or uncurried function signatures is largely a matter of syntactic convenience. Most Scala code tends to be written with uncurried functions, while curried functions are used when they produce more easily readable code.

One of the syntactic features of Scala is the ability to give a curried argument using the curly brace syntax. Compare the two definitions of the function `summation` described in Section 1.7.4:

```
def summation1(a: Int, b: Int, g: Int => Int): Int = (a to b).map(g).sum

def summation2(a: Int, b: Int)(g: Int => Int): Int = (a to b).map(g).sum
```

These functions are applied to arguments like this:

```
scala> summation1(1, 10, { x => x*x*x + 2*x })
res0: Int = 3135

scala> summation2(1, 10) { x => x*x*x + 2*x }
res1: Int = 3135
```

The code that calls `summation2` is easier to read because the curried argument is syntactically separated from the rest of the code by curly braces. This is especially useful when the curried argument is itself a function with a complicated body, since Scala's curly braces syntax allows function bodies to contain local definitions (`val` or `def`) of new bound variables.

Another feature of Scala is the "dotless" method syntax: for example, `xs map f` is equivalent to `xs.map(f)` and `f andThen g` is equivalent to `f.andThen(g)`. The "dotless" syntax is available only for infix methods, such as `map`, defined on specific types such as `Seq`. In Scala 3, the "dotless" syntax will only work for methods having a special `infix` annotation. Do not confuse Scala's "dotless" method syntax with the operator syntax used in Haskell and other languages.

[3] The operator syntax has a long history in programming. It is used in Unix shell commands, for example `cp file1 file2`. In LISP-like languages, function applications are enclosed in parentheses but the arguments are space-separated, for example `(f 10 20)`. Operator syntax is also used in some programming languages such as Tcl, Groovy, and Coffeescript.

4.5.4 Deriving a function's code from its type

We have seen how the procedure of type inference derives the type signature from a function's code. A well-known algorithm for type inference is the Damas-Hindley-Milner algorithm,[4] with a Scala implementation available.[5]

It is remarkable that one can sometimes perform "code inference": derive a function's *code* from the function's type signature. We will now look at some examples of this.

Consider a fully parametric function that performs partial applications for arbitrary other functions. A possible type signature is:

```
def pa[A, B, C](x: A)(f: (A, B) => C): B => C = ???
```

The function `pa` substitutes a fixed argument value `x:A` into another given function `f`.

How can we implement `pa`? Since `pa(x)(f)` must return a function of type `B => C`, we have no choice other than to begin writing a nameless function in the code:

```
def pa[A, B, C](x: A)(f: (A, B) => C): B => C = { y: B =>
  ??? // Need to compute a value of type C in this scope.
}
```

In the inner scope, we need to compute a value of type `C`, and we have values `x:A`, `y:B`, and `f:(A, B)=>C`. How can we compute a value of type `C`? If we knew that `C = Int` when `pa(x)(f)` is applied, we could have simply selected a fixed integer value, say, `1`, as the value of type `C`. If we knew that `C = String`, we could have selected a fixed string, say, `"hello"`, as the value of type `C`. But a fully parametric function cannot use any knowledge of the types of its actual arguments.

So, a fully parametric function cannot produce a value of an arbitrary type `C` from scratch. The only way of producing a value of type `C` is by applying the function `f` to arguments of types `A` and `B`. Since the types `A` and `B` are arbitrary, we cannot obtain any values of these types other than `x:A` and `y:B`. So, the only way of getting a value of type `C` is to compute `f(x, y)`. Thus, the body of `pa` must be:

```
def pa[A, B, C](x: A)(f: (A, B) => C): B => C = { y => f(x, y) }
```

In this way, we have *unambiguously* derived the body of this function from its type signature, by assuming that the function must be fully parametric.

Another example is the operation of forward composition $f \fatsemi g$ viewed as a fully parametric function with type signature:

```
def before[A, B, C](f: A => B, g: B => C): A => C = ???
```

To implement `before`, we need to create a nameless function of type `A => C`:

```
def before[A, B, C](f: A => B, g: B => C): A => C = { x: A =>
  ??? // Need to compute a value of type C in this scope.
}
```

In the inner scope, we need to compute a value of type C from the values $x^{:A}$, $f^{:A\to B}$, and $g^{:B\to C}$. Since the type C is arbitrary, the only way of obtaining a value of type C is by applying g to an argument of type B. In turn, the only way of obtaining a value of type B is to apply f to an argument of type A. Finally, we have only one value of type A, namely $x^{:A}$. So, the only way of obtaining the required result is to compute $g(f(x))$.

We have unambiguously inferred the body of the function from its type signature:

```
def before[A, B, C](f: A => B, g: B => C): A => C = { x => g(f(x)) }
```

Chapter 5 will show how code can be derived from type signatures for a wide range of fully parametric functions.

[4] https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system
[5] http://dysphoria.net/2009/06/28/hindley-milner-type-inference-in-scala/

5 The logic of types. III. The Curry-Howard correspondence

Fully parametric functions (introduced in Section 4.2) perform operations so general that their code works in the same way for all types. An example of a fully parametric function is:

```
def before[A, B, C](f: A => B, g: B => C): A => C = { x => g(f(x)) }
```

We have seen in Section 4.5.4 that for certain functions of this kind one can derive the code unambiguously from the type signature. There exists a mathematical theory (called the **Curry-Howard correspondence**) that gives precise conditions for the possibility of deriving a function's code from its type. There is also a systematic derivation algorithm that either produces the function's code or proves that the given type signature cannot be implemented. This chapter describes the main results and applications of that theory to functional programming.

5.1 Values computed by fully parametric functions

5.1.1 Motivation

Consider the following sketch of a fully parametric function's Scala code:

```
def f[A, B, ...]: ... = {
  val x: Either[A, B] = ... // Some expression here.
  ... }
```

If this program compiles without type errors, it means that the types match and, in particular, that the function `f` is able to compute a value `x` of type `Either[A, B]`.

It is sometimes *impossible* to compute a value of a certain type within the body of a fully parametric function. For example, the fully parametric function `fmap` shown in Section 3.2.3.1 cannot compute any values of type `A`:

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  val x: A = ???            // Cannot compute x here!
  ... }
```

The reason is that a fully parametric function cannot compute values of type `A` from scratch without using a previously given value of type `A` and without applying a function that returns values of type `A`. In `fmap`, no values of type `A` are given as arguments; the given function `f: A => B` returns values of type `B` and not `A`. The code of `fmap` must perform pattern matching on a value of type `Option[A]`:

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  case None       =>
    val x: A = ??? // Cannot compute x here!
    ...
  case Some(a)    =>
    val x: A = a   // Can compute x in this scope.
    ... }
```

Since the case `None` has no values of type `A`, we are unable to compute a value `x` in that scope (as long as `fmap` remains a fully parametric function).

Being able to compute `x:A` "within the body of a function" means that, if needed, the function should be able to *return* `x` as a result value. This requires computing `x` in all cases, not just within one part of the `match` expression.

The body of `fmap` also cannot compute any values of type `B`. Since no arguments of type `B` are given, the only way of obtaining a value of type `B` would be to apply the function `f: A => B` to *some* value of type `A`. But we just saw that the body of `fmap` cannot compute any values of type `A`.

Another example where one cannot compute a value of a certain type is the following code:

```
def before[A, B, C](f: A => B, g: B => C): A => C = {
  //  val h: C => A = ???   // Cannot compute h here!
  a => g(f(a)) // Can compute a value of type A => C.
}
```

The body of `before` may only use the arguments `f` and `g`. We can compose `f` and `g` to get a value of type `A => C`. But it is impossible to compute a value `h` of type `C => A`, no matter what code we try to write. The reason is that the body of `before` has no given values of type `A` and no functions that return values of type `A`, so a nameless function such as `{c:C => ???}` cannot compute a return value of type `A`. Since a fully parametric function cannot create values of an arbitrary type `A` from scratch, computing `h` within the body of `before` seems to be impossible.

Can we prove rigorously that a value of type `C => A` cannot be computed within the body of `before`? Or, perhaps, a clever trick could produce a value of that type? So far, we only saw informal arguments about whether values of certain types can be computed. To make the arguments rigorous, we need to translate statements such as "a fully parametric function `before` can compute a value of type `C => A`" into mathematical formulas with rules for proving them true or false.

The first step towards a rigorous mathematical formulation of a property is to choose a precise notation. In Section 3.5.3, we denoted by $\mathcal{CH}(A)$ the proposition "the $\mathcal{C}$ode $\mathcal{H}$as a value of type A". By "the code" we now mean the body of a given fully parametric function. When writing the code of a function's body, we are allowed to use the values of the function's arguments, which may have other types (say, X, Y, ..., Z). So, the validity of the proposition $\mathcal{CH}(A)$ depends also on the types X, Y, ..., Z. It follows that a "$\mathcal{CH}$-proposition" is actually the following statement:

$$\mathcal{CH}\text{-proposition}: \quad \text{a fully parametric expression can compute a value of type } A \text{ using previously given values of types } X, Y, ..., Z \quad . \tag{5.1}$$

Here X, Y, ..., Z, A may be either type parameters or more complicated type expressions, such as $B \to C$ or $(C \to D) \to E$, built from other type parameters.

If values of types X, Y, ..., Z are given, it means we "already have" values of those types. So, the propositions $\mathcal{CH}(X), \mathcal{CH}(Y), ..., \mathcal{CH}(Z)$ will be already true. Thus, proposition (5.1) is equivalent to "$\mathcal{CH}(A)$ is true assuming $\mathcal{CH}(X), \mathcal{CH}(Y), ..., \mathcal{CH}(Z)$ are true". In mathematical logic, a statement of this form is called a **sequent** and is denoted using the symbol $\vdash$ (the "turnstile"):

$$\text{sequent}: \quad \mathcal{CH}(X), \mathcal{CH}(Y), ..., \mathcal{CH}(Z) \vdash \mathcal{CH}(A) \quad . \tag{5.2}$$

The assumptions $\mathcal{CH}(X)$, $\mathcal{CH}(Y)$, ..., $\mathcal{CH}(Z)$ are called **premises** and the proposition $\mathcal{CH}(A)$ is called the **goal** of the sequent.

Sequents provide a notation for the problem of computing values within the code of fully parametric functions. Since our goal is to solve such problems rigorously, we will need to be able to *prove* sequents of the form (5.2). Our previous examples are denoted by the following sequents:

$$\begin{aligned}
\texttt{fmap for Option}: &\quad \mathcal{CH}(\texttt{A => B}) \vdash \mathcal{CH}(\texttt{Option[A] => Option[B]}) \\
\text{the function } \texttt{before}: &\quad \mathcal{CH}(\texttt{A => B}), \mathcal{CH}(\texttt{B => C}) \vdash \mathcal{CH}(\texttt{A => C}) \\
\text{value of type } A \text{ within } \texttt{fmap}: &\quad \mathcal{CH}(\texttt{A => B}), \mathcal{CH}(\texttt{Option[A]}) \vdash \mathcal{CH}(\texttt{A}) \\
\text{value of type } C \to A \text{ within } \texttt{before}: &\quad \mathcal{CH}(\texttt{A => B}), \mathcal{CH}(\texttt{B => C}) \vdash \mathcal{CH}(\texttt{C => A})
\end{aligned}$$

In this section we gave informal arguments towards proving the first two sequents and disproving the last two. We will now develop tools for proving sequents. This will allow us to prove rigorously that a given fully parametric function can compute values of a certain type (or that it cannot).

In formal logic, sequents are proved by starting with certain axioms and following certain derivation rules. Different choices of axioms and derivation rules will give different *logics*. We will need to discover the correct logic for reasoning about $\mathcal{CH}$-propositions. That is, we will find a set of axioms and derivation rules that give correct answers about implementable and non-implementable types.

5.1.2 Type notation and $\mathcal{CH}$-propositions for standard type constructions

A proposition $\mathcal{CH}(A)$ may be true with one set of premises such as $\mathcal{CH}(X)$, $\mathcal{CH}(Y)$, ..., $\mathcal{CH}(Z)$ but false with another. Here and in the following sections, we will be reasoning about $\mathcal{CH}$-propositions

within the body of a *chosen* fully parametric function, i.e., with a fixed set of premises. We will then temporarily omit the premises and write just $\mathcal{CH}(A)$ instead of $\mathcal{CH}(X), \mathcal{CH}(Y), ..., \mathcal{CH}(Z) \vdash \mathcal{CH}(A)$. In later sections, we will need to write full sets of premises for sequents.

In Section 3.5.3 we saw examples of reasoning about $\mathcal{CH}$-propositions for case classes and for disjunctive types. We will now extend this reasoning systematically to all type constructions that fully parametric programs could use. A special type notation explained in this section will help us write type expressions more concisely. (See Appendix A for a summary of our type notation.)

There exist **six standard type constructions** supported by all functional languages: primitive types (including `Unit` type and the void type, called `Nothing` in Scala), tuples (also called **product types**), disjunctive types (also called **co-product types**), function types, parameterized types, and recursive types. We will now derive the rules for writing $\mathcal{CH}$-propositions for each of these type constructions (except recursive types).

1a) Rule for the `Unit` type The `Unit` type has only a single value `()`, an "empty tuple". This value can be *always* computed since it does not need any previous data:

```
def f[...]: ... = {
  ...
  val x: Unit = () // We can always compute a ‘Unit‘ value.
  ...
```

So, the proposition $\mathcal{CH}(\texttt{Unit})$ is always true. In the type notation, the `Unit` type is denoted by $\mathbb{1}$. We may write $\mathcal{CH}(\mathbb{1}) = True$.

Named unit types also have a single value that is always possible to compute. For example:

```
final case class N1()
```

defines a named unit type. We can compute a value of type `N1` without using any other given values:

```
val x: N1 = N1()
```

So, the proposition $\mathcal{CH}(\texttt{N1})$ is always true. Named unit types are also denoted by $\mathbb{1}$, just as the `Unit` type itself.

1b) Rule for the void type The Scala type `Nothing` has no values, so the proposition $\mathcal{CH}(\texttt{Nothing})$ is always false. The type `Nothing` is denoted by $\mathbb{0}$ in the type notation. So, the rule is $\mathcal{CH}(\mathbb{0}) = False$.

1c) Rule for primitive types For a specific primitive (or library-defined) type such as `Int` or `String`, the corresponding $\mathcal{CH}$-proposition is *always true* because we could use a constant value, e.g.:

```
def f[...]: ... {
   ...
   val x: String = "abc" // We can always compute a ‘String‘ value.
   ...
```

So, the rule for primitive types is the same as that for the `Unit` type. For example, $\mathcal{CH}(\text{String}) = True$.

2) Rule for tuple types To compute a value of a tuple type `(A, B)` requires computing a value of type `A` *and* a value of type `B`. This is expressed by the logic formula $\mathcal{CH}(\texttt{(A, B)}) = \mathcal{CH}(A) \wedge \mathcal{CH}(B)$. A similar formula holds for case classes, as Eq. (3.2) shows. In the type notation, the tuple `(A, B)` is written as $A \times B$. Tuples and case classes with more than two parts are denoted similarly as $A \times B \times ... \times C$. For example, the Scala definition:

```
case class Person(firstName: String, lastName: String, age: Int)
```

is written in the type notation as String × String × Int. So, the rule for tuple types is:

$$\mathcal{CH}\,(A \times B \times ... \times C) = \mathcal{CH}(A) \wedge \mathcal{CH}(B) \wedge ... \wedge \mathcal{CH}(C) \quad .$$

3) Rule for disjunctive types A disjunctive type may consist of several case classes. Having a value of a disjunctive type means to have a value of (at least) one of those case classes. An example of translating this relationship into a formula was shown by Eq. (3.1). For the standard disjunctive type `Either[A, B]`, we have the logical formula $\mathcal{CH}(\texttt{Either[A, B]}) = \mathcal{CH}(A) \vee \mathcal{CH}(B)$. In the type notation, the Scala type `Either[A, B]` is written as $A + B$. As another example, the Scala definition:

```
sealed trait RootsOfQ
final case class NoRoots()                    extends RootsOfQ
```

```
final case class OneRoot(x: Double)             extends RootsOfQ
final case class TwoRoots(x: Double, y: Double) extends RootsOfQ
```

is translated to the type notation as:

$$\text{RootsOfQ} \stackrel{\text{def}}{=} \mathbb{1} + \text{Double} + \text{Double} \times \text{Double} \quad .$$

Here, the type notation is significantly shorter because it omits all case class names and part names from the type definitions. Using the type notation, the rule for disjunctive types is written as:

$$\mathcal{CH}\,(A + B + ... + C) = \mathcal{CH}(A) \vee \mathcal{CH}(B) \vee ... \vee \mathcal{CH}(C) \quad .$$

4) Rule for function types Consider now a function type such as `A => B`. This type is written in the type notation as $A \to B$. To compute a value of that type, we need to write code like this:

```
val f: A => B = { (a: A) =>
  ??? // Compute a value of type B in this scope.
}
```

The inner scope of the function needs to compute a value of type B, and the given value `a:A` may be used for that. So, $\mathcal{CH}(A \to B)$ is true if and only if we are able to compute a value of type B when we are given a value of type A. To translate this statement into the language of logical propositions, we need to use the logical **implication**, $\mathcal{CH}(A) \Rightarrow \mathcal{CH}(B)$, which means that $\mathcal{CH}(B)$ can be proved if we already have a proof of $\mathcal{CH}(A)$. So, the rule for function types is:

$$\mathcal{CH}(A \to B) = \mathcal{CH}(A) \Rightarrow \mathcal{CH}(B) \quad .$$

5) Rule for parameterized types Here is an example of a function with type parameters:

```
def f[A, B]: A => (A => B) => B = { x => g => g(x) }
```

Being able to define the body of such a function is the same as being able to compute a value of type `A => (A => B) => B` for *all* possible types `A` and `B`. In the notation of formal logic, this is written as:

$$\mathcal{CH}\,(\forall(A, B).\, A \to (A \to B) \to B) \quad ,$$

and is equivalent to:

$$\forall(A, B).\, \mathcal{CH}\,(A \to (A \to B) \to B) \quad .$$

The symbol $\forall$ means "for all" and is called the **universal quantifier** in logic.

The type notation for the type signature of `f` is written in one of the following ways:

$$f^{A,B} : A \to (A \to B) \to B \quad , \quad \text{or equivalently :} \quad f : \forall(A, B).\, A \to (A \to B) \to B \quad .$$

The type quantifier ("for all A and B") indicates that f can be used with all types A and B.

In Scala, longer type expressions can be named and their names (called **type aliases**) can be used to make code shorter. Type aliases may also contain type parameters. Defining and using a type alias for the type of the function `f` looks like this:

```
type F[A, B] = A => (A => B) => B
def f[A, B]: F[A, B] = { x => g => g(x) }
```

This is written in the type notation by placing all type parameters into superscripts:

$$F^{A,B} \stackrel{\text{def}}{=} A \to (A \to B) \to B \quad ,$$
$$f^{A,B} : F^{A,B} \stackrel{\text{def}}{=} x^{:A} \to g^{:A \to B} \to g(x) \quad ,$$

or equivalently (although somewhat less readably) as:

$$f : \left(\forall(A, B).\, F^{A,B}\right) \stackrel{\text{def}}{=} \forall(A, B).\, x^{:A} \to g^{:A \to B} \to g(x) \quad .$$

In Scala 3, the function `f` can be written as a value (`val`) via this syntax:

Type construction	Scala syntax	Type notation	$\mathcal{CH}$-proposition
type parameter	`[A]`	A	$\mathcal{CH}(A)$
product type (tuple)	`(A, B)`	$A \times B$	$\mathcal{CH}(A) \wedge \mathcal{CH}(B)$
disjunctive type	`Either[A, B]`	$A + B$	$\mathcal{CH}(A) \vee \mathcal{CH}(B)$
function type	`A => B`	$A \to B$	$\mathcal{CH}(A) \Rightarrow \mathcal{CH}(B)$
unit or a "named unit" type	`Unit`	$\mathbb{1}$	$\mathcal{CH}(\mathbb{1}) = True$
primitive type	`Int`, `String`, ...	Int, String, ...	$\mathcal{CH}(\text{Int}) = True$
void type	`Nothing`	$\mathbb{0}$	$\mathcal{CH}(\mathbb{0}) = False$
value parameterized by type	`def f[A]: F[A]`	$f^A : F^A$	$\forall A. \mathcal{CH}(F^A)$
type with quantifier	`val f: [A] => F[A]` (Scala 3)	$f : \forall A. F^A$	$\forall A. \mathcal{CH}(F^A)$

Table 5.1: The correspondence between type constructions and $\mathcal{CH}$-propositions.

```
val f: [A, B] => A => (A => B) => B = {   // Valid only in Scala 3.
  [A, B] => (x: A) => (g: A => B) => g(x)
}
```

This syntax corresponds more closely to the code notation $\forall(A, B). x^{:A} \to g^{:A\to B} \to g(x)$.

So, the rule for parameterized types with the type notation F^A is:

$$\mathcal{CH}(\forall A. F^A) = \forall A. \mathcal{CH}(F^A) \quad .$$

Case classes and disjunctive types use *names* for the types and their parts. However, those names only add convenience for programmers and do not affect the computational properties of types. The type notation is designed to support nameless type expressions.

Table 5.1 summarizes the type notation and also shows how to translate it into logic formulas with propositions of the form $\mathcal{CH}(...)$.

The precedence of operators in the type notation is chosen to have fewer parentheses in the type expressions that are frequently used. The rules of precedence are:

- The type product operator ($\times$) groups stronger than the disjunctive operator (+), so that type expressions such as $A + B \times C$ have the same operator precedence as in standard arithmetic. That is, $A + B \times C$ means $A + (B \times C)$. This convention makes type expressions easier to reason about (for people familiar with arithmetic).

- The function type arrow ($\to$) groups weaker than the operators + and $\times$, so that often-used types such as $A \to \mathbb{1} + B$ (representing `A => Option[B]`) or $A \times B \to C$ (representing `((A, B)) => C`) can be written without any parentheses. Type expressions such as $(A \to B) \times C$ will require parentheses but are needed less often.

- The type quantifiers group weaker than all other operators, so we can write types such as $\forall A. A \to A \to A$ without parentheses. This is helpful because type quantifiers are most often placed outside a type expression. When this is not the case, parentheses are necessary, e.g., in the type expression $(\forall A. A \to A \to A) \to \mathbb{1} + \mathbb{1}$.

5.1.3 Solved examples: Type notation

From now on, we will prefer to write types in the type notation rather than in the Scala syntax. The type notation allows us to write nameless type expressions and , makes the structure of complicated

types more clear, compared with the Scala syntax. Names of types and parts of types are, of course, helpful for reminding programmers of the meaning of data in a program. However, writing names for every part of every type is not helpful for reasoning about the properties of types. Type notation makes reasoning about types easier, as we will see throughout this chapter. Once the programmer has finished deriving the necessary types and verifying their properties, the type notation can be straightforwardly translated into Scala code. Let us get some experience doing that.

Example 5.1.3.1 Define a function `delta` taking an argument `x` and returning the pair `(x, x)`. Derive the most general type for this function. Write the type signature of `delta` in the type notation, and translate it into a $\mathcal{CH}$-proposition. Simplify the $\mathcal{CH}$-proposition if possible.

Solution Begin by writing the code of the function:

```
def delta(x: ...) = (x, x)
```

To derive the most general type for `delta`, first assume `x:A`, where `A` is a type parameter; then the tuple `(x, x)` has type `(A, A)`. We do not see any constraints on the type parameter `A`. So, `A` represents an arbitrary type and needs to be added to the type signature of `delta`:

```
def delta[A](x: A): (A, A) = (x, x)
```

We find that the most general type of `delta` is `A => (A, A)`. We also note that there is only one way of implementing a fully parametric function with type signature `A => (A, A)`: the function must duplicate its given argument.

We will use the letter Δ for the function `delta`. In the type notation, the type signature of Δ is:

$$\Delta^A : A \to A \times A \quad .$$

So, the proposition $\mathcal{CH}(\Delta)$ (meaning "the function Δ can be implemented") is:

$$\mathcal{CH}(\Delta) = \forall A. \mathcal{CH}\left(A \to A \times A\right) \quad .$$

In the type expression $A \to A \times A$, the product symbol ($\times$) binds stronger than the function arrow ($\to$), so the parentheses in $A \to (A \times A)$ may be omitted.

Using the rules for transforming $\mathcal{CH}$-propositions, we rewrite:

$$\begin{aligned} & \mathcal{CH}(A \to A \times A) \\ \text{rule for function types :} \quad & = \mathcal{CH}(A) \Rightarrow \mathcal{CH}(A \times A) \\ \text{rule for tuple types :} \quad & = \mathcal{CH}(A) \Rightarrow (\mathcal{CH}(A) \wedge \mathcal{CH}(A)) \quad . \end{aligned}$$

Thus the proposition $\mathcal{CH}(\Delta)$ is equivalent to:

$$\mathcal{CH}(\Delta) = \forall A. \mathcal{CH}(A) \Rightarrow (\mathcal{CH}(A) \wedge \mathcal{CH}(A)) \quad .$$

Example 5.1.3.2 The standard types `Either[A, B]` and `Option[A]` are written in the type notation as:

$$\text{Either}^{A,B} \stackrel{\text{def}}{=} A + B \quad , \qquad \text{Opt}^A \stackrel{\text{def}}{=} \mathbb{1} + A \quad .$$

The type `Either[A, B]` is written as $A + B$ by definition of the disjunctive type notation (+). The type `Option[A]` has two disjoint cases, `None` and `Some[A]`. The case class `None` is a "named `Unit`" and is denoted by $\mathbb{1}$. The case class `Some[A]` contains a single value of type A. So, the type notation for `Option[A]` is $\mathbb{1} + A$. We will also sometimes write Opt^A to denote `Option[A]`.

Example 5.1.3.3 The Scala definition of the disjunctive type `UserAction`:

```
sealed trait UserAction
final case class SetName(first: String, last: String) extends UserAction
final case class SetEmail(email: String)              extends UserAction
final case class SetUserId(id: Long)                  extends UserAction
```

is written in the type notation as:

$$\text{UserAction} \stackrel{\text{def}}{=} \text{String} \times \text{String} + \text{String} + \text{Long} \quad . \tag{5.3}$$

The type operation $\times$ groups stronger than $+$, as in arithmetic. To derive the type notation (5.3), we first drop all names from case classes and get three nameless tuples `(String, String)`, `(String)`, and `(Long)`. Each of these tuples is then converted into a product using the operator $\times$, and all products are "summed" in the type notation using the operator $+$.

Example 5.1.3.4 The parameterized disjunctive type `Either3` is a generalization of `Either`:

```
sealed trait Either3[A, B, C]
final case class Left[A, B, C](x: A)   extends Either3[A, B, C]
final case class Middle[A, B, C](x: B) extends Either3[A, B, C]
final case class Right[A, B, C](x: C)  extends Either3[A, B, C]
```

This disjunctive type is written in the type notation as $\text{Either3}^{A,B,C} \stackrel{\text{def}}{=} A + B + C$.

Example 5.1.3.5 Define a Scala type constructor `F[A]` corresponding to the type notation:

$$F^A \stackrel{\text{def}}{=} \mathbb{1} + \text{Int} \times A \times A + \text{Int} \times (\text{Int} \to A) \quad .$$

Solution The formula for F^A defines a disjunctive type `F[A]` with three parts. To implement `F[A]` in Scala, we need to choose names for each of the disjoint parts, which will become case classes. For the purposes of this example, let us choose names `F1`, `F2`, and `F3`. Each of these case classes needs to have the same type parameter `A`. So we begin writing the code as:

```
sealed trait F[A]
final case class F1[A](...) extends F[A]
final case class F2[A](...) extends F[A]
final case class F3[A](...) extends F[A]
```

Each of these case classes represents one part of the disjunctive type: `F1` represents $\mathbb{1}$, `F2` represents $\text{Int} \times A \times A$, and `F3` represents $\text{Int} \times (\text{Int} \to A)$. It remains to choose names and define the case classes:

```
sealed trait F[A]
final case class F1[A]()                     extends F[A]  // Named unit type.
final case class F2[A](n: Int, x1: A, x2: A) extends F[A]
final case class F3[A](n: Int, f: Int => A)  extends F[A]
```

The names `n`, `x1`, `x2`, and `f` are chosen purely for convenience.

Example 5.1.3.6 Write the type signature of the following function in the type notation:

```
def fmap[A, B](f: A => B): Option[A] => Option[B]
```

Solution This is a curried function, so we first rewrite the type signature as:

```
def fmap[A, B]: (A => B) => Option[A] => Option[B]
```

The type notation for `Option[A]` is $\mathbb{1} + A$. Now we can write the type signature of `fmap` as:

$$\text{fmap}^{A,B} : (A \to B) \to \mathbb{1} + A \to \mathbb{1} + B \quad ,$$

$$\text{or equivalently :} \quad \text{fmap} : \forall (A, B).\ (A \to B) \to \mathbb{1} + A \to \mathbb{1} + B \quad .$$

We do not put parentheses around $\mathbb{1} + A$ and $\mathbb{1} + B$ because the function arrow ($\to$) groups weaker than the other type operations. Parentheses around $(A \to B)$ are required.

We will usually prefer to write type parameters in superscripts rather than under type quantifiers. So, for example, we will write $\text{id}^A \stackrel{\text{def}}{=} x^{:A} \to x$ rather than $\text{id} \stackrel{\text{def}}{=} \forall A.\, x^{:A} \to x$.

5.1.4 Exercises: Type notation

Exercise 5.1.4.1 Define a Scala disjunctive type `Q[T, A]` corresponding to the type notation:

$$Q^{T,A} \stackrel{\text{def}}{=} \mathbb{1} + T \times A + \text{Int} \times (T \to T) + \text{String} \times A \quad .$$

Exercise 5.1.4.2 Rewrite `Either[(A, Int), Either[(A, Char), (A, Float)]]` in the type notation.

Exercise 5.1.4.3 Define a Scala type `Opt2[A, B]` written in the type notation as $\mathbb{1} + A + B$.

Exercise 5.1.4.4 Write a Scala type signature for the fully parametric function:

$$\text{flatMap}^{A,B} : \mathbb{1} + A \to (A \to \mathbb{1} + B) \to \mathbb{1} + B$$

and implement this function, preserving information as much as possible.

5.2 The logic of $\mathcal{CH}$-propositions

5.2.1 Motivation and first examples

So far, we were able to convert statements such as "*a fully parametric function can compute values of type A*" into logical propositions that we called $\mathcal{CH}$-propositions. The next step is to determine the proof rules suitable for reasoning about $\mathcal{CH}$-propositions.

Formal logic uses axioms and derivation rules for proving that certain formulas are true or false. We will use Greek letters (α, β, etc.) to denote propositions in logic.

We will often need logical formulas that talk about properties of *arbitrary* propositions. This is denoted by using the **universal quantifier** symbol ($\forall$), pronounced "for all". The universal quantifier will be usually located in front of the formula, for example:

$$\forall(\alpha, \beta).\ (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \alpha \quad .$$

The symbol $\Rightarrow$ denotes **implication**: $\alpha \Rightarrow \beta$ means that *if* α can be proved true *then* β will be proved true.

Formulas whose propositions are universally quantified correspond to type signatures that are made entirely from type parameters. For instance, the formula shown above corresponds to the following type signature:

```
def f[A, B]: (A => B) => A => A
```

The universal quantifier $\forall(\alpha, \beta)$ corresponds to the fact that the function `f` works with any choice of types `A` and `B`.

A simple example of a true logical formula is "any proposition α follows from itself":

$$\forall\alpha.\, \alpha \Rightarrow \alpha \quad . \tag{5.4}$$

If the proposition α is a $\mathcal{CH}$-proposition, that is, if $\alpha \stackrel{\text{def}}{=} \mathcal{CH}(A)$ for some type A, we obtain from Eq. (5.4) the formula:

$$\forall A.\, \mathcal{CH}(A) \Rightarrow \mathcal{CH}(A) \quad . \tag{5.5}$$

We expect true $\mathcal{CH}$-propositions to correspond to types that *can* be computed in a fully parametric function. Let us see if this example fits our expectations. We can rewrite Eq. (5.5) as:

$$\begin{aligned} & \forall A.\, \underline{\mathcal{CH}(A) \Rightarrow \mathcal{CH}(A)} \\ \text{rule for function types :} \quad & = \underline{\forall A}.\, \mathcal{CH}\,(A \to A) \\ \text{rule for parameterized types :} \quad & = \mathcal{CH}\,(\forall A.\, A \to A) \quad . \end{aligned}$$

The last line shows a $\mathcal{CH}$-proposition that corresponds to the type $\forall A.\, A \to A$. Translating this type notation into a Scala type signature, we get:

```
def f[A]: A => A
```

This type signature can be easily implemented as an identity function:

```
def f[A]: A => A = { x => x }
```

So, in this example we see how a formula in logic is converted into a type signature of a function `f`, and we found that the function `f` can be implemented in code.

While the correctness of the formula $\forall\alpha.\alpha \Rightarrow \alpha$ may be self-evident, the point of using formal logic is to have a set of axioms and proof rules that allow us to verify *all* true formulas systematically, without guessing or testing. What axioms and proof rules are suitable for proving $\mathcal{CH}$-propositions?

A set of axioms and proof rules defines a **formal logic**. Mathematicians have studied many different logics that are useful for solving different problems. We are now looking for a specific formal logic that gives correct answers when reasoning about $\mathcal{CH}$-propositions.[1]

5.2.2 The rules of proof for $\mathcal{CH}$-propositions

To derive the suitable logical axioms and proof rules systematically, let us examine what makes a $\mathcal{CH}$-proposition true. A sequent $\mathcal{CH}(A) \vdash \mathcal{CH}(X)$ is true when a value of type X can be computed by fully parametric code that may only use a given value of type A. To describe all possible ways of computing a value of type X, we need to enumerate all possible ways of *writing code* for a fully parametric function body. The requirement of full parametricity means that we are not allowed to use any specific types such as `Int` or `String`, any concrete values such as `123` or `"hello"`, or any library functions that work with specific (non-parametric) types. We are only allowed to work with values of unknown types described by the given type parameters. However, we are permitted to use fully parametric types such as `Either[A, B]` or `Option[A]`.

In fact, we can enumerate all the allowed constructions that may be used by fully parametric code implementing a $\mathcal{CH}$-proposition. There are *eight* code constructions as illustrated by this code:

```
def f[A, B, ...](a: A, b: B): X = {   // Any given type signature.
  val x1: Unit = ()                   // 1) Use a value of type Unit.
  val x2: A = a                       // 2) Use a given argument.
  val x3 = { x: A => b }              // 3) Create a function.
  val x4: D = x3(x2)                  // 4) Use a function.
  val x5: (A, B) = (a, b)             // 5) Create a tuple.
  val x6: B = x5._2                   // 6) Use a tuple.
  val x7: Option[A] = Some(x2)        // 7) Create values of a disjunctive type.
  val x8 = x7 match { ... }           // 8) Use values of a disjunctive type.
}     // 9) Call f() itself recursively. Not included here because not supported by CH-propositions.
```

The proposition $\mathcal{CH}(X)$ is true if we can create a sequence of computed values such as `x1`, `x2`, ..., `xN`, each using one of these eight code constructs, with `xN` having type X. So, each of the eight code constructs will give a proof rule in the logic.

It is important that there are only a finite number of allowed code constructions. This defines rigorously the concept of "fully parametric code" and allows us to *prove* $\mathcal{CH}$-propositions.

Reasoning about proof rules is easier with a compact notation. To obtain that notation, we first need to translate the eight code constructs into the notation of sequents. A proof of a sequent, e.g., $\mathcal{CH}(A) \vdash \mathcal{CH}(X)$, will consist of applying some of those proof rules. We will then combine the code constructs corresponding to each rule and obtain some code that computes a value of type X using an argument of type A.

Conversely, any fully parametric (and non-recursive) code computing a value of type X must be a combination of some of the eight code constructs. That code combination can be translated into a combination of logic rules, which will produce a proof of the proposition $\mathcal{CH}(X)$.

In the following text, we will need to use the full formulation (5.1) of $\mathcal{CH}$-propositions and write them as sequents such as Eq. (5.2). For brevity, we define $\alpha \stackrel{\text{def}}{=} \mathcal{CH}(A)$, $\beta \stackrel{\text{def}}{=} \mathcal{CH}(B)$, etc. It is also customary to use the letter Γ to denote a set of premises, such as $\mathcal{CH}(X)$, $\mathcal{CH}(Y)$, ..., $\mathcal{CH}(Z)$ in Eq. (5.2). So, we can write a shorter formula $\Gamma \vdash \alpha$ instead of the sequent (5.2).

With these notations, we list the rules for proving $\mathcal{CH}$-propositions and the corresponding code:

[1] For an overview and more details about that logic and the necessary proof techniques, see the book by R. Bornat, "Proof and disproof in formal logic: an introduction for programmers". An early draft version of that book is available at https://homepages.phonecoop.coop/randj/richard/books/ProofandDisproof.pdf

1) Use the `Unit` value At any place in the code, we may write the expression `()` of type `Unit`. This expression corresponds to a proof of the proposition $\mathcal{CH}(\mathbb{1})$ with any set Γ of premises (even with an empty set of premises). So, the sequent $\Gamma \vdash \mathcal{CH}(\mathbb{1})$ is always true. The code corresponding to the proof of this sequent is an expression that creates a value of the `Unit` type:

$$\text{Proof}\big(\Gamma \vdash \mathcal{CH}(\mathbb{1})\big) = 1 \quad ,$$

where we denoted by 1 the value `()`.

In formal logic, a sequent that is always true, such as our $\Gamma \vdash \mathcal{CH}(\mathbb{1})$, is called an **axiom** and is written in the following notation:

$$\frac{}{\Gamma \vdash \mathcal{CH}(\mathbb{1})} \quad \text{(create unit)} \qquad .$$

The "fraction with a label" represents a proof rule. The denominator of the "fraction" is the target sequent that we need to prove. The numerator of the "fraction" can have zero or more other sequents that need to be proved before the target sequent can be proved. In this case, the set of previous sequents is empty: the target sequent is an axiom and so requires no previous sequents for its proof. The label "create unit" is an arbitrary name used to refer to the rule.

2) Use a given argument At any place within the code of a fully parametric function, we may use one of the function's arguments, say $x^{:A}$. If some argument has type A, it means that $\alpha \stackrel{\text{def}}{=} \mathcal{CH}(A)$ belongs to the set of premises of the sequent we are trying to prove. To indicate this, we write the set of premises as "Γ, α". The code construct `x:A` computes a value of type A, i.e., show that α is true, given these premises. This is expressed by the sequent $\Gamma, \alpha \vdash \alpha$. The proof of this sequent corresponds to an expression that returns one of the given arguments (which we here called $x^{:A}$):

$$\text{Proof}\big(\Gamma, \alpha \vdash \alpha\big) = x^{:A} \quad .$$

This sequent is an axiom since its proof requires no previous sequents. We denote this axiom by:

$$\frac{}{\Gamma, \alpha \vdash \alpha} \quad \text{(use arg)} \qquad .$$

3) Create a function At any place in the code, we may compute a nameless function of type, say, $A \to B$, by writing `(x:A) => expr` as long as a value `expr` of type B can be computed in the inner scope of the function. The code for `expr` is also required to be fully parametric; it may use `x` and/or other values visible in that scope. So we now need to answer the question of whether a fully parametric function can compute a value of type B, given an argument of type A as well as all other arguments previously given to the parent function. This question is answered by a sequent whose premises contain one more proposition, $\mathcal{CH}(A)$, in addition to all previously available premises. Translating this into the language of $\mathcal{CH}$-propositions, we find that we will prove the sequent:

$$\Gamma \vdash \mathcal{CH}(A \to B) \quad = \quad \Gamma \vdash \mathcal{CH}(A) \Rightarrow \mathcal{CH}(B) \quad = \quad \Gamma \vdash \alpha \Rightarrow \beta$$

if we can prove the sequent $\Gamma, \mathcal{CH}(A) \vdash \mathcal{CH}(B) = \Gamma, \alpha \vdash \beta$. In the notation of formal logic, this is a **derivation rule** (rather than an axiom) and is written as:

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \Rightarrow \beta} \quad \text{(create function)} \qquad .$$

The **turnstile** symbol ($\vdash$) groups weaker than other operators. So, we can write sequents such as $(\Gamma, \alpha) \vdash (\beta \Rightarrow \gamma)$ with fewer parentheses: $\Gamma, \alpha \vdash \beta \Rightarrow \gamma$.

What code corresponds to the "create function" rule? The proof of $\Gamma \vdash \alpha \Rightarrow \beta$ depends on a proof of another sequent. So, the corresponding code must be a *function* that takes a proof of the previous sequent as an argument and returns a proof of the new sequent. By the CH correspondence, a proof of a sequent corresponds to a code expression of the type given by the goal of the sequent. That

expression may use arguments of types corresponding to the premises of the sequent. So, a proof of the sequent $\Gamma, \alpha \vdash \beta$ is an expression `exprB` of type B that may use a given value of type A as well as any other arguments given previously. Then we can write the proof code for the sequent $\Gamma \vdash \alpha \Rightarrow \beta$ as the nameless function `(x:A) => exprB`. This function has type $A \to B$ and requires us to already have a suitable expression `exprB`. This exactly corresponds to the proof rule "create function". We may write the corresponding code as:

$$\text{Proof}\left(\Gamma \vdash \mathcal{CH}(A) \Rightarrow \mathcal{CH}(B)\right) = x^{:A} \to \text{Proof}\left(\Gamma, x^{:A} \vdash \mathcal{CH}(B)\right) \quad .$$

Here we wrote $x^{:A}$ instead of $\mathcal{CH}(A)$ since the value $x^{:A}$ is a proof of the proposition $\mathcal{CH}(A)$. We will see in Section 5.2.3 how premises such as $\Gamma, x^{:A}$ are implemented in code.

4) Use a function At any place in the code, we may apply an already defined function of type $A \to B$ to an already computed value of type A. The result will be a value of type B. This corresponds to assuming $\mathcal{CH}(A \to B)$ and $\mathcal{CH}(A)$, and then deriving $\mathcal{CH}(B)$. The notation for this proof rule is:

$$\frac{\Gamma \vdash \alpha \qquad \Gamma \vdash \alpha \Rightarrow \beta}{\Gamma \vdash \beta} \quad \text{(use function)} \quad .$$

The code corresponding to this proof rule takes previously computed values `x:A` and `f:A => B`, and writes the expression `f(x)`. This can be written as a function application:

$$\text{Proof}\,(\Gamma \vdash \beta) = \text{Proof}\,(\Gamma \vdash \alpha \Rightarrow \beta)\,(\text{Proof}\,(\Gamma \vdash \alpha)) \quad .$$

5) Create a tuple If we have already computed some values `a:A` and `b:B`, we may write the expression `(a, b)` and so compute a value of the tuple type `(A, B)`. The proof rule is:

$$\frac{\Gamma \vdash \alpha \qquad \Gamma \vdash \beta}{\Gamma \vdash \alpha \wedge \beta} \quad \text{(create tuple)} \quad .$$

Writing $a \times b$ to mean the pair `(a, b)`, we can write the corresponding code expression as:

$$\text{Proof}\,(\Gamma \vdash \alpha \wedge \beta) = \text{Proof}\,(\Gamma \vdash \alpha) \times \text{Proof}\,(\Gamma \vdash \beta) \quad .$$

This rule describes creating a tuple of 2 values. A larger tuple, such as `(w, x, y, z)`, can be expressed via nested pairs, e.g., as `(w, (x, (y, z)))`. So, it suffices to have a derivation rule for creating pairs. That rule allows us to express the rules for creating all larger tuples, so we do not need to define separate rules for, say, $\Gamma \vdash \alpha \wedge \beta \wedge \gamma$.

6) Use a tuple If we already have a value `t:(A,B)` of a tuple type $A \times B$, we can extract one of the parts of the tuple and obtain a value of type `A` or a value of type `B`. The code is `t._1` and `t._2` respectively, and the corresponding sequent proof rules are:

$$\frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \alpha} \quad \text{(use tuple-1)} \qquad\qquad \frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \beta} \quad \text{(use tuple-2)} \quad .$$

The code can be written as:

$$\text{Proof}\,(\Gamma \vdash \alpha) = \pi_1\,(\text{Proof}\,(\Gamma \vdash \alpha \wedge \beta)) \quad , \qquad \text{Proof}\,(\Gamma \vdash \beta) = \pi_2\,(\text{Proof}\,(\Gamma \vdash \alpha \wedge \beta)) \quad ,$$

where we introduced the notation π_1 and π_2 to mean the Scala code `_._1` and `_._2`.

Since all tuples can be expressed through pairs, it is sufficient to have proof rules for pairs.

7) Create a disjunctive value The type `Either[A, B]` corresponding to the disjunction $\alpha \vee \beta$ can be used to define any other disjunctive type; e.g., a disjunctive type with three parts can be expressed as `Either[A, Either[B, C]]`. So, it suffices to have proof rules for a disjunction of *two* propositions.

There are two ways of creating a value of the type `Either[A, B]`: the code expressions are `Left(x:A)` and `Right(y:B)`. The values `x:A` or `y:B` must have been computed previously (and correspond to previously proved sequents). So, the sequent proof rules are:

$$\frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \vee \beta} \quad \text{(create Left)} \qquad\qquad \frac{\Gamma \vdash \beta}{\Gamma \vdash \alpha \vee \beta} \quad \text{(create Right)} \quad .$$

The corresponding code can be written using the case class names `Left` and `Right` as:

$$\text{Proof}\,(\Gamma \vdash \alpha \vee \beta) = \text{Left}\,(\text{Proof}\,(\Gamma \vdash \alpha)) \quad , \qquad \text{Proof}\,(\Gamma \vdash \alpha \vee \beta) = \text{Right}\,(\text{Proof}\,(\Gamma \vdash \beta)) \quad .$$

axioms :

$$\frac{}{\Gamma \vdash \mathcal{CH}(\mathbb{1})} \text{ (use unit)} \qquad \frac{}{\Gamma, \alpha \vdash \alpha} \text{ (use arg)}$$

derivation rules :

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \Rightarrow \beta} \text{ (create function)}$$

$$\frac{\Gamma \vdash \alpha \qquad \Gamma \vdash \alpha \Rightarrow \beta}{\Gamma \vdash \beta} \text{ (use function)}$$

$$\frac{\Gamma \vdash \alpha \qquad \Gamma \vdash \beta}{\Gamma \vdash \alpha \wedge \beta} \text{ (create tuple)}$$

$$\frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \alpha} \text{ (use tuple-1)} \qquad \frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \beta} \text{ (use tuple-2)}$$

$$\frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \vee \beta} \text{ (create } \texttt{Left}\text{)} \qquad \frac{\Gamma \vdash \beta}{\Gamma \vdash \alpha \vee \beta} \text{ (create } \texttt{Right}\text{)}$$

$$\frac{\Gamma \vdash \alpha \vee \beta \qquad \Gamma, \alpha \vdash \gamma \qquad \Gamma, \beta \vdash \gamma}{\Gamma \vdash \gamma} \text{ (use } \texttt{Either}\text{)}$$

Table 5.2: Proof rules for the constructive logic.

8) Use a disjunctive value Pattern matching is the basic way of using a value of type `Either[A, B]`:

```
val result: C = (e: Either[A, B]) match {
  case Left(x:A)    => expr1(x)
  case Right(y:B)   => expr2(y)
}
```

Here, `expr1(x)` is an expression of type `C` computed using `x:A` and any previously computed values. Similarly, `expr2(y)` is computed using `y:B` and previous values. The values used in computation correspond to the premises of a sequent. So, `expr1(x)` represents a proof of a sequent with an additional premise of type `A`. Denoting $\gamma \stackrel{\text{def}}{=} \mathcal{CH}(C)$, we write that sequent as: $\Gamma, \alpha \vdash \gamma$. Similarly, `expr2(y)` is a proof of the sequent $\Gamma, \beta \vdash \gamma$. We can compute `result` only if we can compute `e`, `expr1`, and `expr2`. So, the proof rule is:

$$\frac{\Gamma \vdash \alpha \vee \beta \qquad \Gamma, \alpha \vdash \gamma \qquad \Gamma, \beta \vdash \gamma}{\Gamma \vdash \gamma} \text{ (use Either)} \quad .$$

The corresponding code can be written as:

$$\text{Proof}\,(\Gamma \vdash \gamma) = \text{Proof}\,(\Gamma \vdash \alpha \vee \beta) \;\text{ match } \begin{cases} \text{case } a^{:A} \to & \text{Proof}\,(\Gamma, \alpha \vdash \gamma)_{\text{given } a} \\ \text{case } b^{:B} \to & \text{Proof}\,(\Gamma, \beta \vdash \gamma)_{\text{given } b} \end{cases} \quad .$$

We found eight proof rules shown in Table 5.2. These rules define the **intuitionistic propositional logic**, also called **constructive propositional logic**. We will call this logic "constructive" for short.

5.2.3 Example: Proving a $\mathcal{CH}$-proposition and deriving code from type

We will implement a fully parametric function:

```
def f[A, B]: ((A => A) => B) => B = ???
```

Implementing this function is the same as being able to compute a value of type F, where F is defined as:

$$F \stackrel{\text{def}}{=} \forall(A, B).\,((A \to A) \to B) \to B \quad .$$

Since the type parameters A and B are arbitrary, the body of the fully parametric function `f` cannot use any previously defined values of types A or B. So, the task is formulated as computing a value of type F with *no* previously defined values. This is written as the sequent $\Gamma \vdash \mathcal{CH}(F)$, where the set Γ of premises is empty ($\Gamma = \emptyset$). Rewriting this sequent using the rules of Table 5.1, we get:

$$\forall(\alpha, \beta).\,\emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta \quad , \tag{5.6}$$

where we denoted $\alpha \stackrel{\text{def}}{=} \mathcal{CH}(A)$ and $\beta \stackrel{\text{def}}{=} \mathcal{CH}(B)$.

Figure 5.1: Proof tree for sequent (5.6).

The next step is to prove the sequent (5.6) using the proof rules of Table 5.2. For brevity, we will omit the quantifier $\forall(\alpha, \beta)$ since it will be present in front of every sequent.

Begin by looking for a proof rule whose "denominator" has a sequent similar to Eq. (5.6), i.e., has an implication $(p \Rightarrow q)$ in the goal. We have only one rule that can prove a sequent of the form $\Gamma \vdash (p \Rightarrow q)$; this is the rule "create function". That rule requires us to already have a proof of the sequent $(\Gamma, p) \vdash q$. So, we use this rule with $\Gamma = \emptyset$, and we set $p \stackrel{\text{def}}{=} (\alpha \Rightarrow \alpha) \Rightarrow \beta$ and $q \stackrel{\text{def}}{=} \beta$:

$$\frac{(\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta}{\emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta} \quad .$$

We now need to prove the sequent $(\alpha \Rightarrow \alpha) \Rightarrow \beta) \vdash \beta$, which we can write as $\Gamma_1 \vdash \beta$ where $\Gamma_1 \stackrel{\text{def}}{=} [(\alpha \Rightarrow \alpha) \Rightarrow \beta]$ denotes the set containing the single premise $(\alpha \Rightarrow \alpha) \Rightarrow \beta$.

There are no proof rules that derive a sequent with a premise in the form of an implication $p \Rightarrow q$. However, we have a rule called "use function" that derives a sequent by assuming another sequent containing an implication. After some trial and error, one finds that the rule may be used like this:

$$\frac{\Gamma_1 \vdash \alpha \Rightarrow \alpha \qquad \Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta}{\Gamma_1 \vdash \beta} \quad .$$

Now we need to prove the two new sequents $\Gamma_1 \vdash \alpha \Rightarrow \alpha$ and $\Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta$. To prove these sequents, note that the rule "create function" applies to $\Gamma_1 \vdash \alpha \Rightarrow \alpha$ like this:

$$\frac{\Gamma_1, \alpha \vdash \alpha}{\Gamma_1 \vdash \alpha \Rightarrow \alpha} \quad .$$

The sequent $\Gamma_1, \alpha \vdash \alpha$ is proved directly by the axiom "use arg". The sequent $\Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta$ is also proved by the axiom "use arg" because Γ_1 already contains $(\alpha \Rightarrow \alpha) \Rightarrow \beta$.

The proof of the sequent (5.6) is now complete and can be drawn as a tree (Figure 5.1). The next step is to convert that proof to Scala code.

To do that, we combine the code expressions that correspond to each of the proof rules we used. We need to retrace the proof backwards, starting from the leaves of the tree and going towards the root. We will then assemble the Proof (...) code expressions one after another.

Begin with the left-most leaf: "use arg". That rule gives the code $x^{:A}$:

$$\text{Proof}\,(\Gamma_1, \alpha \vdash \alpha)_{\text{given } x} = x^{:A} \quad .$$

Here "given x" means that $x^{:A}$ must be a proof of the premise α in the sequent $\Gamma_1, \alpha \vdash \alpha$ (recall that α denotes $\mathcal{CH}(A)$ and x has type A). So, we need to use the same $x^{:A}$ when we write the code for the previous rule, "create function":

$$\text{Proof}\,(\Gamma_1 \vdash \alpha \Rightarrow \alpha) = (x^{:A} \to \text{Proof}\,(\Gamma_1, \alpha \vdash \alpha)_{\text{given } x}) = (x^{:A} \to x) \quad .$$

The right-most leaf "use arg" corresponds to the code $f^{:(A\to A)\to B}$, where f is the premise contained in Γ_1. So we can write:

$$\text{Proof}\,(\Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta)_{\text{given } f} = f^{:(A\to A)\to B} \quad .$$

The previous rule, "use function", combines the two preceding proofs:

$$\begin{aligned}
&\text{Proof}\,((\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta)_{\text{given } f^{:(A\to A)\to B}} \\
&= \text{Proof}\,(\Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta)\,(\text{Proof}\,(\Gamma_1 \vdash \alpha \Rightarrow \alpha))_{\text{given } f} \\
&= f(x^{:A} \to x) \quad .
\end{aligned}$$

Going further backwards, we find that the rule applied before "use function" was "create function". We need to provide the same $f^{:(A\to A)\to B}$ as in the premise above, and so we obtain the code:

$$\begin{aligned}
&\text{Proof}\,(\emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta) \\
&= f^{:(A\to A)\to B} \to \text{Proof}\,((\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta)_{\text{given } f^{:(A\to A)\to B}} \\
&= f^{:(A\to A)\to B} \to f(x^{:A} \to x) \quad .
\end{aligned}$$

This is the final code expression that implements the type $((A \to A) \to B) \to B$. In this way, we have systematically derived the code from the type signature of a function. That function can be written in Scala as:

```
def f[A, B]: ((A => A) => B) => B = { f => f(x => x) }
```

We found the proof tree in Figure 5.1 by guessing how to combine various proof rules. In particular, we had to guess how to apply the "use function" rule. So, if we *somehow* find a proof tree for a sequent, we can prove the sequent and derive the corresponding code. However, it is not always obvious how to apply the proof rules, and we sometimes have to guess or to try different possibilities. In other words, the rules in Table 5.7 do not provide an algorithm for finding a proof tree automatically. It turns out that one can replace the rules in Table 5.7 by a different but equivalent set of derivations rules that do give an algorithm called the "LJT algorithm" in Section C below. That algorithm first translates a type signature containing tuples, disjunctive types, and function types into a logical formula. Then the algorithm either finds that the given formula cannot be proved, or it finds a proof and infers code that has the given type signature.

The library `curryhoward`[2] implements the LJT algorithm. Here are some examples of using this library for "code inference". We will run the `ammonite`[3] shell to load the library more easily.

As a non-trivial (but artificial) example, consider the type signature:

$$\forall(A, B).\ ((((A \to B) \to A) \to A) \to B) \to B \quad .$$

It is not obvious whether a function with this type signature exists. The LJT algorithm can figure that out and derive the code automatically. The library does this via the method `implement`:

```
@ import $ivy.`io.chymyst::curryhoward:0.3.8`, io.chymyst.ch._

@ def f[A, B]: ((((A => B) => A) => A) => B) => B   =   implement
defined function f

@ println(f.lambdaTerm.prettyPrint)
a => a (b => b (c => a (d => c)))
```

[2]See https://github.com/Chymyst/curryhoward
[3]http://ammonite.io/#Ammonite-Shell

The code $a \to a\,(b \to b\,(c \to a\,(d \to c)))$ was produced automatically for the function `f`. The function `f` has been compiled and is ready to be used in any subsequent code.

A compile-time error occurs when no fully parametric function has the given type signature:

```
@ def g[A, B]: ((A => B) => A) => A   =   implement
cmd3.sc:1: type ((A => B) => A) => A cannot be implemented
```

The logical formula corresponding to this type signature is:

$$\forall(\alpha,\beta).\ ((\alpha \Rightarrow \beta) \Rightarrow \alpha) \Rightarrow \alpha \quad . \tag{5.7}$$

This formula is known as "Peirce's law"[4] and gives an example showing that the logic of types in functional programming languages is not Boolean (other examples are shown in Sections 5.2.4 and 5.5.4). Peirce's law is true in Boolean logic but does not hold in the constructive logic, i.e., it cannot be derived using the proof rules in Table 5.7. If we try to implement `g[A, B]` with the type signature shown above via fully parametric code, we will fail to write code that compiles without type errors. This is because no such code exists, — not because we are insufficiently clever. The LJT algorithm can *prove* that the given type signature cannot be implemented. The `curryhoward` library will then print an error message, and compilation will fail.

As another example, let us verify that the type signature from Section 5.2.4 is not implementable:

```
@ def bad[A, B, C](g: A => Either[B, C]): Either[A => B, A => C]   =   implement
cmd4.sc:1: type (A => Either[B, C]) => Either[A => B, A => C] cannot be implemented
```

The LJT algorithm will sometimes find several inequivalent proofs of the same logic formula. In that case, each of the different proofs will be automatically translated into code. The `curryhoward` library tries to select the code that has the least information loss, according to several heuristics. In many cases, the heuristics select the implementation that is most useful to the programmer.

The rules of constructive logic and the LJT algorithm define rigorously what it means to write code "guided by the types". However, in order to use the LJT algorithm well, a programmer needs to learn how to infer code from types by hand. We will practice doing that throughout the book.

5.2.4 Failure of Boolean logic in reasoning about $\mathcal{CH}$-propositions

Programmers are most familiar with the Boolean logic whose operations are written in Scala as `x && y` (conjunction), `x || y` (disjunction), and `!x` (negation). However, it turns out that the Boolean logic does *not* always produce correct conclusions when reasoning about $\mathcal{CH}$-propositions. One needs to use the constructive logic to reason correctly about implementable type signatures.

Let us nevertheless briefly look at how Boolean logic would handle that reasoning. In the Boolean logic, each proposition ($\alpha, \beta, ...$) is either $True$ or $False$. The operations are $\alpha \wedge \beta$ (conjunction), $\alpha \vee \beta$ (disjunction), and $\neg\alpha$ (negation). The **implication** ($\Rightarrow$) is defined through other operations by:

$$(\alpha \Rightarrow \beta) \stackrel{\text{def}}{=} ((\neg\alpha) \vee \beta) \quad . \tag{5.8}$$

To verify whether a formula is true in the Boolean logic, we can substitute either $True$ or $False$ into every variable and check if the formula has the value $True$ in all possible cases. The result can be arranged into a truth table. The formula is true if all values in its truth table are $True$.

Disjunction, conjunction, negation, and implication operations have the following truth table:

α	β	$\alpha \vee \beta$	$\alpha \wedge \beta$	$\neg\alpha$	$\alpha \Rightarrow \beta$
$True$	$True$	$True$	$True$	$False$	$True$
$True$	$False$	$True$	$False$	$False$	$False$
$False$	$True$	$True$	$False$	$True$	$True$
$False$	$False$	$False$	$False$	$True$	$True$

[4] https://en.wikipedia.org/wiki/Peirce%27s_law

Using this table, we find that the formula $\alpha \Rightarrow \alpha$ has the value $True$ in all cases, whether α itself is $True$ or $False$. This check is sufficient to show that $\forall\alpha.\,\alpha \Rightarrow \alpha$ is true in Boolean logic.

Here is the truth table for the formulas $\forall(\alpha,\beta).\,(\alpha \wedge \beta) \Rightarrow \alpha$ and $\forall(\alpha,\beta).\,\alpha \Rightarrow (\alpha \wedge \beta)$. The first formula is true in Boolean logic since all values in its column are $True$, while the second formula is not true since one value in the last column is $False$:

α	β	$\alpha \wedge \beta$	$(\alpha \wedge \beta) \Rightarrow \alpha$	$\alpha \Rightarrow (\alpha \wedge \beta)$
$True$	$True$	$True$	$True$	$True$
$True$	$False$	$False$	$True$	$False$
$False$	$True$	$False$	$True$	$True$
$False$	$False$	$False$	$True$	$True$

Table 5.3 shows more examples of logical formulas that are true in Boolean logic. Each formula is first written in terms of $\mathcal{CH}$-propositions (we denote $\alpha \stackrel{\text{def}}{=} \mathcal{CH}(A)$ and $\beta \stackrel{\text{def}}{=} \mathcal{CH}(B)$ for brevity) and then as a Scala type signature of a function. So, all these type signatures *can* be implemented.

Logic formula	**Type formula**	**Scala code**
$\forall\alpha.\,\alpha \Rightarrow \alpha$	$\forall A.\,A \to A$	`def id[A](x: A): A = x`
$\forall\alpha.\,\alpha \Rightarrow True$	$\forall A.\,A \to \mathbb{1}$	`def toUnit[A](x: A): Unit = ()`
$\forall(\alpha,\beta).\,\alpha \Rightarrow (\alpha \vee \beta)$	$\forall(A,B).\,A \to A+B$	`def toL[A, B](x: A): Either[A, B] = Left(x)`
$\forall(\alpha,\beta).\,(\alpha \wedge \beta) \Rightarrow \alpha$	$\forall(A,B).\,A \times B \to A$	`def first[A, B](p: (A, B)): A = p._1`
$\forall(\alpha,\beta).\,\alpha \Rightarrow (\beta \Rightarrow \alpha)$	$\forall(A,B).\,A \to (B \to A)$	`def const[A, B](x: A): B => A = (_ => x)`

Table 5.3: Examples of logical formulas that are true theorems in Boolean logic.

Table 5.4 some examples of formulas that are *not true* in Boolean logic. Translated into type formulas and then into Scala, these formulas yield type signatures that *cannot* be implemented by fully parametric functions.

Logic formula	**Type formula**	**Scala type signature**
$\forall\alpha.True \Rightarrow \alpha$	$\forall A.\,\mathbb{1} \to A$	`def f[A](x: Unit): A`
$\forall(\alpha,\beta).\,(\alpha \vee \beta) \Rightarrow \alpha$	$\forall(A,B).\,A+B \to A$	`def f[A,B](x: Either[A, B]): A`
$\forall(\alpha,\beta).\,\alpha \Rightarrow (\alpha \wedge \beta)$	$\forall(A,B).\,A \to A \times B$	`def f[A,B](p: A): (A, B)`
$\forall(\alpha,\beta).\,(\alpha \Rightarrow \beta) \Rightarrow \alpha$	$\forall(A,B).\,(A \to B) \to A$	`def f[A,B](x: A => B): A`

Table 5.4: Examples of logical formulas that are *not* true in Boolean logic.

At first sight, it may appear from these examples that whenever a logical formula is true in Boolean logic, the corresponding type signature can be implemented in code, and vice versa. However, this is *incorrect*: the rules of Boolean logic are not fully suitable for reasoning about types in a functional language. We will now show some examples of formulas that are true in Boolean logic but correspond to unimplementable type signatures.

The first example is given by the following type:

$$\forall(A,B,C).\,(A \to B+C) \to (A \to B)+(A \to C) \quad , \tag{5.9}$$

which corresponds to the Scala type signature:

```
def bad[A, B, C](g: A => Either[B, C]): Either[A => B, A => C] = ???
```

The function `bad` *cannot* be implemented via fully parametric code. To see why, consider that the only available data is a function $g^{:A\to B+C}$, which returns values of type B or C depending (in some unknown way) on the input value of type A. The function `bad` must return either a function of type $A \to B$ or a function of type $A \to C$. How can the code of `bad` make that decision? The only input data is the function g that takes an argument of type A. We could imagine applying g to various arguments of type A and to see whether g returns a B or a C. However, the type A is arbitrary, and a fully parametric function cannot produce a value of type A in order to apply g to it. So, the decision about whether to return $A \to B$ or $A \to C$ must be independent of g. That decision must be hard-coded in the function `bad`.

Suppose we hard-coded the decision to return a function of type $A \to B$. How would we create a function of type $A \to B$ in the body of `bad`? Given a value $x^{:A}$ of type A, we would need to compute some value of type B. Since the type B is arbitrary (it is a type parameter), we cannot produce a value of type B from scratch. The only potential source of values of type B is the given function g. The only way of using g is to apply it to $x^{:A}$. However, for some x, the value $g(x)$ may be of the form `Right(c)`, where `c` is of type C. In that case, we will have a value of type C, not B. So, in general, we cannot guarantee that we can always obtain a value of type B from a given value $x^{:A}$. This means we cannot build a function of type $A \to B$ out of the function g. Similarly, we cannot build a function of type $A \to C$ out of g.

Whether we decide to return $A \to B$ or $A \to C$, we will not be able to return a value of the required type, as we just saw. We must conclude that we cannot implement `bad` as a fully parametric function.

We could try to switch between $A \to B$ and $A \to C$ depending on a given value of type A. This idea, however, means that we are working with a different type signature:

$$\forall(A, B, C).\ (A \to B + C) \to A \to (A \to B) + (A \to C) \quad .$$

This type signature *can* be implemented, for instance, by this Scala code:

```
def q[A, B, C](g: A => Either[B, C]): A => Either[A => B, A => C] = { a =>
  g(a) match {
    case Left(b) => Left(_ => b)
    case Right(c) => Right(_ => c)
  }
}
```

But this is not the required type signature (5.9).

Now let us convert the type signature (5.9) into a $\mathcal{CH}$-proposition:

$$\forall(\alpha, \beta, \gamma).\ (\alpha \Rightarrow (\beta \vee \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \vee (\alpha \Rightarrow \gamma)) \quad , \tag{5.10}$$

$$\text{where we denoted} \quad \alpha \stackrel{\text{def}}{=} \mathcal{CH}(A), \quad \beta \stackrel{\text{def}}{=} \mathcal{CH}(B), \quad \gamma \stackrel{\text{def}}{=} \mathcal{CH}(C) \quad .$$

It turns out that this formula is true in Boolean logic. To prove this, we need to show that Eq. (5.10) is equal to $True$ for any Boolean values of the variables α, β, γ. One way is to rewrite the expression (5.10) using the rules of Boolean logic, such as Eq. (5.8):

$$\begin{aligned}
& \underline{\alpha \Rightarrow} (\beta \vee \gamma) \\
\text{definition of } \Rightarrow \text{ via Eq. (5.8)}: \quad & = (\neg\alpha) \vee \beta \vee \gamma \quad , \\
& \underline{(\alpha \Rightarrow \beta)} \vee \underline{(\alpha \Rightarrow \gamma)} \\
\text{definition of } \Rightarrow \text{ via Eq. (5.8)}: \quad & = \underline{(\neg\alpha)} \vee \beta \vee \underline{(\neg\alpha)} \vee \gamma \\
\text{property } x \vee x = x \text{ in Boolean logic}: \quad & = (\neg\alpha) \vee \beta \vee \gamma \quad ,
\end{aligned}$$

showing that $\alpha \Rightarrow (\beta \vee \gamma)$ is in fact *equal* to $(\alpha \Rightarrow \beta) \vee (\alpha \Rightarrow \gamma)$ in Boolean logic.

Let us also give a proof via truth-value reasoning. The only possibility for an implication $X \Rightarrow Y$ to be $False$ is when $X = True$ and $Y = False$. So, Eq. (5.10) can be $False$ only if $(\alpha \Rightarrow (\beta \vee \gamma)) = True$

and $(\alpha \Rightarrow \beta) \vee (\alpha \Rightarrow \gamma) = False$. A disjunction can be false only when both parts are false; so we must have both $(\alpha \Rightarrow \beta) = False$ and $(\alpha \Rightarrow \gamma) = False$. This is only possible if $\alpha = True$ and $\beta = \gamma = False$. But, with these value assignments, we find $(\alpha \Rightarrow (\beta \vee \gamma)) = False$ rather than $True$ as we assumed. It follows that we cannot ever make Eq. (5.10) equal to $False$. So, Eq. (5.10) is true in Boolean logic.

5.3 Equivalence of types

We found a correspondence between types, code, logical propositions, and proofs, which is known as the **Curry-Howard correspondence**. An example of the CH correspondence is that a proof of the logical proposition:

$$\forall(\alpha, \beta).\, \alpha \Rightarrow (\beta \Rightarrow \alpha) \tag{5.11}$$

corresponds to the code of the following function:

```
def f[A, B]: A => (B => A) = { x => _ => x }
```

With the CH correspondence in mind, we may say that the *existence* of the code `x => _ => x` with the type $A \to (B \to A)$ "is" a proof of the logical formula (5.11), because it shows how to compute a value of type $\forall(A, B).\, A \to B \to A$.

The Curry-Howard correspondence maps logic formulas such as $(\alpha \vee \beta) \wedge \gamma$ into type expressions such as $(A + B) \times C$. We have seen that types behave similarly to logic formulas in one respect: A logic formula is a true theorem of constructive logic when the corresponding type signature can be implemented as a fully parametric function, and vice versa.

It turns out that the similarity ends here. In other respects, type expressions behave as *arithmetic* expressions and not as logic formulas. For this reason, the type notation used in this book denotes disjunctive types by $A + B$ and tuples by $A \times B$, which is designed to remind us of arithmetic expressions (such as $1 + 2$ and 2×3) rather than of logical formulas (such as $A \vee B$ and $A \wedge B$).

An important use of the type notation is for writing equations with types. Can we use the arithmetic intuition for writing type equations such as:

$$(A + B) \times C = A \times C + B \times C \quad ? \tag{5.12}$$

In this section, we will learn how to check whether one type expression is equivalent to another.

5.3.1 Logical identity does not correspond to type equivalence

The CH correspondence maps Eq. (5.12) into the logic formula:

$$\forall(A, B, C).\ (A \vee B) \wedge C = (A \wedge C) \vee (B \wedge C) \quad . \tag{5.13}$$

This formula is the well-known "distributive law"[5] valid in Boolean logic as well as in the constructive logic. Since a logical equation $P = Q$ means $P \Rightarrow Q$ and $Q \Rightarrow P$, the distributive law (5.13) means that the two formulas hold:

$$\forall(A, B, C).\ (A \vee B) \wedge C \Rightarrow (A \wedge C) \vee (B \wedge C) \quad , \tag{5.14}$$

$$\forall(A, B, C).\ (A \wedge C) \vee (B \wedge C) \Rightarrow (A \vee B) \wedge C \quad . \tag{5.15}$$

The CH correspondence maps these logical formulas to fully parametric functions with types:

```
def f1[A, B, C]: ((Either[A, B], C))    => Either[(A, C), (B, C)] = ???
def f2[A, B, C]: Either[(A, C), (B, C)] => (Either[A, B], C)      = ???
```

[5] See https://en.wikipedia.org/wiki/Distributive_property#Rule_of_replacement

In the type notation, these type signatures are written as:

$$f_1^{A,B,C} : (A+B)\times C \to A\times C + B\times C \quad ,$$
$$f_2^{A,B,C} : A\times C + B\times C \to (A+B)\times C \quad .$$

Since the two logical formulas (5.14)–(5.15) are true theorems in constructive logic, we expect to be able to implement the functions `f1` and `f2`. It is not straightforward to guess how to combine the proof rules of Table 5.7 to obtain proofs of Eqs. (5.14)–(5.15). So, instead of deriving the implementations of `f1` and `f2` from the CH correspondence, we will write the Scala code directly.

To implement `f1`, we need to perform pattern matching on the argument:

```
def f1[A, B, C]: ((Either[A, B], C)) => Either[(A, C), (B, C)] = {
  case (Left(a), c)   => Left((a, c))  // No other choice here.
  case (Right(b), c)  => Right((b, c)) // No other choice here.
}
```

In both cases, we have only one possible expression of the correct type.

Similarly, the implementation of `f2` leaves us no choices:

```
def f2[A, B, C]: Either[(A, C), (B, C)] => (Either[A, B], C) = {
  case Left((a, c))   => (Left(a), c)  // No other choice here.
  case Right((b, c))  => (Right(b), c) // No other choice here.
}
```

The code of `f1` and `f2` never discards any given values; in other words, these functions appear to preserve information. We can formulate this property rigorously as a requirement that an arbitrary value `x: (Either[A, B], C)` be mapped by `f1` to some value `y: Either[(A, C), (B, C)]` and then mapped by `f2` back to *the same* value `x`. Similarly, any value `y` of type `Either[(A, C), (B, C)]` should be transformed by `f2` and then by `f1` back to the same value `y`.

Let us write these conditions as equations:

$$\forall x^{:(A+B)\times C}.\, f_2(f_1(x)) = x \quad , \qquad \forall y^{:A\times C+B\times C}.\, f_1(f_2(y)) = y \quad .$$

If these equations hold, it means that all the information in a value $x^{:(A+B)\times C}$ is completely preserved inside the value $y \stackrel{\text{def}}{=} f_1(x)$; the original value x can be recovered as $x = f_2(y)$. Then the function f_1 is the **inverse** of f_2. Conversely, all the information in a value $y^{:A\times C+B\times C}$ is preserved inside $x \stackrel{\text{def}}{=} f_2(y)$ and can be recovered by applying f_1. Since the values $x^{:(A+B)\times C}$ and $y^{:A\times C+B\times C}$ are arbitrary, it will follow that the *data types* themselves, $(A+B)\times C$ and $A\times C + B\times C$, carry equivalent information. Such types are called equivalent or isomorphic.

Generally, we say that types P and Q are **equivalent** or **isomorphic** (denoted $P \cong Q$) when there exist functions $f_1 : P \to Q$ and $f_2 : Q \to P$ that are inverses of each other. We can write these conditions using the notation $(f_1 ⨾ f_2)(x) \stackrel{\text{def}}{=} f_2(f_1(x))$ as:

$$f_1 ⨾ f_2 = \text{id} \quad , \qquad f_2 ⨾ f_1 = \text{id} \quad .$$

(In Scala, the forward composition $f_1 ⨾ f_2$ is the function `f1 andThen f2`. We omit type annotations since we already checked that the types match.) If these conditions hold, there is a one-to-one correspondence between values of types P and Q. This is the same as to say that the data types P and Q "carry equivalent information".

To verify that the Scala functions `f1` and `f2` defined above are inverses of each other, we first check if $f_1 ⨾ f_2 = \text{id}$. Applying $f_1 ⨾ f_2$ means to apply f_1 and then to apply f_2 to the result. Begin by applying f_1 to an arbitrary value $x^{:(A+B)\times C}$. A value x of that type can be in only one of the two disjoint cases: a tuple `(Left(a), c)` or a tuple `(Right(b), c)`, for some values `a:A`, `b:B`, and `c:C`. The Scala code of `f1` maps these tuples to `Left((a, c))` and to `Right((b, c))` respectively; we can see this directly from the code of `f1`. We then apply f_2 to those values, which maps them back to a tuple `(Left(a), c)` or to a tuple `(Right(b), c)` respectively, according to the code of `f2`. These tuples are exactly the value x we started with. So, applying $f_1 ⨾ f_2$ to an arbitrary $x^{:(A+B)\times C}$ returns that value x. This is the same as to say that $f_1 ⨾ f_2 = \text{id}$.

To check whether $f_2 ; f_1 = \text{id}$, we apply f_2 to an arbitrary value $y^{:A\times C+B\times C}$, which must be one of the two disjoint cases, `Left((a, c))` or `Right((b, c))`. The code of `f2` maps these two cases into tuples `(Left(a), c)` and `(Right(b), c)` respectively. Then we apply `f1` and map these tuples back to `Left((a, c))` and `Right((b, c))` respectively. It follows that applying f_2 and then f_1 will always return the initial value. As a formula, this is written as $f_2 ; f_1 = \text{id}$.

By looking at the code of `f1` and `f2`, we can directly observe that these functions are inverses of each other: the tuple pattern `(Left(a), c)` is mapped to `Left((a, c))`, and the pattern `(Right(b), c)` to `Right((b, c))`, or vice versa. It is visually clear that no information is lost and that the original values are returned by function compositions $f_1 ; f_2$ or $f_2 ; f_1$.

We find that the logical identity (5.13) leads to an equivalence of the corresponding types:

$$(A+B)\times C \cong A\times C + B\times C \quad . \tag{5.16}$$

To get Eq. (5.16) from Eq. (5.13), we need to convert a logical formula to an arithmetic expression by replacing the disjunction operations $\vee$ by $+$ and the conjunctions $\wedge$ by $\times$ everywhere.

As another example of a logical identity, consider the associativity law for conjunction:

$$(\alpha\wedge\beta)\wedge\gamma = \alpha\wedge(\beta\wedge\gamma) \quad . \tag{5.17}$$

The corresponding types are $(A\times B)\times C$ and $A\times(B\times C)$; in Scala, `((A, B), C)` and `(A, (B, C))`. We can define functions that convert between these types without information loss:

```
def f3[A, B, C]: (((A, B), C)) => (A, (B, C)) = { case ((a, b), c) => (a, (b, c)) }
def f4[A, B, C]: (A, (B, C)) => (((A, B), C)) = { case (a, (b, c)) => ((a, b), c) }
```

By applying these functions to arbitrary values of types `((A, B), C)` and `(A, (B, C))`, it is easy to see that the functions `f3` and `f4` are inverses of each other. This is also directly visible in the code: the nested tuple pattern `((a, b), c)` is mapped to the pattern `(a, (b, c))` and back. So, the types $(A\times B)\times C$ and $A\times(B\times C)$ are equivalent, and we can write $A\times B\times C$ without parentheses.

Does a logical identity always correspond to an equivalence of types? This turns out to be *not* so. A simple example of a logical identity that does not correspond to a type equivalence is:

$$True\vee\alpha = True \quad . \tag{5.18}$$

Since the CH correspondence maps the logical constant $True$ into the unit type $\mathbb{1}$, the type equivalence corresponding to Eq. (5.18) is $\mathbb{1}+A\cong\mathbb{1}$. The type denoted by $\mathbb{1}+A$ means `Option[A]` in Scala, so the corresponding equivalence is `Option[A]≅Unit`. Intuitively, this type equivalence should not hold: an `Option[A]` may carry a value of type `A`, which cannot possibly be stored in a value of type `Unit`. We can verify this intuition rigorously by proving that any fully parametric functions with type signatures $g_1 : \mathbb{1}+A \to \mathbb{1}$ and $g_2 : \mathbb{1}\to\mathbb{1}+A$ will not satisfy $g_1 ; g_2 = \text{id}$. To verify this, we note that $g_2 : \mathbb{1}\to\mathbb{1}+A$ must have this type signature:

```
def g2[A]: Unit => Option[A] = ???
```

This function must always return `None`, since a fully parametric function cannot produce values of an arbitrary type `A` from scratch. Therefore, $g_1 ; g_2$ is also a function that always returns `None`. The function $g_1 ; g_2$ has type signature $\mathbb{1}+A\to\mathbb{1}+A$ or, in Scala syntax, `Option[A] => Option[A]`, and is not equal to the identity function, because the identity function does *not* always return `None`.

Another example of a logical identity without a type equivalence is the distributive law:

$$\forall(A,B,C).\ (A\wedge B)\vee C = (A\vee C)\wedge(B\vee C) \quad , \tag{5.19}$$

which is "dual" to the law (5.13), i.e., it is obtained from Eq. (5.13) by swapping all conjunctions ($\wedge$) with disjunctions ($\vee$). In logic, a dual formula to an identity is also an identity. The CH correspondence maps Eq. (5.19) into the type equation:

$$\forall(A,B,C).\ (A\times B)+C = (A+C)\times(B+C) \quad . \tag{5.20}$$

However, the types $A\times B+C$ and $(A+C)\times(B+C)$ are *not* equivalent. To see why, look at the possible code of a function $g_3 : (A+C)\times(B+C)\to A\times B+C$:

```
1 def g3[A,B,C]: ((Either[A, C], Either[B, C])) => Either[(A, B), C] = {
2   case (Left(a), Left(b))      => Left((a, b)) // No other choice.
3   case (Left(a), Right(c))     => Right(c)     // No other choice.
4   case (Right(c), Left(b))     => Right(c)     // No other choice.
5   case (Right(c1), Right(c2))  => Right(c1)    // Must discard c1 or c2 here!
6 }   // May return Right(c2) instead of Right(c1) in the last line.
```

In line 5, we have a choice of returning `Right(c1)` or `Right(c2)`. Whichever we choose, we will lose information because we will have discarded one of the given values `c1`, `c2`. After evaluating g_3, we will not be able to restore *both* `c1` and `c2`, no matter what code we write. So, the composition $g_3 ; g_4$ with any g_4 cannot be equal to the identity function. The type equation (5.20) is incorrect.

We find that a logical identity $\mathcal{CH}(P) = \mathcal{CH}(Q)$ guarantees, via the CH correspondence, that we can implement *some* fully parametric functions of types $P \to Q$ and $Q \to P$. However, it is not guaranteed that these functions are inverses of each other, i.e., that the type conversions $P \to Q$ or $Q \to P$ have no information loss. So, the type equivalence $P \cong Q$ does not automatically follow from the logical identity $\mathcal{CH}(P) = \mathcal{CH}(Q)$.

The CH correspondence means that for true propositions $\mathcal{CH}(X)$ we can compute *some* value x of type X. However, the CH correspondence does not guarantee that the computed value $x^{:X}$ will satisfy any additional properties or laws.

5.3.2 Arithmetic identity corresponds to type equivalence

Looking at the examples of equivalent types, we notice that correct type equivalences correspond to *arithmetical* identities rather than *logical* identities. For instance, the logical identity in Eq. (5.13) leads to the type equivalence (5.16), which looks like a standard identity of arithmetic, such as:

$$(1+10)\times 20 = 1\times 20 + 10\times 20 \quad .$$

The logical identity in Eq. (5.19), which does *not* yield a type equivalence, leads to an incorrect arithmetic equation 5.20, e.g., $(1\times 10)+20 \neq (1+20)\times(10+20)$. Similarly, the associativity law (5.17) leads to a type equivalence and to the arithmetic identity:

$$(a\times b)\times c = a\times(b\times c) \quad ,$$

The logical identity in Eq. (5.18), which does not yield a type equivalence, leads to an incorrect arithmetic statement $(\forall a.\, 1+a = 1)$.

Table 5.5 summarizes these and other examples of logical identities and the corresponding type equivalences. In all rows, quantifiers such as $\forall\alpha$ or $\forall(A,B)$ are implied as necessary.

Because we chose the type notation to be similar to the ordinary arithmetic notation, it is easy to translate a possible type equivalence into an arithmetic equation. In all cases, valid arithmetic identities correspond to type equivalences, and failures to obtain a type equivalence correspond to incorrect arithmetic identities. With regard to type equivalence, types such as $A+B$ and $A\times B$ behave similarly to arithmetic expressions such as $10+20$ and 10×20 and not similarly to logical formulas such as $\alpha\vee\beta$ and $\alpha\wedge\beta$.

We already verified the first line and the last three lines of Table 5.5. Other identities are verified in a similar way. Let us begin with lines 3 and 4 of Table 5.5, which involve the proposition $False$ and the corresponding void type $\mathbb{0}$ (Scala's `Nothing`). Reasoning about the void type needs a special technique that we will now develop while verifying the type isomorphisms $\mathbb{0}\times A \cong \mathbb{0}$ and $\mathbb{0}+A \cong A$.

Example 5.3.2.1 Verify the type equivalence $\mathbb{0}\times A \cong \mathbb{0}$.

Solution Recall that the type notation $\mathbb{0}\times A$ represents the Scala tuple type `(Nothing, A)`. To demonstrate that the type `(Nothing, A)` is equivalent to the type `Nothing`, we need to show that the type `(Nothing, A)` has *no* values. Indeed, how could we create a value of type, say, `(Nothing, Int)`? We would need to fill *both* parts of the tuple. We have values of type `Int`, but we can never get a value of type `Nothing`. So, regardless of the type `A`, it is impossible to create any values of type `(Nothing, A)`. In

Logical identity	Type equivalence (if it holds)
$True \vee \alpha = True$	$\mathbb{1} + A \not\cong \mathbb{1}$
$True \wedge \alpha = \alpha$	$\mathbb{1} \times A \cong A$
$False \vee \alpha = \alpha$	$\mathbb{0} + A \cong A$
$False \wedge \alpha = False$	$\mathbb{0} \times A \cong \mathbb{0}$
$\alpha \vee \beta = \beta \vee \alpha$	$A + B \cong B + A$
$\alpha \wedge \beta = \beta \wedge \alpha$	$A \times B \cong B \times A$
$(\alpha \vee \beta) \vee \gamma = \alpha \vee (\beta \vee \gamma)$	$(A + B) + C \cong A + (B + C)$
$(\alpha \wedge \beta) \wedge \gamma = \alpha \wedge (\beta \wedge \gamma)$	$(A \times B) \times C \cong A \times (B \times C)$
$(\alpha \vee \beta) \wedge \gamma = (\alpha \wedge \gamma) \vee (\beta \wedge \gamma)$	$(A + B) \times C \cong A \times C + B \times C$
$(\alpha \wedge \beta) \vee \gamma = (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$	$(A \times B) + C \not\cong (A + C) \times (B + C)$

Table 5.5: Logic identities with disjunction and conjunction, and the possible type equivalences.

other words, the set of values of the type `(Nothing, A)` is empty. But that is the definition of the void type `Nothing`. The types `(Nothing, A)` (denoted by $\mathbb{0} \times A$) and `Nothing` (denoted by $\mathbb{0}$) are both void and therefore equivalent.

Example 5.3.2.2 Verify the type equivalence $\mathbb{0} + A \cong A$.

Solution Recall that the type notation $\mathbb{0} + A$ represents the Scala type `Either[Nothing, A]`. We need to show that any value of that type can be mapped without loss of information to a value of type `A`, and vice versa. This means implementing functions $f_1 : \mathbb{0} + A \to A$ and $f_2 : A \to \mathbb{0} + A$ such that $f_1 ⨾ f_2 = \text{id}$ and $f_2 ⨾ f_1 = \text{id}$.

The argument of f_1 is of type `Either[Nothing, A]`. How can we create a value of that type? Our only choices are to create a `Left(x)` with `x:Nothing`, or to create a `Right(y)` with `y:A`. However, we cannot create a value `x` of type `Nothing` because the type `Nothing` has *no* values. We cannot create a `Left(x)`. The only remaining possibility is to create a `Right(y)` with some value `y` of type `A`. So, any values of type $\mathbb{0} + A$ must be of the form `Right(y)`, and we can extract that `y` to obtain a value of type `A`:

```
def f1[A]: Either[Nothing, A] => A = {
  case Right(y) => y
  // No need for ‘case Left(x) => ...‘ since no ‘x‘ can ever be given as ‘Left(x)‘.
}
```

For the same reason, there is only one implementation of the function `f2`:

```
def f2[A]: A => Either[Nothing, A] = { y => Right(y) }
```

It is clear from the code that the functions `f1` and `f2` are inverses of each other.

We have just seen that a value of type $\mathbb{0} + A$ is always a `Right(y)` with some `y:A`. Similarly, a value of type $A + \mathbb{0}$ is always a `Left(x)` with some `x:A`. So, we will use the notation $A + \mathbb{0}$ and $\mathbb{0} + A$ to *denote* the `Left` and the `Right` parts of the disjunctive type `Either`. This notation agrees with the behavior of the Scala compiler, which will infer the types `Either[A, Nothing]` or `Either[Nothing, A]` for these parts:

```
def toLeft[A, B]: A => Either[A, B] = x => Left(x)
def toRight[A, B]: B => Either[A, B] = y => Right(y)

scala> toLeft(123)
res0: Either[Int, Nothing] = Left(123)

scala> toRight("abc")
res1: Either[Nothing, String] = Right("abc")
```

We can write the functions `toLeft` and `toRight` in a code notation as:

$$\text{toLeft}^{A,B} \stackrel{\text{def}}{=} x^{:A} \to x + \mathbb{0}^{:B} \quad ,$$
$$\text{toRight}^{A,B} \stackrel{\text{def}}{=} y^{:B} \to \mathbb{0}^{:A} + y \quad .$$

In this notation, a value of the disjunctive type is shown without using Scala class

names such as `Either`, `Right`, and `Left`. This shortens the writing and speeds up code reasoning.

The type annotation $\mathbb{0}^{:A}$ is helpful to remind ourselves about the type parameter A used, e.g., by the disjunctive value $\mathbb{0}^{:A} + y^{:B}$ in the body of `toRight[A, B]`. Without this type annotation, $\mathbb{0} + y^{:B}$ means a value of type `Either[A, B]` where the parameter A should be determined by matching the types of other expressions. When it is clear what types are being used, we may omit type annotations and write simply $\mathbb{0} + y$ instead of $\mathbb{0}^{:A} + y^{:B}$.

In the notation $\mathbb{0} + y^{:B}$, we use the symbol $\mathbb{0}$ rather than an ordinary zero (0), to avoid suggesting that 0 is a value of type $\mathbb{0}$. The void type $\mathbb{0}$ has *no* values, unlike the `Unit` type, $\mathbb{1}$, which has a value denoted by 1 in the code notation.

Example 5.3.2.3 Verify the type equivalence $A \times \mathbb{1} \cong A$.

Solution The corresponding Scala types are the tuple `(A, Unit)` and the type `A`. We need to implement functions $f_1 : \forall A. A \times \mathbb{1} \to A$ and $f_2 : \forall A. A \to A \times \mathbb{1}$ and to demonstrate that they are inverses of each other. The Scala code for these functions is:

```
def f1[A]: ((A, Unit)) => A = { case (a, ()) => a }
def f2[A]: A => (A, Unit) = { a => (a, ()) }
```

Let us first write a proof by reasoning directly with Scala code:

```
(f1 andThen f2)((a,())) == f2(f1((a,()))) == f2(a) == (a, ())
(f2 andThen f1)(a) == f1(f2(a)) == f1((a, ())) = a
```

Now let us write a proof in the code notation. The codes of f_1 and f_2 are:

$$f_1 \stackrel{\text{def}}{=} a^{:A} \times 1 \to a \quad , \qquad f_2 \stackrel{\text{def}}{=} a^{:A} \to a \times 1 \quad ,$$

where we denoted by 1 the value `()` of the `Unit` type. We find:

$$(f_1 \,{}^{\circ}\!{}_{;}\, f_2)(a^{:A} \times 1) = f_2\,(f_1(a \times 1)) = f_2\,(a) = a \times 1 \quad ,$$
$$(f_2 \,{}^{\circ}\!{}_{;}\, f_1)(a^{:A}) = f_1(f_2(a)) = f_1(a \times 1) = a \quad .$$

This shows that both compositions are identity functions. Another way of writing the proof is by computing the function compositions symbolically, without applying to a value $a^{:A}$:

$$f_1 \,{}^{\circ}\!{}_{;}\, f_2 = (a \times 1 \to a) \,{}^{\circ}\!{}_{;}\, (a \to a \times 1) = (a \times 1 \to a \times 1) = \text{id}^{A \times \mathbb{1}} \quad ,$$
$$f_2 \,{}^{\circ}\!{}_{;}\, f_1 = (a \to a \times 1) \,{}^{\circ}\!{}_{;}\, (a \times 1 \to a) = (a \to a) = \text{id}^{A} \quad .$$

Example 5.3.2.4 Verify the type equivalence $A + B \cong B + A$.

Solution The corresponding Scala types are `Either[A, B]` and `Either[B, A]`. We use pattern matching to implement the functions required for the type equivalence:

```
def f1[A, B]: Either[A, B] => Either[B, A] = {
  case Left(a)    => Right(a) // No other choice here.
  case Right(b)   => Left(b)  // No other choice here.
}
def f2[A, B]: Either[B, A] => Either[A, B] = f1[B, A]
```

The functions `f1` and `f2` are implemented by code that can be derived unambiguously from the type signatures. For instance, the line `case Left(a) => ...` is required to return a value of type `Either[B, A]` by using a given value `a:A`. The only way of doing that is by returning `Right(a)`.

It is clear from the code that the functions `f1` and `f2` are inverses of each other. To verify that rigorously, we need to show that `f1 andThen f2` is equal to an identity function. The function `f1 andThen f2` applies `f2` to the result of `f1`. The code of `f1` contains two `case ...` lines, each returning a result. So, we need to apply `f2` separately in each line. Evaluate the code symbolically:

```
(f1 andThen f2) == {
  case Left(a)    => f2(Right(a))
  case Right(b)   => f2(Left(b))
} == {
  case Left(a)    => Left(a)
  case Right(b)   => Right(b)
}
```

The result is a function of type `Either[A, B] => Either[A, B]` that does not change its argument; so, it is equal to the identity function.

Let us now write the function `f1` in the code notation and perform the same derivation. We will also develop a useful notation for functions operating on disjunctive types.

The pattern matching construction in the Scala code of `f1` is similar to a pair of functions with types `A => Either[B, A]` and `B => Either[B, A]`. One of these functions is applied depending on whether the argument of `f1` has type $A + \mathbb{0}$ or $\mathbb{0} + B$. So, we may write the code of `f1` as:

$$f_1 \stackrel{\text{def}}{=} x^{:A+B} \to \begin{cases} \text{if } x = a^{:A} + \mathbb{0}^{:B} & : \quad \mathbb{0}^{:B} + a^{:A} \\ \text{if } x = \mathbb{0}^{:A} + b^{:B} & : \quad b^{:B} + \mathbb{0}^{:A} \end{cases}$$

Since both the argument and the result of f_1 are disjunctive types with 2 parts each, it is convenient to write the code of f_1 as a 2×2 *matrix* that maps the input parts to the output parts:

```
def f1[A, B]: Either[A, B] => Either[B, A] = {
  case Left(a)    => Right(a)
  case Right(b)   => Left(b)
}
```

$$f_1 \stackrel{\text{def}}{=} \left| \begin{array}{c||cc} & B & A \\ \hline A & \mathbb{0} & a^{:A} \to a \\ B & b^{:B} \to b & \mathbb{0} \end{array} \right| \quad .$$

The rows of the matrix correspond to the `case` rows in the Scala code. There is one row for each part of the disjunctive type of the argument. The columns of the matrix correspond to the parts of the disjunctive type of the result. The double line shows where the argument types are.

The code of f_2 is written similarly. Let us rename arguments for clarity:

```
def f2[A, B]: Either[B, A] => Either[A, B] = {
  case Left(y)    => Right(y)
  case Right(x)   => Left(x)
}
```

$$f_2 \stackrel{\text{def}}{=} \left| \begin{array}{c||cc} & A & B \\ \hline B & \mathbb{0} & y^{:B} \to y \\ A & x^{:A} \to x & \mathbb{0} \end{array} \right| \quad .$$

The forward composition $f_1 ⨾ f_2$ is computed by the standard rules of row-by-column matrix multiplication.[6] Any terms containing $\mathbb{0}$ are omitted, and the remaining functions are composed:

$$f_1 ⨾ f_2 = \left| \begin{array}{c||cc} & B & A \\ \hline A & \mathbb{0} & a^{:A} \to a \\ B & b^{:B} \to b & \mathbb{0} \end{array} \right| ⨾ \left| \begin{array}{c||cc} & A & B \\ \hline B & \mathbb{0} & y^{:B} \to y \\ A & x^{:A} \to x & \mathbb{0} \end{array} \right|$$

matrix multiplication/composition :

$$= \left| \begin{array}{c||cc} & A & B \\ \hline A & (a^{:A} \to a) ⨾ (x^{:A} \to x) & \mathbb{0} \\ B & \mathbb{0} & (b^{:B} \to b) ⨾ (y^{:B} \to y) \end{array} \right|$$

function composition :

$$= \left| \begin{array}{c||cc} & A & B \\ \hline A & \text{id} & \mathbb{0} \\ B & \mathbb{0} & \text{id} \end{array} \right| = \text{id}^{:A+B \to A+B} \quad .$$

Several features of the matrix notation are helpful in such calculations. The parts of the code of f_1 are automatically composed with the corresponding parts of the code of f_2. To check that the types match in the function composition, we just need to compare the types in the output row $\left\| \begin{array}{cc} B & A \end{array} \right|$ of f_1 with the input column $\left| \begin{array}{c} B \\ A \end{array} \right\|$ of f_2. Once we verified that all types match, we may omit the type

[6] https://en.wikipedia.org/wiki/Matrix_multiplication

annotations and write the same derivation more concisely as:

$$
\begin{aligned}
f_1 \fatsemi f_2 &= \left\| \begin{matrix} \mathbb{0} & a^{:A} \to a \\ b^{:B} \to b & \mathbb{0} \end{matrix} \right| \fatsemi \left\| \begin{matrix} \mathbb{0} & y^{:B} \to y \\ x^{:A} \to x & \mathbb{0} \end{matrix} \right| \\
\text{matrix multiplication/composition :} \quad &= \left\| \begin{matrix} (a^{:A} \to a) \fatsemi (x^{:A} \to x) & \mathbb{0} \\ \mathbb{0} & (b^{:B} \to b) \fatsemi (y^{:B} \to y) \end{matrix} \right| \\
\text{function composition :} \quad &= \left\| \begin{matrix} \mathrm{id} & \mathbb{0} \\ \mathbb{0} & \mathrm{id} \end{matrix} \right| = \mathrm{id} \quad .
\end{aligned}
$$

The identity function is represented by the diagonal matrix: $\left\| \begin{matrix} \mathrm{id} & \mathbb{0} \\ \mathbb{0} & \mathrm{id} \end{matrix} \right|$.

Exercise 5.3.2.5 Verify the type equivalence $A \times B \cong B \times A$.

Exercise 5.3.2.6 Verify the type equivalence $(A + B) + C \cong A + (B + C)$. Since Section 5.3.1 proved the equivalences $(A + B) + C \cong A + (B + C)$ and $(A \times B) \times C \cong A \times (B \times C)$, we may write $A + B + C$ and $A \times B \times C$ without any parentheses.

Exercise 5.3.2.7 Verify the type equivalence:

$$(A + B) \times (A + B) = A \times A + \mathbb{2} \times A \times B + B \times B \quad ,$$

where $\mathbb{2}$ denotes the `Boolean` type (which may be defined as $\mathbb{2} \stackrel{\text{def}}{=} \mathbb{1} + \mathbb{1}$).

5.3.3 Type cardinalities and type equivalence

To understand why type equivalences are related to arithmetic identities, consider the question of how many different values a given type can have.

Begin by counting the number of distinct values for simple types. For example, the `Unit` type has only one distinct value; the type `Nothing` has zero values; the `Boolean` type has two distinct values (`true` and `false`); and the type `Int` has 2^{32} distinct values.

It is more difficult to count the number of distinct values in a type such as `String`, which is equivalent to a list of unknown length, `List[Char]`. However, each computer's memory is limited, so there will exist a maximum length for values of type `String`. So, the total number of possible different strings will be finite (but will depend on the computer).

For a given type A, let us denote by $|A|$ the number of distinct values of type A. The number $|A|$ is called the **cardinality** of type A. This is the same as the number of elements in the set of all values of type A. Since any computer's memory is finite, there will be *finitely* many different values of a given type A that can exist in the computer. So, we may assume that $|A|$ is always a finite integer value. This assumption will simplify our reasoning. We will not actually need to compute the precise number of, say, all the different possible strings. It is sufficient to know that the set of all strings is finite, so that we can denote its cardinality by |String|.

The next step is to consider the cardinality of types such as $A \times B$ and $A + B$. If the types A and B have cardinalities $|A|$ and $|B|$, it follows that the set of all distinct pairs `(A, B)` has $|A| \times |B|$ elements. So, the cardinality of the type $A \times B$ is equal to the (arithmetic) product of the cardinalities of A and B. The set of all pairs, denoted in mathematics by:

$$\{(a, b) | a \in A, b \in B\} \quad ,$$

is called the **Cartesian product** of sets A and B, and is denoted by $A \times B$. For this reason, the tuple type is also called the **product type**. Accordingly, the type notation adopts the symbol $\times$ for the product type.

The set of all distinct values of the type $A + B$, i.e., of the Scala type `Either[A, B]`, is a labeled union of the set of values of the form `Left(a)` and the set of values of the form `Right(b)`. It is clear that the cardinalities of these two sets are equal to $|A|$ and $|B|$ respectively. So, the cardinality of the type `Either[A, B]` is equal to $|A| + |B|$. For this reason, disjunctive types are also called **sum types**, and the type notation adopts the symbol + for these types.

We can write our conclusions as:

$$|A \times B| = |A| \times |B| \quad ,$$
$$|A + B| = |A| + |B| \quad .$$

The type notation, $A \times B$ for `(A,B)` and $A + B$ for `Either[A, B]`, translates directly into type cardinalities.

The last step is to notice that two types can be equivalent, $P \cong Q$, only if their cardinalities are equal, $|P| = |Q|$. When the cardinalities are not equal, $|P| \neq |Q|$, it will be impossible to have a one-to-one correspondence between the sets of values of type P and values of type Q. So, it will be impossible to convert values from type P to type Q and back without loss of information.

We conclude that types are equivalent when a logical identity *and* an arithmetic identity hold.

The presence of both identities does not automatically guarantee a useful type equivalence. The fact that information in one type can be identically stored in another type does not necessarily mean that it is helpful to do so in a given application.

For example, the types `Option[Option[A]]` and `Either[Boolean, A]` are equivalent because both types contain $2 + |A|$ distinct values. The short notation for these types is $\mathbb{1} + \mathbb{1} + A$ and $\mathbb{2} + A$ respectively. The type Boolean is denoted by $\mathbb{2}$ since it has only *two* distinct values.

One could write code for converting between these types without loss of information:

```
def f1[A]: Option[Option[A]] => Either[Boolean, A] = {
  case None           => Left(false) // Or maybe Left(true)?
  case Some(None)     => Left(true)
  case Some(Some(x))  => Right(x)
}

def f2[A]: Either[Boolean, A] => Option[Option[A]] = {
  case Left(false)    => None
  case Left(true)     => Some(None)
  case Right(x)       => Some(Some(x))
}
```

The presence of an arbitrary choice in this code is a warning sign. In `f1`, we could map `None` to `Left(false)` or to `Left(true)` and adjust the rest of the code accordingly. The type equivalence holds with either choice. So, these types *are* equivalent, but there is no "natural" choice of the conversion functions `f1` and `f2` that will work correctly in all applications, because the meaning of these data types will be application-dependent. We call this type equivalence **accidental**.

Example 5.3.3.1 Are the types `Option[A]` and `Either[Unit, A]` equivalent? Check whether the corresponding logic identity and arithmetic identity hold.

Solution Begin by writing the given types in the type notation: `Option[A]` is written as $\mathbb{1} + A$, and `Either[Unit, A]` is written also as $\mathbb{1} + A$. The notation already indicates that the types are equivalent. But let us verify explicitly that the type notation is not misleading us here.

To establish the type equivalence, we need to implement two fully parametric functions:

```
def f1[A]: Option[A] => Either[Unit, A] = ???
def f2[A]: Either[Unit, A] => Option[A] = ???
```

These functions must satisfy $f_1 ⨾ f_2 = \text{id}$ and $f_2 ⨾ f_1 = \text{id}$. It is straightforward to implement `f1` and `f2`:

```
def f1[A]: Option[A] => Either[Unit, A] = {
  case None      => Left(())
  case Some(x)   => Right(x)
}
def f2[A]: Either[Unit, A] => Option[A] = {
  case Left(())   => None
```

```
  case Right(x)   => Some(x)
}
```

The code clearly shows that `f1` and `f2` are inverses of each other. This verifies the type equivalence.

The logic identity is $True \vee A = True \vee A$ and holds trivially. It remains to check the arithmetic identity, which relates the cardinalities of types `Option[A]` and `Either[Unit, A]`. Assume that the cardinality of type `A` is $|A|$. Any possible value of type `Option[A]` must be either `None` or `Some(x)`, where `x` is a value of type `A`. So, the number of distinct values of type `Option[A]` is $1 + |A|$. All possible values of type `Either[Unit, A]` are of the form `Left(())` or `Right(x)`, where `x` is a value of type `A`. So, the cardinality of type `Either[Unit, A]` is $1 + |A|$. We see that the arithmetic identity holds: the types `Option[A]` and `Either[Unit, A]` have equally many distinct values.

This example shows that the type notation is helpful for reasoning about type equivalences. The answer was found immediately when we wrote the type notation $(\mathbb{1} + A)$ for the given types.

5.3.4 Type equivalence involving function types

Until now, we have looked at product types and disjunctive types. We now turn to type constructions involving function types.

Consider two types A and B having known cardinalities $|A|$ and $|B|$. How many distinct values does the function type $A \to B$ have? A function `f: A => B` needs to select a value of type B for each possible value of type A. Therefore, the number of different functions `f: A => B` is $|B|^{|A|}$.

Here $|B|^{|A|}$ denotes the **numeric exponent**, that is, $|B|$ to the power $|A|$. We use the numeric exponent notation (a^b) only when computing with numbers. When denoting types and code, this book uses superscripts for type parameters and type annotations.

For the types $A = B = \text{Int}$, we have $|A| = |B| = 2^{32}$, and the exponential formula gives:

$$|A \to B| = (2^{32})^{(2^{32})} = 2^{32 \times 2^{32}} = 2^{2^{37}} \approx 10^{4.1 \times 10^{10}} \quad .$$

This number greatly exceeds the number of atoms in the observable Universe![7] However, almost all of those functions will map integers to integers in extremely complicated (and practically useless) ways. The code of those functions will be much larger than the available memory of a realistic computer. So, the number of practically implementable functions of type $A \to B$ can be much smaller than $|B|^{|A|}$. Since the code of a function is a string of bytes that needs to fit into the computer's memory, the number of implementable functions is no larger than the number of possible byte strings.

Nevertheless, the formula $|B|^{|A|}$ is useful since it shows the number of distinct functions that are possible in principle. When types A and B have only a small number of distinct values (for example, $A =$ `Option[Boolean]]` and $B =$ `Either[Boolean, Boolean]`), the formula $|B|^{|A|}$ gives an exact and practically relevant answer.

Let us now look for logic identities and arithmetic identities involving function types. Table 5.6 lists the available identities and the corresponding type equivalences. (In the last column, we defined $a \stackrel{\text{def}}{=} |A|$, $b \stackrel{\text{def}}{=} |B|$, and $c \stackrel{\text{def}}{=} |C|$ for brevity.)

It is notable that no logic identity is available for the formula $\alpha \Rightarrow (\beta \vee \gamma)$, and correspondingly no type equivalence is available for the type expression $A \to B + C$ (although there is an identity for $A \to B \times C$). Type reasoning becomes more complicated in the presence of type expressions of the form $A \to B + C$ because those expressions cannot be transformed into simpler type formulas.

We will now prove some of the type identities in Table 5.6.

Example 5.3.4.1 Verify the type equivalence $\mathbb{1} \to A \cong A$.

Solution Recall that the type notation $\mathbb{1} \to A$ means the Scala function type `Unit => A`. There is only one value of type `Unit`. The choice of a function of type `Unit => A` is the same as the choice of a value of type `A`. So, the type $\mathbb{1} \to A$ has $|A|$ distinct values, and the arithmetic identity holds.

To verify the type equivalence explicitly, we need to implement two functions:

[7] Estimated to be between 10^{78} and 10^{82}, see https://www.universetoday.com/36302/atoms-in-the-universe/amp/

Logical identity (if holds)	Type equivalence	Arithmetic identity
$(True \Rightarrow \alpha) = \alpha$	$\mathbb{1} \to A \cong A$	$a^1 = a$
$(False \Rightarrow \alpha) = True$	$\mathbb{0} \to A \cong \mathbb{1}$	$a^0 = 1$
$(\alpha \Rightarrow True) = True$	$A \to \mathbb{1} \cong \mathbb{1}$	$1^a = 1$
$(\alpha \Rightarrow False) \neq False$	$A \to \mathbb{0} \ncong \mathbb{0}$	$0^a \neq 0$
$(\alpha \vee \beta) \Rightarrow \gamma = (\alpha \Rightarrow \gamma) \wedge (\beta \Rightarrow \gamma)$	$A + B \to C \cong (A \to C) \times (B \to C)$	$c^{a+b} = c^a \times c^b$
$(\alpha \wedge \beta) \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)$	$A \times B \to C \cong A \to B \to C$	$c^{a \times b} = (c^b)^a$
$\alpha \Rightarrow (\beta \wedge \gamma) = (\alpha \Rightarrow \beta) \wedge (\alpha \Rightarrow \gamma)$	$A \to B \times C \cong (A \to B) \times (A \to C)$	$(b \times c)^a = b^a \times c^a$

Table 5.6: Logical identities with implication, and the corresponding type equivalences and arithmetic identities.

```
def f1[A]: (Unit => A) => A = ???
def f2[A]: A => Unit => A = ???
```

The first function needs to produce a value of type `A`, given an argument of the function type `Unit => A`. The only possibility is to apply that function to the value of type `Unit`. We can always produce that value as `()`:

```
def f1[A]: (Unit => A) => A = (h: Unit => A) => h(())
```

Implementing `f2` is straightforward. We can just discard the `Unit` argument:

```
def f2[A]: A => Unit => A = (x: A) => _ => x
```

It remains to show that the functions `f1` and `f2` are inverses of each other. Let us show the proof using Scala code and then using the code notation.

Writing Scala code, compute `f1(f2(x))` for an arbitrary `x:A`. Using the code of `f1` and `f2`, we get:

```
f1(f2(x)) == f1(_ => x) == (_ => x)(()) == x
```

Now compute `f2(f1(h))` for arbitrary `h: Unit => A` in Scala code:

```
f2(f1(h)) == f2(h(())) == { _ => h(()) }
```

How can we show that the function `{_ => h(())}` is equal to `h`? Whenever we apply equal functions to equal arguments, they return equal results. In our case, the argument of `h` is of type `Unit`, so we only need to verify that the result of applying `h` to the value `()` is the same as the result of applying `{_ => h(())}` to `()`. In other words, we need to apply both sides to an additional argument `()`:

```
f2(f1(h))(()) == { _ => h(()) } (()) == h(())
```

This completes the proof.

For comparison, let us show the same proof in the code notation. The functions f_1 and f_2 are:

$$f_1 \stackrel{\text{def}}{=} h^{:\mathbb{1} \to A} \to h(1) \quad , \qquad f_2 \stackrel{\text{def}}{=} x^{:A} \to 1 \to x \quad .$$

Now write the function compositions in both directions:

$$\begin{aligned} \text{expect to equal id}: \quad & f_1 \,;\, f_2 = (h^{:\mathbb{1} \to A} \to h(1)) \,;\, (x^{:A} \to 1 \to x) \\ \text{compute composition}: \quad & = h^{:\mathbb{1} \to A} \to \underline{1 \to h(1)} \\ \text{note that } 1 \to h(1) \text{ is the same as } h: \quad & = (h^{:\mathbb{1} \to A} \to h) = \text{id} \quad . \end{aligned}$$

$$\begin{aligned}
\text{expect to equal id}: \quad & f_2 \fatsemi f_1 = (x^{:A} \to 1 \to x) \fatsemi (h^{:\mathbb{1}\to A} \to h(1)) \\
\text{compute composition}: \quad & = x^{:A} \to \underline{(1 \to x)(1)} \\
\text{apply function}: \quad & = (x^{:A} \to x) = \text{id} \quad .
\end{aligned}$$

The type $\mathbb{1} \to A$ is equivalent to the type A in the sense of carrying the same information, but these types are not exactly the same. An important difference between these types is that a value of type A is available immediately, while a value of type $\mathbb{1} \to A$ is a function that still needs to be applied to an argument (of type $\mathbb{1}$) before a value of type A is obtained. The type $\mathbb{1} \to A$ may represent an "on-call" value of type A. That value is computed on demand every time it is requested. (See Section 2.6.3 for more details about "on-call" values.)

The void type $\mathbb{0}$ needs special reasoning, as the next examples show:

Example 5.3.4.2 Verify the type equivalence $\mathbb{0} \to A \cong \mathbb{1}$.

Solution To verify that a type X is equivalent to the `Unit` type, we need to show that there is only one distinct value of type X. So, let us find out how many values the type $\mathbb{0} \to A$ has. Consider a value of that type, which is a function $f^{:\mathbb{0}\to A}$ from the type $\mathbb{0}$ to a type A. Since there exist no values of type $\mathbb{0}$, the function f will never be applied to any arguments and so *does not need* to compute any actual values of type A. So, f is a function whose body may be "empty". At least, f's body does not need to contain any expressions of type A. In Scala, such a function can be written as:

```
def absurd[A]: Nothing => A = { ??? }
```

This code will compile without type errors. An equivalent code is:

```
def absurd[A]: Nothing => A = { x => ??? }
```

The symbol `???` is defined in the Scala library and represents code that is "not implemented". Trying to evaluate this symbol will produce an error:

```
scala> val x = ???
scala.NotImplementedError: an implementation is missing
```

Since the function `absurd` can never be applied to an argument, this error will never happen. So, one can pretend that the result value (which will never be computed) has any required type, e.g., the type A. In this way, the compiler will accept the definition of `absurd`.

Let us now verify that there exists *only one* distinct function of type $\mathbb{0} \to A$. Take any two functions of that type, $f^{:\mathbb{0}\to A}$ and $g^{:\mathbb{0}\to A}$. Are they different? The only way of showing that f and g are different is by finding a value x such that $f(x) \neq g(x)$. But then x would be of type $\mathbb{0}$, and there are *no values* of type $\mathbb{0}$. So, we will never be able to find the required value x. It follows that any two functions f and g of type $\mathbb{0} \to A$ are equal, $f = g$. In other words, there exists only one distinct value of type $\mathbb{0} \to A$. Since the cardinality of the type $\mathbb{0} \to A$ is 1, we obtain the type equivalence $\mathbb{0} \to A \cong \mathbb{1}$.

Example 5.3.4.3 Show that $A \to \mathbb{0} \ncong \mathbb{0}$ and $A \to \mathbb{0} \ncong \mathbb{1}$, where A is an arbitrary type.

Solution To prove that two types are *not* equivalent, it is sufficient to show that their cardinalities are different. Let us determine the cardinality of the type $A \to \mathbb{0}$, assuming that the cardinality of A is known. We note that a function of type, say, $\text{Int} \to \mathbb{0}$ is impossible to implement. (If we had such a function $f^{:\text{Int}\to\mathbb{0}}$, we could evaluate, say, $x \stackrel{\text{def}}{=} f(123)$ and obtain a value x of type $\mathbb{0}$, which is impossible by definition of the type $\mathbb{0}$. It follows that $|\text{Int} \to \mathbb{0}| = 0$. However, Example 5.3.4.2 shows that $\mathbb{0} \to \mathbb{0}$ has cardinality 1. So, we find that $|A \to \mathbb{0}| = 1$ if the type A is itself $\mathbb{0}$ but $|A \to \mathbb{0}| = 0$ for all other types A. We conclude that the type $A \to \mathbb{0}$ is not equivalent to $\mathbb{0}$ or $\mathbb{1}$ when A is an unknown type. The type $A \to \mathbb{0}$ is void only for non-void types A, and vice versa.

Example 5.3.4.4 Verify the type equivalence $A \to \mathbb{1} \cong \mathbb{1}$.

Solution There is only one fully parametric function that returns $\mathbb{1}$:

```
def f[A]: A => Unit = { _ => () }
```

The function f cannot use its argument of type A since nothing is known about the type A. So, the code of f *must* discard its argument and return the fixed value `()` of type `Unit`. In the code notation,

this function is written as:

$$f^{:A\to\mathbb{1}} \stackrel{\text{def}}{=} _ \to 1 \quad .$$

We can show that there exist only *one* distinct function of type $A \to \mathbb{1}$ (that is, the type $A \to \mathbb{1}$ has cardinality 1). Assume that f and g are two such functions, and try to find a value $x^{:A}$ such that $f(x) \neq g(x)$. We cannot find any such x because $f(x) = 1$ and $g(x) = 1$ for all x. So, any two functions f and g of type $A \to \mathbb{1}$ must be equal to each other. The cardinality of the type $A \to \mathbb{1}$ is 1.

Any type having cardinality 1 is equivalent to the `Unit` type ($\mathbb{1}$). So, $A \to \mathbb{1} \cong \mathbb{1}$.

Example 5.3.4.5 Verify the following type equivalence:

$$A + B \to C \cong (A \to C) \times (B \to C) \quad .$$

Solution Begin by implementing two functions with type signatures:

```
def f1[A,B,C]: (Either[A, B] => C) => (A => C, B => C) = ???
def f2[A,B,C]: ((A => C, B => C)) => Either[A, B] => C = ???
```

The code can be derived unambiguously from the type signatures. For the first function, we need to produce a pair of functions of type `(A => C, B => C)`. Can we produce the first part of that pair? Computing a function of type `A => C` means that we need to produce a value of type `C` given an arbitrary value `a:A`. The available data is a function of type `Either[A, B] => C` called, say, `h`. We can apply that function to `Left(a)` and obtain a value of type `C` as required. So, a function of type `A => C` is computed as `a => h(Left(a))`. We can produce a function of type `B => C` similarly. The code is:

```
def f1[A,B,C]: (Either[A, B] => C) => (A => C, B => C) =
  (h: Either[A, B] => C) => (a => h(Left(a)), b => h(Right(b)))
```

We write this function in the code notation like this:

$$f_1 : (A + B \to C) \to (A \to C) \times (B \to C) \quad ,$$
$$f_1 \stackrel{\text{def}}{=} h^{:A+B\to C} \to \left(a^{:A} \to h(a + \mathbb{0}^{:B})\right) \times \left(b^{:B} \to h(\mathbb{0}^{:A} + b)\right) \quad .$$

For the function `f2`, we need to apply pattern matching to both curried arguments and then return a value of type `C`. This can be achieved in only one way:

```
def f2[A,B,C](f: A => C, g: B => C): Either[A, B] => C = {
  case Left(a)    => f(a)
  case Right(b)   => g(b)
}
```

We write this function in the code notation like this:

$$f_2 : (A \to C) \times (B \to C) \to A + B \to C \quad ,$$

$$f_2 \stackrel{\text{def}}{=} f^{:A\to C} \times g^{:B\to C} \to \begin{array}{|c||c|} & C \\ \hline A & a \to f(a) \\ B & b \to g(b) \end{array} \quad .$$

The matrix in the last line has only one column because the result type, C, is not known to be a disjunctive type. We may also simplify the functions, e.g., replace $a \to f(a)$ by just f, and write:

$$f_2 \stackrel{\text{def}}{=} f^{:A\to C} \times g^{:B\to C} \to \begin{array}{|c||c|} & C \\ \hline A & f \\ B & g \end{array} \quad .$$

It remains to verify that $f_1 ⨾ f_2 = \text{id}$ and $f_2 ⨾ f_1 = \text{id}$. To compute $f_1 ⨾ f_2$, we write (omitting types):

$$f_1 ⨾ f_2 = \left(h \to (a \to h(a+\mathbb{0})) \times (b \to h(\mathbb{0}+b))\right) ⨾ \left(f \times g \to \left\| \begin{array}{c} f \\ g \end{array} \right|\right)$$

$$\text{compute composition :} \quad = h \to \left\| \begin{array}{l} a \to h(a+\mathbb{0}) \\ b \to h(\mathbb{0}+b) \end{array} \right| \quad .$$

To proceed, we need to simplify the expressions $h(a+\mathbb{0})$ and $h(\mathbb{0}+b)$. We rewrite the argument h (an arbitrary function of type $A+B \to C$) in the matrix notation:

$$h \stackrel{\text{def}}{=} \left| \begin{array}{c|c} & C \\ \hline A & a \to p(a) \\ B & b \to q(b) \end{array} \right| = \left| \begin{array}{c|c} & C \\ \hline A & p \\ B & q \end{array} \right| \quad ,$$

where $p^{:A\to C}$ and $q^{:B\to C}$ are new arbitrary functions. Since we already checked the types, we can omit all type annotations and write h as:

$$h \stackrel{\text{def}}{=} \left\| \begin{array}{c} p \\ q \end{array} \right| \quad .$$

To evaluate expressions such as $h(a+\mathbb{0})$ and $h(\mathbb{0}+b)$, we need to use one of the rows of this matrix. The correct row will be selected *automatically* by the rules of matrix multiplication if we place a row vector to the left of the matrix and use the convention of omitting terms containing $\mathbb{0}$:

$$\left| \begin{array}{cc} a & \mathbb{0} \end{array} \right| ▷ \left\| \begin{array}{c} p \\ q \end{array} \right| = a ▷ p \quad , \qquad \left| \begin{array}{cc} \mathbb{0} & b \end{array} \right| ▷ \left\| \begin{array}{c} p \\ q \end{array} \right| = b ▷ q \quad .$$

Here we used the symbol ▷ to separate an argument from a function when the argument is written to the *left* of the function. The symbol ▷ (pronounced "pipe") is defined by $x ▷ f \stackrel{\text{def}}{=} f(x)$. In Scala, this operation is available as `x.pipe(f)` as of Scala 2.13.

We can write values of disjunctive types, such as $a+\mathbb{0}$, as row vectors $\left| \begin{array}{cc} a & \mathbb{0} \end{array} \right|$:

$$h(a+\mathbb{0}) = (a+\mathbb{0}) ▷ h = \left| \begin{array}{cc} a & \mathbb{0} \end{array} \right| ▷ h \quad . \tag{5.21}$$

With these notations, we compute further. Omit all terms applying $\mathbb{0}$ or applying something to $\mathbb{0}$:

$$\left| \begin{array}{cc} a & \mathbb{0} \end{array} \right| ▷ h = \left| \begin{array}{cc} a & \mathbb{0} \end{array} \right| ▷ \left\| \begin{array}{c} p \\ q \end{array} \right| = a ▷ p = p(a) \quad ,$$

$$\left| \begin{array}{cc} \mathbb{0} & b \end{array} \right| ▷ h = \left| \begin{array}{cc} \mathbb{0} & b \end{array} \right| ▷ \left\| \begin{array}{c} p \\ q \end{array} \right| = b ▷ q = q(b) \quad .$$

Now we can complete the proof of $f_1 ⨾ f_2 = \text{id}$:

$$f_1 ⨾ f_2 = h \to \left\| \begin{array}{l} a \to h(a+\mathbb{0}) \\ b \to h(\mathbb{0}+b) \end{array} \right|$$

$$\text{previous equations :} \quad = \left\| \begin{array}{c} p \\ q \end{array} \right| \to \left\| \begin{array}{l} a \to p(a) \\ b \to q(b) \end{array} \right|$$

$$\text{simplify functions :} \quad = \left\| \begin{array}{c} p \\ q \end{array} \right| \to \left\| \begin{array}{c} p \\ q \end{array} \right| = \text{id} \quad .$$

To prove that $f_2 ⨾ f_1 = \text{id}$, use the notation (5.21):

$$f_2 ⨾ f_1 = \left(f \times g \to \left\| \begin{array}{c} f \\ g \end{array} \right\| \right) ⨾ \left(h \to (a \to (a + \mathbb{0}) \triangleright h) \times (b \to (\mathbb{0} + b) \triangleright h) \right)$$

$$\text{compute composition}: \quad = f \times g \to \left(a \to \underline{\left| \begin{array}{cc} a & \mathbb{0} \end{array} \right| \triangleright \left\| \begin{array}{c} f \\ g \end{array} \right\|} \right) \times \left(b \to \underline{\left| \begin{array}{cc} \mathbb{0} & b \end{array} \right| \triangleright \left\| \begin{array}{c} f \\ g \end{array} \right\|} \right)$$

$$\text{apply functions}: \quad = f \times g \to (a \to \underline{a \triangleright f}) \times (b \to \underline{b \triangleright g})$$

$$\text{definition of } \triangleright: \quad = f \times g \to \underline{(a \to f(a))} \times \underline{(b \to g(b))}$$

$$\text{simplify functions}: \quad = (f \times g \to f \times g) = \text{id} \quad .$$

In this way, we have proved that f_1 and f_2 are mutual inverses. The proofs appear long because we took time to motivate and introduce new notation for applying matrices to row vectors. Once this notation is understood, the proof for $f_1 ⨾ f_2 = \text{id}$ can be written as:

$$f_1 ⨾ f_2 = (h \to (a \to (a + \mathbb{0}) \triangleright h) \times (b \to (\mathbb{0} + b) \triangleright h)) ⨾ \left(f \times g \to \left\| \begin{array}{c} f \\ g \end{array} \right\| \right)$$

$$\text{compute composition}: \quad = h \to \left\| \begin{array}{c} a \to \left| \begin{array}{cc} a & \mathbb{0} \end{array} \right| \triangleright h \\ b \to \left| \begin{array}{cc} \mathbb{0} & b \end{array} \right| \triangleright h \end{array} \right\| = \left\| \begin{array}{c} p \\ q \end{array} \right\| \to \left\| \begin{array}{c} a \to \left| \begin{array}{cc} a & \mathbb{0} \end{array} \right| \triangleright \left\| \begin{array}{c} p \\ q \end{array} \right\| \\ b \to \left| \begin{array}{cc} \mathbb{0} & b \end{array} \right| \triangleright \left\| \begin{array}{c} p \\ q \end{array} \right\| \end{array} \right\|$$

$$\text{apply functions}: \quad = \left\| \begin{array}{c} p \\ q \end{array} \right\| \to \left\| \begin{array}{c} a \to a \triangleright p \\ b \to b \triangleright q \end{array} \right\| = \left\| \begin{array}{c} p \\ q \end{array} \right\| \to \left\| \begin{array}{c} p \\ q \end{array} \right\| = \text{id} \quad .$$

Proofs in the code notation are shorter than in Scala syntax because certain names and keywords (such as `Left`, `Right`, `case`, `match`, etc.) are omitted. From now on, we will prefer to use the code notation in proofs, keeping in mind that one can always convert the code notation to Scala.

Note that the function arrow ($\to$) binds weaker than the pipe operation ($\triangleright$), so the code notation $x \to y \triangleright z$ means $x \to (y \triangleright z)$. We will review the code notation more systematically in Chapter 7.

Example 5.3.4.6 Verify the type equivalence:

$$A \times B \to C \cong A \to B \to C \quad .$$

Solution Begin by implementing the two functions:

```
def f1[A,B,C]: (((A, B)) => C) => A => B => C = ???
def f2[A,B,C]: (A => B => C) => ((A, B)) => C = ???
```

The Scala code can be derived from the type signatures unambiguously:

```
def f1[A,B,C]: (((A, B)) => C) => A => B => C = g => a => b => g((a, b))
def f2[A,B,C]: (A => B => C) => ((A, B)) => C = h => { case (a, b) => h(a)(b) }
```

Write these functions in the code notation:

$$f_1 = g^{:A \times B \to C} \to a^{:A} \to b^{:B} \to g(a \times b) \quad ,$$
$$f_2 = h^{:A \to B \to C} \to (a \times b)^{:A \times B} \to h(a)(b) \quad .$$

We denote by $(a \times b)^{:A \times B}$ the argument of type `(A, B)` with pattern matching implied. This notation allows us to write shorter code formulas involving tupled arguments.

Compute the function composition $f_1 ⨾ f_2$:

$$\begin{aligned} f_1 ⨾ f_2 &= (g \to \underline{a \to b \to g(a \times b)}) ⨾ (h \to a \times b \to h(a)(b)) \\ \text{substitute } h = a \to b \to g(a \times b): \quad &= g \to \underline{a \times b \to g(a \times b)} \\ \text{simplify function}: \quad &= (g \to g) = \text{id} \quad . \end{aligned}$$

Compute the function composition $f_2 ⨾ f_1$:

$$\begin{aligned} f_2 ⨾ f_1 &= (h \to \underline{a \times b \to h(a)(b)}) ⨾ (g \to a \to b \to g(a \times b)) \\ \text{substitute } g = a \times b \to h(a)(b): \quad &= h \to a \to \underline{b \to h(a)(b)} \\ \text{simplify function } b \to h(a)(b): \quad &= h \to \underline{a \to h(a)} \\ \text{simplify function } a \to h(a) \text{ to } h: \quad &= (h \to h) = \text{id} \quad . \end{aligned}$$

5.4 Summary

What tasks can we perform now?

- Convert a fully parametric type signature into a logical formula and:
  - Decide whether the type signature can be implemented in code.
  - If possible, derive the code using the CH correspondence.
- Use the type notation (Table 5.1) for reasoning about types to:
  - Decide type equivalence using the rules in Tables 5.5–5.6.
  - Simplify type expressions before writing code.
- Use the matrix notation and the pipe notation to write code that works on disjunctive types.

What tasks *cannot* be performed with these tools?

- Automatically generate code for *recursive* functions. The CH correspondence is based on propositional logic, which cannot describe recursion. Accordingly, recursion is absent from the eight code constructions of Section 5.2.2. Recursive functions need to be coded by hand.
- Automatically generate code satisfying a property (e.g., isomorphism). We may generate some code, but the CH correspondence does not guarantee that properties will hold. We need to verify the required properties manually, after deriving the code.
- Express complicated conditions (e.g., "array is sorted") in a type signature. This can be done using **dependent types** (i.e., types that directly depend on values in some way). This is an advanced technique beyond the scope of this book. Programming languages such as Coq, Agda, and Idris fully support dependent types, while Scala has only limited support.
- Generate code using type constructors with known methods (e.g., the `map` method).

As an example of using type constructors with known methods, consider this type signature:

```
def q[A]: Array[A] => (A => Option[B]) => Array[Option[B]]
```

Can we generate the code of this function from its type signature? We know that the Scala library defines a `map` method on the `Array` class. So, an implementation of `q` is:

```
def q[A]: Array[A] => (A => Option[B]) => Array[Option[B]] = { arr => f => arr.map(f) }
```

However, it is hard to create an *algorithm* that can derive this implementation automatically from the type signature of `q` via the Curry-Howard correspondence. The algorithm would have to convert the type signature of `q` into this logical formula:

$$\mathcal{CH}(\text{Array}^A) \Rightarrow \mathcal{CH}(A \to \text{Opt}^B) \Rightarrow \mathcal{CH}(\text{Array}^{\text{Opt}^B}) \quad . \tag{5.22}$$

To derive an implementation, the algorithm would need to use the available `map` method for `Array`. That method has the type signature:

$$\text{map} : \forall(A, B).\, \text{Array}^A \to (A \to B) \to \text{Array}^B \quad .$$

To derive the $\mathcal{CH}$-proposition (5.22), the algorithm will need to assume that the $\mathcal{CH}$-proposition:

$$\mathcal{CH}\left(\forall(A, B).\, \text{Array}^A \to (A \to B) \to \text{Array}^B\right) \tag{5.23}$$

already holds. In other words Eq. (5.23) is one of the premises of a sequent. Reasoning about premises such as Eq. (5.23) requires **first-order logic** — a logic whose proof rules can handle quantified types such as $\forall(A, B)$ inside premises. However, first-order logic is **undecidable**: no algorithm can find a proof (or verify the absence of a proof) in all cases.

The constructive propositional logic with the rules listed in Table 5.2) is **decidable**, i.e., it has an algorithm that either finds a proof or disproves any given formula. However, that logic cannot handle type constructors such as Array^A. It also cannot handle premises containing type quantifiers such as $\forall(A, B)$, because all the available rules have the quantifiers placed *outside* the premises.

So, code for functions such as `q` can only be derived by trial and error, informed by intuition. This book will help programmers to acquire the necessary intuition and technique.

5.4.1 Solved examples

Example 5.4.1.1 Find the cardinality of the type `P = Option[Option[Boolean] => Boolean]`. Write `P` in the type notation and simplify it to an equivalent type.

Solution Begin with the type `Option[Boolean]`, which can be either `None` or `Some(x)` with an `x:Boolean`. Since the type `Boolean` has 2 possible values, the type `Option[Boolean]` has 3 values:

$$|\text{Opt}^{\text{Boolean}}| = |\mathbb{1} + \text{Boolean}| = 1 + |\text{Boolean}| = 1 + 2 = 3 \quad .$$

In the type notation, `Boolean` is denoted by the symbol $\mathbb{2}$, and the type `Option[Boolean]` by $\mathbb{1} + \mathbb{2}$. So, the type notation $\mathbb{1} + \mathbb{2}$ is consistent with the cardinality 3 of that type:

$$|\mathbb{1} + \text{Boolean}| = |\mathbb{1} + \mathbb{2}| = 1 + 2 = 3 \quad .$$

The function type `Option[Boolean] => Boolean` is denoted by $\mathbb{1} + \mathbb{2} \to \mathbb{2}$. Compute its cardinality as:

$$|\text{Opt}^{\text{Boolean}} \to \text{Boolean}| = |\mathbb{1} + \mathbb{2} \to \mathbb{2}| = |\mathbb{2}|^{|\mathbb{1}+\mathbb{2}|} = 2^3 = 8 \quad .$$

Finally, the we write `P` in the type notation as $P = \mathbb{1} + (\mathbb{1} + \mathbb{2} \to \mathbb{2})$ and find:

$$|P| = |\mathbb{1} + (\mathbb{1} + \mathbb{2} \to \mathbb{2})| = 1 + |\mathbb{1} + \mathbb{2} \to \mathbb{2}| = 1 + 8 = 9 \quad .$$

Example 5.4.1.2 Implement a Scala type `P[A]` given by this type notation:

$$P^A \stackrel{\text{def}}{=} 1 + A + \text{Int} \times A + (\text{String} \to A) \quad .$$

Solution To translate type notation into Scala code, begin by defining the disjunctive types as case classes, choosing class names for convenience. In this case, P^A is a disjunctive type with four parts, so we need four case classes:

```
sealed trait P[A]
final case class P1[A](???) extends P[A]
final case class P2[A](???) extends P[A]
final case class P3[A](???) extends P[A]
final case class P4[A](???) extends P[A]
```

Each of the case classes represents one part of the disjunctive type. Now we write the contents for each of the case classes, in order to implement the data in each of the disjunctive parts:

```
sealed trait P[A]
final case class P1[A]()                  extends P[A]
final case class P2[A](x: A)              extends P[A]
final case class P3[A](n: Int, x: A)      extends P[A]
final case class P4[A](f: String => A)    extends P[A]
```

Example 5.4.1.3 Find an equivalent disjunctive type for the type `P = (Either[A, B], Either[C, D])`.

Solution Begin by writing the given type in the type notation. The tuple becomes the product type, and `Either` becomes the disjunctive (or "sum") type:

$$P \stackrel{\text{def}}{=} (A+B)\times(C+D) \quad .$$

By the usual rules of arithmetic, we expand brackets and obtain an equivalent type:

$$P \cong A\times C + A\times D + B\times C + B\times D \quad .$$

This is a disjunctive type having 4 parts.

Example 5.4.1.4 Show that the following type equivalences do *not* hold: $A+A \ncong A$ and $A\times A \ncong A$, although the corresponding logical identities hold.

Solution Note that the arithmetic equalities do not hold, $A+A \neq A$ and $A\times A \neq A$. This already indicates that the types are not equivalent. To build further intuition, consider that a value of type $A+A$ (in Scala, `Either[A, A]`) is a `Left(a)` or a `Right(a)` for some `a:A`. In the code notation, it is either $a^{:A}+\mathbb{0}$ or $\mathbb{0}+a^{:A}$. So, a value of type $A+A$ contains a value of type A with the additional information about whether it is the first or the second part of the disjunctive type. We cannot represent that information in a single value of type A.

Similarly, a value of type $A\times A$ contains two (possibly different) values of type A, which cannot be represented by a single value of type A without loss of information.

However, the corresponding logical identities $\alpha\vee\alpha=\alpha$ and $\alpha\wedge\alpha=\alpha$ hold. To see that, we could derive the four formulas:

$$\alpha\vee\alpha\Rightarrow\alpha \quad , \qquad \alpha\Rightarrow\alpha\vee\alpha \quad , \qquad \alpha\wedge\alpha\Rightarrow\alpha \quad , \qquad \alpha\Rightarrow\alpha\wedge\alpha \quad ,$$

using the proof rules of Section 5.2.2. Alternatively, we may use the CH correspondence and show that the type signatures:

$$\forall A.\, A+A\to A \quad , \qquad \forall A.\, A\to A+A \quad , \qquad \forall A.\, A\times A\to A \quad , \qquad \forall A.\, A\to A\times A$$

can be implemented via fully parametric functions. For a programmer, it is easier to write code than to guess the correct sequence of proof rules. For the first pair of type signatures, we find:

```
def f1[A]: Either[A, A] => A = {
  case Left(a)    => a   // No other choice here.
  case Right(a)   => a   // No other choice here.
}
def f2[A]: A => Either[A, A] = { a => Left(a) } // Can be also Right(a).
```

The presence of an arbitrary choice, to return `Left(a)` or `Right(a)`, is a warning sign showing that additional information is required to create a value of type `Either[A, A]`. This is precisely the information present in the type $A + A$ but missing in the type A.

The code notation for these functions is:

$$f_1 \stackrel{\text{def}}{=} \left|\begin{array}{c||c} & A \\ \hline A & a^{:A} \to a \\ A & a^{:A} \to a \end{array}\right| = \left|\begin{array}{c||c} & A \\ \hline A & \text{id} \\ A & \text{id} \end{array}\right| \quad , \qquad f_2 \stackrel{\text{def}}{=} a^{:A} \to a + \mathbb{0}^{:A} = \left|\begin{array}{c||cc} & A & A \\ \hline A & a^{:A} \to a & \mathbb{0} \end{array}\right| = \left|\begin{array}{c||cc} & A & A \\ \hline A & \text{id} & \mathbb{0} \end{array}\right| \quad .$$

The composition of these functions is not equal to identity:

$$f_1 ⨾ f_2 = \left\|\begin{array}{c} \text{id} \\ \text{id} \end{array}\right| ⨾ \left\|\begin{array}{cc} \text{id} & \mathbb{0} \end{array}\right| = \left\|\begin{array}{cc} \text{id} & \mathbb{0} \\ \text{id} & \mathbb{0} \end{array}\right| \quad , \quad \text{while we have} \quad \text{id}^{:A+A\to A+A} = \left\|\begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \end{array}\right| \quad .$$

For the second pair of type signatures, the code is:

```
def f1[A]: ((A, A)) => A = { case (a1, a2) => a1 }   // Could be also ‘a2‘.
cef f2[A]:   A => (A, A) = { a => (a, a) }           // No other choice here.
```

It is clear that the first function loses information when it returns `a1` and discards `a2` (or vice versa).

The code notation for the functions `f1` and `f2` is:

$$f_1 \stackrel{\text{def}}{=} a_1^{:A} \times a_2^{:A} \to a_1 = \pi_1^{:A\times A\to A} \quad , \qquad f_2 \stackrel{\text{def}}{=} a^{:A} \to a \times a = \Delta^{:A\to A\times A} \quad .$$

Computing the compositions of these functions, we find that $f_2 ⨾ f_1 = \text{id}$ while $f_1 ⨾ f_2 \neq \text{id}$:

$$\begin{aligned} f_1 ⨾ f_2 &= (a_1 \times a_2 \to a_1) ⨾ (a \to a \times a) \\ &= (a_1 \times a_2 \to a_1 \times a_1) \neq \text{id} = (a_1 \times a_2 \to a_1 \times a_2) \quad . \end{aligned}$$

We have implemented all four type signatures as fully parametric functions, which shows that the corresponding logical formulas are all true (i.e., can be derived using the proof rules). However, the functions cannot be inverses of each other. So, the type equivalences do not hold.

Example 5.4.1.5 Show that $((A \wedge B) \Rightarrow C) \neq (A \Rightarrow C) \vee (B \Rightarrow C)$ in the constructive logic, but the equality holds in Boolean logic. This is another example where the Boolean reasoning fails to give correct answers about implementability of type signatures.

Solution Begin by rewriting the logical equality as two implications:

$$(A \wedge B \Rightarrow C) \Rightarrow (A \Rightarrow C) \vee (B \Rightarrow C) \quad \text{and} \quad ((A \Rightarrow C) \vee (B \Rightarrow C)) \Rightarrow ((A \wedge B) \Rightarrow C) \quad .$$

It is sufficient to show that one of these implications is incorrect. Rather than looking for a proof tree in the constructive logic (which would be difficult, since we need to demonstrate that *no* proof exists), let us use the CH correspondence. According to the CH correspondence, an equivalent task is to implement fully parametric functions with the type signatures:

$$(A \times B \to C) \to (A \to C) + (B \to C) \quad \text{and} \quad (A \to C) + (B \to C) \to A \times B \to C \quad .$$

For the first type signature, the Scala code is:

```
def f1[A,B,C]: (((A, B)) => C) => Either[A => C, B => C] = { k => ??? }
```

We are required to return either a `Left(g)` with `g: A => C`, or a `Right(h)` with `h: B => C`. The only given data is a function `k` of type $A \times B \to C$, so the decision of whether to return a `Left` or a `Right` must be hard-coded in the function `f1` independently of `k`. Can we produce a function `g` of type `A => C`? Given a value of type `A`, we would need to return a value of type `C`. The only way to obtain a value of type `C` is by applying `k` to some arguments. But to apply `k`, we need a value of type `B`, which we do not have. So we cannot produce a `g: A => C`. Similarly, we cannot produce a function `h` of type `B => C`.

We repeat the same argument in the type notation. Obtaining a value of type $(A \to C) + (B \to C)$ means to compute either $g^{:A\to C} + \mathbb{0}$ or $\mathbb{0} + h^{:B\to C}$. This decision must be hard-coded since the only data is a function $k^{:A\times B\to C}$. We can compute $g^{:A\to C}$ only by partially applying $k^{:A\times B\to C}$ to a value of type B. However, we have no values of type B. Similarly, we cannot compute a value $h^{:B\to C}$.

The inverse type signature *can* be implemented:

```
def f2[A,B,C]: Either[A=>C, B=>C] => ((A,B)) => C = {
  case Left(g)    =>   { case (a, b)    => g(a) }
  case Right(h)   =>   { case (a, b)    => h(b) }
}
```

$$f_2 \stackrel{\text{def}}{=} \begin{array}{|c||c|} & A \times B \to C \\ \hline A \to C & g^{:A\to C} \to a \times b \to g(a) \\ B \to C & h^{:B\to C} \to a \times b \to h(b) \end{array} \quad .$$

Let us now show that the logical identity:

$$((\alpha \wedge \beta) \Rightarrow \gamma) = ((\alpha \Rightarrow \gamma) \vee (\beta \Rightarrow \gamma)) \tag{5.24}$$

holds in Boolean logic. A straightforward calculation is to simplify the Boolean expression using Eq. (5.8), which only holds in Boolean logic (but not in the constructive logic). We find:

$$\begin{aligned}
\text{left-hand side of Eq. (5.24)}: &\quad (\alpha \wedge \beta) \underline{\Rightarrow} \gamma \\
\text{use Eq. (5.8)}: &\quad = \underline{\neg(\alpha \wedge \beta)} \vee \gamma \\
\text{use de Morgan's law}: &\quad = \neg\alpha \vee \neg\beta \vee \gamma \quad . \\
\text{right-hand side of Eq. (5.24)}: &\quad \underline{(\alpha \Rightarrow \gamma)} \vee \underline{(\beta \Rightarrow \gamma)} \\
\text{use Eq. (5.8)}: &\quad = \neg\alpha \vee \underline{\gamma} \vee \neg\beta \vee \underline{\gamma} \\
\text{use identity } \gamma \vee \gamma = \gamma: &\quad = \neg\alpha \vee \neg\beta \vee \gamma \quad .
\end{aligned}$$

Both sides of Eq. (5.24) are equal to the same formula, $\neg\alpha \vee \neg\beta \vee \gamma$, so the identity holds.

This calculation does not work in the constructive logic because its proof rules can derive neither the Boolean formula (5.8) nor the **law of de Morgan**, $\neg(\alpha \wedge \beta) = (\neg\alpha \vee \neg\beta)$.

Another way of proving the Boolean identity (5.24) is to enumerate all possible truth values for the variables α, β, and γ. The left-hand side, $(\alpha \wedge \beta) \Rightarrow \gamma$, can be $False$ only if $\alpha \wedge \beta = True$ (that is, both α and β are $True$) and $\gamma = False$. For all other truth values of α, β, and γ, the formula $(\alpha \wedge \beta) \Rightarrow \gamma$ is $True$. Let us determine when the right-hand side, $(\alpha \Rightarrow \gamma) \vee (\beta \Rightarrow \gamma)$, can be $False$. This can happen only if both parts of the disjunction are $False$. That means $\alpha = True$, $\beta = True$, and $\gamma = False$. So, the two sides of the identity (5.24) are both $True$ or both $False$ with any choice of truth values of α, β, and γ. In Boolean logic, this is sufficient to prove the identity (5.24). □

The following example shows how to use the formulas from Tables 5.5–5.6 to derive the type equivalence of complicated type expressions without need for proofs.

Example 5.4.1.6 Use known formulas to verify the type equivalences without direct proofs:

(a) $A \times (A + \mathbb{1}) \times (A + \mathbb{1} + \mathbb{1}) \cong A \times (\mathbb{1} + \mathbb{1} + A \times (\mathbb{1} + \mathbb{1} + \mathbb{1} + A))$.

(b) $\mathbb{1} + A + B \to \mathbb{1} \times B \cong (B \to B) \times (A \to B) \times B$.

Solution **(a)** We can expand brackets in type expressions as in arithmetic:

$$\begin{aligned}
A \times (A + \mathbb{1}) &\cong A \times A + A \times \mathbb{1} \cong A \times A + A \quad , \\
A \times (A + \mathbb{1}) \times (A + \mathbb{1} + \mathbb{1}) &\cong (A \times A + A) \times (A + \mathbb{1} + \mathbb{1}) \\
&\cong A \times A \times A + A \times A + A \times A \times (\mathbb{1} + \mathbb{1}) + A \times (\mathbb{1} + \mathbb{1}) \\
&\cong A \times A \times A + A \times A \times (\mathbb{1} + \mathbb{1} + \mathbb{1}) + A \times (\mathbb{1} + \mathbb{1}) \quad .
\end{aligned}$$

The result looks like a polynomial in A, which we can now rearrange into the required form:

$$A \times A \times A + A \times A \times (\mathbb{1} + \mathbb{1} + \mathbb{1}) + A \times (\mathbb{1} + \mathbb{1}) \cong A \times (\mathbb{1} + \mathbb{1} + A \times (\mathbb{1} + \mathbb{1} + \mathbb{1} + A)) \quad .$$

(b) Keep in mind that the conventions of the type notation make the function arrow ($\to$) group weaker than other type operations. So, the type expression $\mathbb{1} + A + B \to \mathbb{1} \times B$ means a function from $\mathbb{1} + A + B$ to $\mathbb{1} \times B$.

Begin by using the equivalence $\mathbb{1} \times B \cong B$ to obtain $\mathbb{1} + A + B \to B$. Now we use another rule:

$$A + B \to C \cong (A \to C) \times (B \to C)$$

and derive the equivalence:

$$\mathbb{1} + A + B \to B \cong (\mathbb{1} \to B) \times (A \to B) \times (B \to B) \quad .$$

Finally, we note that $\mathbb{1} \to B \cong B$ and that the type product is commutative, so we can rearrange the last type expression into the required form:

$$B \times (A \to B) \times (B \to B) \cong (B \to B) \times (A \to B) \times B \quad .$$

We obtain the required type expression: $(B \to B) \times (A \to B) \times B$.

Example 5.4.1.7 Denote $\text{Read}^{E,A} \stackrel{\text{def}}{=} E \to A$ and implement fully parametric functions with types $A \to \text{Read}^{E,A}$ and $\text{Read}^{E,A} \to (A \to B) \to \text{Read}^{E,B}$.

Solution Begin by defining a type alias for the type constructor $\text{Read}^{E,A}$:

```
type Read[E, A] = E => A
```

The first type signature has only one implementation:

```
def p[E, A]: A => Read[E, A] = { x => _ => x }
```

We *must* discard the argument of type E; we cannot use it for computing a value of type `A` given `x:A`.

The second type signature has three type parameters. It is the curried version of the function `map`:

```
def map[E, A, B]: Read[E, A] => (A => B) => Read[E, B] = ???
```

Expanding the type alias, we see that the two curried arguments are functions of types $E \to A$ and $A \to B$. The forward composition of these functions is a function of type $E \to B$, or $\text{Read}^{E,B}$, which is exactly what we are required to return. So, the code can be written as:

```
def map[E, A, B]: (E => A) => (A => B) => E => B = { r => f => r andThen f }
```

If we did not notice this shortcut, we would reason differently: We are required to compute a value of type B given *three* curried arguments $r^{:E\to A}$, $f^{:A\to B}$, and $e^{:E}$. Write this requirement as:

$$\text{map} \stackrel{\text{def}}{=} r^{:E\to A} \to f^{:A\to B} \to e^{:E} \to ???^{:B} \quad ,$$

The symbol $???^{:B}$ is called a **typed hole**. It stands for a value that we are still figuring out how to compute, but whose type is already known. Typed holes are supported in Scala by an experimental compiler plugin.[8] The plugin will print the known information about the typed hole.

To fill the typed hole $???^{:B}$, we need a value of type B. Since no arguments have type B, the only way of getting a value of type B is to apply $f^{:A\to B}$ to some value of type A. So we write:

$$\text{map} \stackrel{\text{def}}{=} r^{:E\to A} \to f^{:A\to B} \to e^{:E} \to f(???^{:A}) \quad .$$

The only way of getting an A is to apply r to some value of type E:

$$\text{map} \stackrel{\text{def}}{=} r^{:E\to A} \to f^{:A\to B} \to e^{:E} \to f(r(???^{:E})) \quad .$$

We have exactly one value of type E, namely $e^{:E}$. So, the code must be:

$$\text{map}^{E,A,B} \stackrel{\text{def}}{=} r^{:E\to A} \to f^{:A\to B} \to e^{:E} \to f(r(e)) \quad .$$

Translate this to the Scala syntax:

```
def map[E, A, B]: (E => A) => (A => B) => E => B = { r => f => e => f(r(e)) }
```

We may now notice that the expression $e \to f(r(e))$ is a function composition $r \,;\, f$ applied to e, and simplify the code accordingly.

[8] See https://github.com/cb372/scala-typed-holes

Example 5.4.1.8 Show that the type signature `Read[A, T] => (A => B) => Read[B, T]` cannot be implemented as a fully parametric function.

Solution Expand the type signature and try implementing this function:

```
def m[A, B, T] : (A => T) => (A => B) => B => T = { r => f => b => ??? }
```

Given values $r^{:A\to T}$, $f^{:A\to B}$, and $b^{:B}$, we need to compute a value of type T:

$$m = r^{:A\to T} \to f^{:A\to B} \to b^{:B} \to ???^{:T} \quad .$$

The only way of getting a value of type T is to apply r to some value of type A:

$$m = r^{:A\to T} \to f^{:A\to B} \to b^{:B} \to r(???^{:A}) \quad .$$

However, we do not have any values of type A. We have a function $f^{:A\to B}$ that *consumes* values of type A, and we cannot use f to produce any values of type A. So, it seems that we are unable to fill the typed hole $???^{:A}$ and implement the function `m`.

In order to verify that `m` is unimplementable, we need to prove that the logical formula:

$$\forall(\alpha,\beta,\tau).\,(\alpha \Rightarrow \tau) \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \tau) \tag{5.25}$$

is not true in the constructive logic. We could use the `curryhoward` library for that:

```
@ def m[A, B, T] : (A => T) => (A => B) => B => T = implement
cmd1.sc:1: type (A => T) => (A => B) => B => T cannot be implemented
def m[A, B, T] : (A => T) => (A => B) => B => T = implement
                                                 ^
Compilation Failed
```

Another way is to check whether this formula is true in Boolean logic. A formula that holds in constructive logic will always hold in Boolean logic, because all rules shown in Section 5.2.2 preserve Boolean truth values (see Section 5.5.4 for a proof). It follows that any formula that fails to hold in Boolean logic will also not hold in constructive logic.

It is relatively easy to check whether a given Boolean formula is always equal to $True$. Simplifying Eq. (5.25) with the rules of Boolean logic, we find:

$$\begin{aligned}
& (\alpha \Rightarrow \tau) \underline{\Rightarrow} (\alpha \Rightarrow \beta) \underline{\Rightarrow} (\beta \Rightarrow \tau) \\
\text{use Eq. (5.8)}: \quad & = \neg(\underline{\alpha \Rightarrow \tau}) \vee \neg(\underline{\alpha \Rightarrow \beta}) \vee (\underline{\beta \Rightarrow \tau}) \\
\text{use Eq. (5.8)}: \quad & = \underline{\neg(\neg\alpha \vee \tau)} \vee \underline{\neg(\neg\alpha \vee \beta)} \vee (\neg\beta \vee \tau) \\
\text{use de Morgan's law}: \quad & = (\alpha \wedge \neg\tau) \vee \underline{(\alpha \wedge \neg\beta) \vee \neg\beta} \vee \tau \\
\text{use identity } (p \wedge q) \vee q = q: \quad & = \underline{(\alpha \wedge \neg\tau)} \vee \neg\beta \vee \underline{\tau} \\
\text{use identity } (p \wedge \neg q) \vee q = p \vee q: \quad & = \alpha \vee \neg\beta \vee \tau \quad .
\end{aligned}$$

This formula is not identically $True$: it is $False$ when $\alpha = \tau = False$ and $\beta = True$. So, Eq. (5.25) is not true in Boolean logic, therefore it is also not true in constructive logic. By the CH correspondence, we conclude that the type signature of `m` cannot be implemented by a fully parametric function.

Example 5.4.1.9 Define the type constructor $P^A \stackrel{\text{def}}{=} \mathbb{1} + A + A$ and implement `map` for it:

$$\text{map}^{A,B} : P^A \to (A \to B) \to P^B \quad .$$

To check that `map` preserves information, verify the law `map(p)(x => x) == p` for all `p: P[A]`.

Solution It is implied that `map` should be fully parametric and information-preserving. Begin by defining a Scala type constructor for the notation $P^A \stackrel{\text{def}}{=} \mathbb{1} + A + A$:

```
sealed trait P[A]
final case class P1[A]()     extends P[A]
final case class P2[A](x: A) extends P[A]
final case class P3[A](x: A) extends P[A]
```

Now we can write code to implement the required type signature. Each time we have several choices of an implementation, we will choose to preserve information as much as possible.

```
def map[A, B]: P[A] => (A => B) => P[B] =
  p => f => p match {
    case P1()    => P1() // No other choice.
    case P2(x)   => ???
    case P3(x)   => ???
  }
```

In the case `P2(x)`, we are required to produce a value of type P^B from a value $x^{:A}$ and a function $f^{:A\to B}$. Since P^B is a disjunctive type with three parts, we can produce a value of type P^B in three different ways: `P1()`, `P2(...)`, and `P3(...)`. If we return `P1()`, we will lose the information about the value `x`. If we return `P3(...)`, we will preserve the information about `x` but lose the information that the input value was a `P2` rather than a `P3`. By returning `P2(...)` in that scope, we preserve the entire input information.

The value under `P2(...)` must be of type B, and the only way of getting a value of type B is to apply f to x. So, we return `P2(f(x))`.

Similarly, in the case `P3(x)`, we should return `P3(f(x))`. The final code of `map` is:

```
def map[A, B]: P[A] => (A => B) => P[B] = p => f => p match {
  case P1()    => P1()          // No other choice here.
  case P2(x)   => P2(f(x))     // Preserve information.
  case P3(x)   => P3(f(x))     // Preserve information.
}
```

To verify the given law, we first write a matrix notation for `map`:

$$\text{map}^{A,B} \stackrel{\text{def}}{=} p^{:1+A+A} \to f^{:A\to B} \to p \triangleright \left\| \begin{array}{c|ccc} & \mathbb{1} & B & B \\ \hline \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} \\ A & \mathbb{0} & f & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & f \end{array} \right| \quad .$$

The required law is written as an equation $\text{map}\,(p)\,(\text{id}) = p$, called the **identity law**. Substituting the code notation for `map`, we verify the law:

$$\begin{aligned} \text{expect to equal } p: \quad & \text{map}\,(p)\,(\text{id}) \\ \text{apply map()() to arguments}: \quad & = p \triangleright \left\| \begin{array}{ccc} \text{id} & \mathbb{0} & \mathbb{0} \\ \mathbb{0} & \text{id} & \mathbb{0} \\ \mathbb{0} & \mathbb{0} & \text{id} \end{array} \right| \\ \text{identity function in matrix notation}: \quad & = p \triangleright \text{id} \\ \triangleright\text{-notation}: \quad & = \text{id}\,(p) = p \quad . \end{aligned}$$

Example 5.4.1.10 Implement `map` and `flatMap` for `Either[L, R]`, applied to the type parameter `L`.

Solution For a type constructor, say, P^A, the standard type signatures for `map` and `flatMap` are:

$$\text{map} : P^A \to (A \to B) \to P^B \quad , \qquad \text{flatMap} : P^A \to (A \to P^B) \to P^B \quad .$$

If a type constructor has more than one type parameter, e.g., $P^{A,S,T}$, one can define the functions `map` and `flatMap` applied to a chosen type parameter. For example, when applied to the type parameter A, the type signatures are:

$$\begin{aligned} \text{map} : P^{A,S,T} &\to (A \to B) \to P^{B,S,T} \quad , \\ \text{flatMap} : P^{A,S,T} &\to (A \to P^{B,S,T}) \to P^{B,S,T} \quad . \end{aligned}$$

Being "applied to the type parameter A" means that the other type parameters S, T in $P^{A,S,T}$ remain fixed while the type parameter A is replaced by B in the type signatures of `map` and `flatMap`.

For the type `Either[L, R]` (in the type notation, $L + R$), we keep the type parameter R fixed while L is replaced by M. So we obtain the type signatures:

$$\begin{aligned} \text{map} : L + R &\to (L \to M) \to M + R \quad , \\ \text{flatMap} : L + R &\to (L \to M + R) \to M + R \quad . \end{aligned}$$

Implementing these functions is straightforward:

```
def map[L,M,R]: Either[L, R] => (L => M) => Either[M, R] = e => f => e match {
  case Left(x)    => Left(f(x))
  case Right(y)   => Right(y)
}

def flatMap[L,M,R]: Either[L, R] => (L => Either[M, R]) => Either[M, R] = e => f => e match {
  case Left(x)    => f(x)
  case Right(y)   => Right(y)
}
```

The code notation for these functions is:

$$\text{map} \stackrel{\text{def}}{=} e^{:L+R} \to f^{:L\to M} \to e \triangleright \begin{array}{c|c|cc|} & & M & R \\ \hline L & & f & \mathbb{0} \\ R & & \mathbb{0} & \text{id} \\ \hline \end{array} \quad ,$$

$$\text{flatMap} \stackrel{\text{def}}{=} e^{:L+R} \to f^{:L\to M+R} \to e \triangleright \begin{array}{c|c|c|} & & M+R \\ \hline L & & f \\ R & & y^{:R} \to \mathbb{0}^{:M} + y \\ \hline \end{array} \quad .$$

Note that we cannot split f into the M and R columns since we do not know in advance which part of the disjunctive type $M + R$ is returned by $f(x^{:L})$.

Example 5.4.1.11* Define a type constructor $\text{State}^{S,A} \equiv S \to A \times S$ and implement the functions:

(a) $\text{pure}^{S,A} : A \to \text{State}^{S,A}$.

(b) $\text{map}^{S,A,B} : \text{State}^{S,A} \to (A \to B) \to \text{State}^{S,B}$.

(c) $\text{flatMap}^{S,A,B} : \text{State}^{S,A} \to (A \to \text{State}^{S,B}) \to \text{State}^{S,B}$.

Solution It is assumed that all functions must be fully parametric and preserve as much information as possible. We define the type alias:

```
type State[S, A] = S => (A, S)
```

(a) The type signature is $A \to S \to A \times S$, and there is only one implementation:

```
def pure[S, A]: A => State[S, A] = a => s => (a, s)
```

In the code notation, this is written as:

$$\text{pu}^{S,A} \stackrel{\text{def}}{=} a^{:A} \to s^{:S} \to a \times s \quad .$$

(b) The type signature is:

$$\text{map}^{S,A,B} : (S \to A \times S) \to (A \to B) \to S \to B \times S \quad .$$

Begin writing a Scala implementation:

```
def map[S, A, B]: State[S, A] => (A => B) => State[S, B] = { t => f => s => ??? }
```

We need to compute a value of $B \times S$ from the curried arguments $t^{:S\to A\times S}$, $f^{:A\to B}$, and $s^{:S}$. We begin writing the code of `map` using a typed hole:

$$\text{map} \stackrel{\text{def}}{=} t^{:S\to A\times S} \to f^{:A\to B} \to s^{:S} \to ???^{:B} \times ???^{:S} \quad .$$

The only way of getting a value of type B is by applying f to a value of type A:

$$\text{map} \stackrel{\text{def}}{=} t^{:S\to A\times S} \to f^{:A\to B} \to s^{:S} \to f(???^{:A}) \times ???^{:S} \quad .$$

The only possibility of filling the typed hole $???^{:A}$ is to apply t to a value of type S. We already have such a value, $s^{:S}$. Computing $t(s)$ yields a pair of type $A \times S$, from which we may take the first part (of type A) to fill the typed hole $???^{:A}$. The second part of the pair is a value of type S that we may use to fill the second typed hole, $???^{:S}$. So, the Scala code is:

```
1 def map[S, A, B]: State[S, A] => (A => B) => State[S, B] = {
2   t => f => s =>
3     val (a, s2) = t(s)
4     (f(a), s2)     // We could also return ‘(f(a), s)‘ here.
5 }
```

Why not return the original value `s` in the tuple $B \times S$, instead of the new value `s2`? The reason is that we would like to preserve information as much as possible. If we return `(f(a), s)` in line 4, we will have discarded the computed value `s2`, which is a loss of information.

To write the code notation for `map`, we need to destructure the pair that $t(s)$ returns. We can write explicit destructuring code like this:

$$\text{map} \stackrel{\text{def}}{=} t^{:S\to A\times S} \to f^{:A\to B} \to s^{:S} \to (a^{:A} \times s_2^{:S} \to f(a) \times s_2)(t(s)) \quad .$$

If we temporarily denote by q the following destructuring function:

$$q \stackrel{\text{def}}{=} (a^{:A} \times s_2^{:S} \to f(a) \times s_2) \quad ,$$

we will notice that the expression $s \to q(t(s))$ is a function composition applied to s. So, we rewrite $s \to q(t(s))$ as the composition $t \fatsemi q$ and obtain shorter code:

$$\text{map} \stackrel{\text{def}}{=} t^{:S\to A\times S} \to f^{:A\to B} \to t \fatsemi (a^{:A} \times s^{:S} \to f(a) \times s) \quad .$$

Shorter formulas are often easier to reason about in derivations, although not necessarily easier to read when converted to program code.

(c) The required type signature is:

$$\text{flatMap}^{S,A,B} : (S \to A \times S) \to (A \to S \to B \times S) \to S \to B \times S \quad .$$

We perform code reasoning with typed holes:

$$\text{flatMap} \stackrel{\text{def}}{=} t^{:S\to A\times S} \to f^{:A\to S\to B\times S} \to s^{:S} \to ???^{:B\times S} \quad .$$

To fill $???^{:B\times S}$, we need to apply f to some arguments, since f is the only function that returns any values of type B. Applying f to two values will yield a value of type $B \times S$, just as we need:

$$\text{flatMap} \stackrel{\text{def}}{=} t^{:S\to A\times S} \to f^{:A\to S\to B\times S} \to s^{:S} \to f(???^{:A})(???^{:S}) \quad .$$

To fill the new typed holes, we need to apply t to an argument of type S. We have only one given value $s^{:S}$ of type S, so we must compute $t(s)$ and destructure it:

$$\text{flatMap} \stackrel{\text{def}}{=} t^{:S\to A\times S} \to f^{:A\to S\to B\times S} \to s^{:S} \to (a \times s_2 \to f(a)(s_2))\,(t(s)) \quad .$$

Translating this notation into Scala code, we obtain:

```
def flatMap[S, A, B]: State[S, A] => (A => State[S, B]) => State[S, B] = {
  t => f => s =>
    val (a, s2) = t(s)
    f(a)(s2)            // We could also return ‘f(a)(s)‘ here, but that would lose information.
}
```

In order to preserve information, we choose not to discard the computed value `s2`.

The code notation for this `flatMap` can be simplified to:

$$\text{flatMap} \stackrel{\text{def}}{=} t^{:S\to A\times S} \to f^{:A\to S\to B\times S} \to t \fatsemi (a \times s \to f(a)(s)) \quad .$$

5.4.2 Exercises

Exercise 5.4.2.1 Find the cardinality of the type `P = Option[Boolean => Option[Boolean]]`. Show that `P` is equivalent to `Option[Boolean] => Boolean`, and that the equivalence is accidental and not “natural”.

Exercise 5.4.2.2 Verify the type equivalences $A + A \cong \mathbb{2} \times A$ and $A \times A \cong \mathbb{2} \to A$, where $\mathbb{2}$ denotes the `Boolean` type.

Exercise 5.4.2.3 Show that $A \Rightarrow (B \vee C) \neq (A \Rightarrow B) \wedge (A \Rightarrow C)$ in constructive and Boolean logic.

Exercise 5.4.2.4 Verify the type equivalence $(A \to B \times C) \cong (A \to B) \times (A \to C)$ with full proofs.

Exercise 5.4.2.5 Use known rules to verify the type equivalences without need for proofs:

(a) $(A + B) \times (A \to B) \cong A \times (A \to B) + (\mathbb{1} + A \to B)$.

(b) $(A \times (\mathbb{1} + A) \to B) \cong (A \to B) \times (A \to A \to B)$.

(c) $A \to (\mathbb{1} + B) \to C \times D \cong (A \to C) \times (A \to D) \times (A \times B \to C) \times (A \times B \to D)$.

Exercise 5.4.2.6 Write the type notation for `Either[(A, Int), Either[(A, Char), (A, Float)]]`. Transform this type into an equivalent type of the form $A \times (...)$.

Exercise 5.4.2.7 Define a type $\text{OptE}^{T,A} \stackrel{\text{def}}{=} \mathbb{1} + T + A$ and implement information-preserving `map` and `flatMap` for it, applied to the type parameter A. Get the same result using the equivalent type $(\mathbb{1} + A) + T$, i.e., `Either[Option[A], T]`. The required type signatures are:

$$\text{map}^{A,B,T} : \text{OptE}^{T,A} \to (A \to B) \to \text{OptE}^{T,B} \quad ,$$
$$\text{flatMap}^{A,B,T} : \text{OptE}^{T,A} \to (A \to \text{OptE}^{T,B}) \to \text{OptE}^{T,B} \quad .$$

Exercise 5.4.2.8 Implement the `map` function for the type constructor `P[A]` from Example 5.4.1.2. The required type signature is $P^A \to (A \to B) \to P^B$. Preserve information as much as possible.

Exercise 5.4.2.9 For the type constructor $Q^{T,A}$ defined in Exercise 5.1.4.1, define the `map` function, preserving information as much as possible:

$$\text{map}^{T,A,B} : Q^{T,A} \to (A \to B) \to Q^{T,B} \quad .$$

Exercise 5.4.2.10 Define a recursive type constructor Tr_3 as $\text{Tr}_3{}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A \times A \times \text{Tr}_3{}^A$ and implement the `map` function for it, with the standard type signature: $\text{map}^{A,B} : \text{Tr}_3{}^A \to (A \to B) \to \text{Tr}_3{}^B$.

Exercise 5.4.2.11 Implement fully parametric, information-preserving functions with the types:

(a) $Z + A \times A \to (A \to B) \to Z + B \times B$.

(b) $A + Z \to B + Z \to (A \to B \to C) \to C + Z$.

(c) $P + A \times A \to (A \to B) \to (P \to A + Q) \to Q + B \times B$.

(d) $\text{flatMap}^{E,A,B} : \text{Read}^{E,A} \to (A \to \text{Read}^{E,B}) \to \text{Read}^{E,B}$.

(e) $\text{State}^{S,A} \to (S \times A \to B) \to \text{State}^{S,B}$.

Exercise 5.4.2.12* Denote $\text{Cont}^{R,T} \stackrel{\text{def}}{=} (T \to R) \to R$ and implement the functions:

(a) $\text{map}^{R,T,U} : \text{Cont}^{R,T} \to (T \to U) \to \text{Cont}^{R,U}$.

(b) $\text{flatMap}^{R,T,U} : \text{Cont}^{R,T} \to (T \to \text{Cont}^{R,U}) \to \text{Cont}^{R,U}$.

Exercise 5.4.2.13* Denote $\text{Sel}^{Z,T} \stackrel{\text{def}}{=} (T \to Z) \to T$ and implement the functions:

(a) $\text{map}^{Z,A,B} : \text{Sel}^{Z,A} \to (A \to B) \to \text{Sel}^{Z,B}$.

(b) $\text{flatMap}^{Z,A,B} : \text{Sel}^{Z,A} \to (A \to \text{Sel}^{Z,B}) \to \text{Sel}^{Z,B}$.

5.5 Discussion and further developments

5.5.1 Using the Curry-Howard correspondence for writing code

The CH correspondence is used in two practically important reasoning tasks: checking whether a type signature can be implemented as a fully parametric function, and determining whether two types are equivalent. For the first task, we map type expressions into formulas in the constructive logic and apply the proof rules of that logic. For the second task, we map type expressions into *arithmetic* formulas and apply the ordinary rules of arithmetic.

Although tools such as the `curryhoward` library can sometimes derive code from types, it is beneficial if a programmer is able to derive an implementation by hand or to determine that an implementation is impossible. For instance, the programmer should recognize that the type signature:

```
def f[A, B]: A => (A => B) => B
```

has only one fully parametric implementation, while the following two type signatures have none:

```
def g[A, B]: A => (B => A) => B
def h[A, B]: ((A => B) => A) => A
```

Exercises in this chapter help to build up the required technique and intuition. The two main guidelines for code derivation are: "values of parametric types cannot be constructed from scratch" and "one must hard-code the decision to return a chosen part of a disjunctive type when no other disjunctive value is given". These guidelines can be justified by referring to the rigorous rules of proof (Table 5.2). Sequents producing a value of type A can be proved only if there is a premise containing A or a function that returns a value of type A.[9] One can derive a disjunction without hard-coding only if one already has a disjunction in the premises (and then the rule "use `Either`" could apply).

Throughout this chapter, we require all code to be fully parametric. This is because the CH correspondence gives useful, non-trivial results only for parameterized types and fully parametric code. For concrete, non-parameterized types (`Int`, `String`, etc.), one can always produce *some* values even with no previous data. So, the propositions $\mathcal{CH}(\text{Int})$ or $\mathcal{CH}(\text{String})$ are always true.

Consider the function `(x:Int) => x + 1`. Its type signature, `Int => Int`, may be implemented by many other functions, such as `x => x - 1`, `x => x * 2`, etc. So, the type signature `Int => Int` is insufficient to specify the code of the function, and deriving code from that type is not a meaningful task. Only a fully parametric type signature, such as $A \to (A \to B) \to B$, could give enough information for deriving the function's code. Additionally, we must require the code of functions to be fully parametric. Otherwise we will be unable to reason about code derivation from type signatures.

Validity of a $\mathcal{CH}$-proposition $\mathcal{CH}(T)$ means that we can implement *some* value of the given type T. But this does not give any information about the properties of that value, such as whether it satisfies any laws. This is why type equivalence (which requires the laws of isomorphisms) is not determined by an equivalence of logical formulas.

It is useful for programmers to be able to transform type expressions to equivalent simpler types before starting to write code. The type notation introduced in this book is designed to help programmers to recognize patterns in type expressions and to reason about them more easily. We have shown that a type equivalence corresponds to *each* standard arithmetic identity such as $(a+b)+c = a+(b+c)$, $(a\times b)\times c = a\times(b\times c)$, $1\times a = a$, $(a+b)\times c = a\times c + b\times c$, and so on. Because of this, we are allowed to transform and simplify types as if they were arithmetic expressions, e.g., to rewrite:

$$\mathbb{1}\times(A+B)\times C + D \cong D + A\times C + B\times C \quad .$$

The type notation makes this reasoning more intuitive (for people familiar with arithmetic).

These results apply to all type expressions built up using product types, disjunctive types (also called "sum" types because they correspond to arithmetic sums), and function types (also called "exponential" types because they correspond to arithmetic exponentials). Type expressions that contain only products and sum types are called **polynomial**.[10] Type expressions that also contain function types may be called **exponential-polynomial**. This book focuses on exponential-polynomial types because they are sufficient for almost all design patterns used in functional programming.

There are no types corresponding to subtraction or division, so arithmetic equations such as:

$$(1-t)\times(1+t) = 1 - t\times t \quad , \quad \text{and} \quad \frac{t + t\times t}{t} = 1 + t \quad ,$$

do not directly yield any type equivalences. However, consider this well-known formula:

$$\frac{1}{1-t} = 1 + t + t^2 + t^3 + \ldots + t^n + \ldots \quad .$$

[9] This is proved rigorously in the paper https://research-repository.st-andrews.ac.uk/handle/10023/8824 (2016) by R. Dyckhoff. See the "Theorem" in section 6 ("Goal-directed pruning").

[10] These types are often called "algebraic data types" but this book prefers the more precise term "polynomial types".

At first sight, this formula appears to involve subtraction, division, and an infinite series, and so cannot be directly translated into a type equivalence. However, the formula can be rewritten as:

$$\frac{1}{1-t} = L(t) \quad \text{where} \quad L(t) \stackrel{\text{def}}{=} 1 + t + t^2 + t^3 + \ldots + t^n \times L(t) \quad . \tag{5.26}$$

The definition of $L(t)$ is finite and only contains additions and multiplications. So, Eq. (5.26) can be translated into a type equivalence:

$$L^A \cong 1 + A + A \times A + A \times A \times A + \ldots + \underbrace{A \times \ldots \times A}_{n \text{ times}} \times L^A \quad . \tag{5.27}$$

This type formula (with $n = 1$) is equivalent to a recursive definition of the type constructor `List`:

$$\text{List}^A \stackrel{\text{def}}{=} 1 + A \times \text{List}^A \quad .$$

The type equivalence (5.27) suggests that we may view the recursive type `List` heuristically as an "infinite disjunction" describing lists of zero, one, two, etc., elements.

5.5.2 Implications for designing new programming languages

Today's functional programming practice assumes, at the minimum, that programmers will use the six standard type constructions (Section 5.1.2) and the eight standard code constructions (Section 5.2.2). These constructions are foundational in the sense that they are used to express all design patterns of functional programming. A language that does not directly support some of these constructions cannot be considered a functional programming language.

A remarkable result of the CH correspondence is that the type system of any given programming language (functional or not) is mapped into a *certain logic*, i.e., a system of logical operations and proof rules. A logical operation will correspond to each of the *type* constructions available in the programming language. A proof rule will correspond to each of the available *code* constructions. Programming languages that support all the standard type and code constructions — for instance, OCaml, Haskell, F#, Scala, Swift, Rust, — are mapped into the constructive logic with all standard logical operations available ($True$, $False$, disjunction, conjunction, and implication).

Languages such as C, C++, Java, C#, Go are mapped into logics that do not have the disjunction operation or the constants $True$ and $False$. In other words, these languages are mapped into *incomplete* logics where some true formulas cannot be proved. Incompleteness of the logic of types will make a programming language unable to express certain computations, e.g., directly handle data that belongs to a disjoint domain.

Languages that do not enforce type checking (e.g., Python or JavaScript) are mapped to inconsistent logics where any proposition can be proved — even propositions normally considered $False$. The CH correspondence will map such absurd proofs to code that *appears* to compute a certain value (since the $\mathcal{CH}$-proposition was proved to be $True$) although that value is not actually available. In practice, such code will crash because of a value that has a wrong type, is "null", or is a pointer to an invalid memory location. Those errors will not happen in a programming language whose logic of types is consistent and whose compiler checks all types at compile time.

So, the CH correspondence gives a mathematically justified procedure for designing new programming languages. The procedure has the following steps:

- Choose a formal logic that is complete and free of inconsistencies.
- For each logical operation, provide a type construction in the language.
- For each axiom and proof rule of the logic, provide a code construction in the language.

Mathematicians have studied different logics, such as modal logic, temporal logic, or linear logic. Compared with the constructive logic, those other logics have some additional type operations. For instance, modal logic adds the operations "necessarily" and "possibly", and temporal logic adds the operation "until". For each logic, mathematicians have determined the minimal complete sets of operations, axioms, and proof rules that do not lead to inconsistency. Programming language designers can use this mathematical knowledge by choosing a logic and translating it into a minimal "core" of a programming language. Code in that language will be guaranteed *never to crash* as long as all types match. This mathematical guarantee (known as **type safety**) is a powerful help for programmers since it automatically prevents a large number of coding errors. So, programmers will benefit if they use languages designed using the CH correspondence.

Practically useful programming languages will of course need more features than the minimal set of mathematically necessary features derived from a chosen logic. Language designers need to make sure that all added features are consistent with the core language.

At present, it is still not fully understood how a practical programming language could use, say, modal or linear logic as its logic of types. Experience suggests that, at least, the operations of the plain constructive logic should be available. So, it appears that the six type constructions and the eight code constructions will remain available in all future languages of functional programming.

It is possible to apply the FP paradigm while writing code in any programming language. However, some languages lack certain features that make FP techniques easier to use in practice. For example, in a language such as JavaScript, Python, or Ruby, one can productively use the map/reduce operations but not disjunctive types. More advanced FP constructions (such as typeclasses) are impractical in these languages because the required code becomes too hard to read and to write without errors, which negates the advantages of rigorous reasoning about functional programs.

Some programming languages, such as Haskell and OCaml, were designed specifically for advanced use in the FP paradigm. Other languages, such as F#, Scala, Swift, PureScript, Elm, and Rust, have different design goals but still support enough FP features to be considered FP languages. This book uses Scala, but the same constructions may be implemented in other FP languages in a similar way. At the level of detail needed in this book, the differences between languages such as OCaml, Haskell, F#, Scala, Swift, PureScript, Elm, and Rust, do not play a significant role.

5.5.3 Uses of the void type (Scala's `Nothing`)

The void type[11] (Scala's `Nothing`) corresponds to the logical constant $False$. (The proposition "*a value of the void type can be computed*" is always false.) There are few practical uses of the void type. One use case is for a branch of a `match/case` expression that throws an exception instead of returning a value. In this sense, returning a value of the void type corresponds to a crash in the program. So, a `throw` expression is defined as if it returns a value of type `Nothing`. We can then pretend to convert that "value" (which will never be actually returned) into a value of any other type. Example 5.3.4.2 shows how to write a function `absurd[A]` of type `Nothing => A`.

To see how this trick is used, consider this code defining a value `x`:

```
val x: Double = if (t >= 0.0) math.sqrt(t) else throw new Exception("error")
```

The `else` branch does not return a value, but `x` is declared to have type `Double`. For this code to typecheck, both branches must return values of the same type. So, the compiler needs to pretend that the `else` branch also returns a value of type `Double`. The compiler first assigns the type `Nothing` to the expression `throw ...` and then automatically uses the conversion `Nothing => Double` to convert that type to `Double`. In this way, types will match in the definition of the value `x`.

This book does not discuss exceptions in much detail. The functional programming paradigm does not use exceptions because their presence significantly complicates reasoning about code.

As another example of using the void type, suppose an external library implements a function:

[11] The "void" type as defined in this book is a type with no values. It is *not* the same as the `void` keyword in Java or C that denotes functions returning "nothing". Those functions are equivalent to Scala functions returning the `Unit` type.

Constructive logic	Boolean logic
$\frac{}{\Gamma \vdash \mathcal{CH}(\mathbb{1})}$ (create unit)	$\neg\Gamma \vee True = True$
$\frac{}{\Gamma, \alpha \vdash \alpha}$ (use arg)	$\neg\Gamma \vee \neg\alpha \vee \alpha = True$
$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \Rightarrow \beta}$ (create function)	$(\neg\Gamma \vee \neg\alpha \vee \beta) = (\neg\Gamma \vee (\alpha \Rightarrow \beta))$
$\frac{\Gamma \vdash \alpha \qquad \Gamma \vdash \alpha \Rightarrow \beta}{\Gamma \vdash \beta}$ (use function)	$((\neg\Gamma \vee \alpha) \wedge (\neg\Gamma \vee (\alpha \Rightarrow \beta))) \Rightarrow (\neg\Gamma \vee \beta)$
$\frac{\Gamma \vdash \alpha \qquad \Gamma \vdash \beta}{\Gamma \vdash \alpha \wedge \beta}$ (create tuple)	$(\neg\Gamma \vee \alpha) \wedge (\neg\Gamma \vee \beta) = (\neg\Gamma \vee (\alpha \wedge \beta))$
$\frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \alpha}$ (use tuple-1)	$(\neg\Gamma \vee (\alpha \wedge \beta)) \Rightarrow (\neg\Gamma \vee \alpha)$
$\frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \vee \beta}$ (create Left)	$(\neg\Gamma \vee \alpha) \Rightarrow (\neg\Gamma \vee (\alpha \vee \beta))$
$\frac{\Gamma \vdash \alpha \vee \beta \qquad \Gamma, \alpha \vdash \gamma \qquad \Gamma, \beta \vdash \gamma}{\Gamma \vdash \gamma}$ (use Either)	$((\neg\Gamma \vee \alpha \vee \beta) \wedge (\neg\Gamma \vee \neg\alpha \vee \gamma) \wedge (\neg\Gamma \vee \neg\beta \vee \gamma)) \Rightarrow (\neg\Gamma \vee \gamma)$

Table 5.7: Proof rules of constructive logic are true also in the Boolean logic.

```
def parallel_run[E, A, B](f: A => Either[E, B]): Either[E, B] = ???
```

We may imagine that `parallel_run(f)` performs some parallel computations using a given function f. In general, functions $f^{:A \to E+B}$ may return an error of type E or a result of type B. Suppose we know that a particular function f never fails to compute its result. To express that knowledge in code, we may explicitly set the type parameter E to the void type `Nothing` when applying `parallel_run`:

```
parallel_run[Nothing, A, B](f)  // Types match only when values f(a) always are of the form Right(b).
```

Returning an error is now impossible (the type `Nothing` has no values). If the function `parallel_run` is fully parametric, it will work in the same way with all types E, including $E = \mathbb{0}$. The code implements our intention via type parameters, giving a compile-time guarantee of correct results.

5.5.4 Relationship between Boolean logic and constructive logic

We have seen that some true theorems of Boolean logic are not true in constructive logic. For example, the Boolean identities $\neg(\neg\alpha) = \alpha$ and $(\alpha \Rightarrow \beta) = (\neg\alpha \vee \beta)$ do not hold in the constructive logic. However, as we will now show, any theorem of constructive logic is also a theorem of Boolean logic. The reason is that all eight rules of constructive logic (Section 5.2.2) are also true in Boolean logic.

To verify that a formula is true in Boolean logic, it is sufficient to check that the value of the formula is $True$ for all possible truth values ($True$ or $False$) of its variables. A sequent such as $\alpha, \beta \vdash \gamma$ is true in Boolean logic if and only if $\gamma = True$ under the assumption that $\alpha = \beta = True$. So, the sequent $\alpha, \beta \vdash \gamma$ is translated into the Boolean formula:

$$\alpha, \beta \vdash \gamma = ((\alpha \wedge \beta) \Rightarrow \gamma) = (\neg\alpha \vee \neg\beta \vee \gamma) \quad .$$

Table 5.7 translates all proof rules of Section 5.2.2 into Boolean formulas. The first two lines are axioms, while the subsequent lines are Boolean theorems that can be verified by calculation.

To simplify the calculations, note that all terms in the formulas contain the operation $(\neg\Gamma \vee ...)$ corresponding to the context Γ. Now, if Γ is $False$, the entire formula becomes automatically $True$, and there is nothing else to check. So, it remains to verify the formula in case $\Gamma = True$, and then we can simply omit all instances of $\neg\Gamma$ in the formulas. Let us show the Boolean derivations for the rules "use function" and "use Either"; other formulas are checked in a similar way:

$$\begin{aligned}
\text{formula "use function"}: &\quad (\alpha \wedge (\alpha \Rightarrow \beta)) \Rightarrow \beta \\
\text{use Eq. (5.8)}: &\quad = \underline{\neg}(\alpha \underline{\wedge} (\neg\alpha \underline{\vee} \beta)) \vee \beta \\
\text{de Morgan's laws}: &\quad = \underline{\neg\alpha \vee (\alpha \wedge \neg\beta)} \vee \beta
\end{aligned}$$

$$\begin{aligned}
\text{identity } p \vee (\neg p \wedge q) = p \vee q \text{ with } p = \neg\alpha \text{ and } q = \beta : \quad & = \neg\alpha \vee \underline{\neg\beta \vee \beta} \\
\text{axiom “use arg”} : \quad & = True \quad .
\end{aligned}$$

$$\begin{aligned}
\text{formula “use Either”} : \quad & ((\alpha \vee \beta) \wedge (\alpha \Rightarrow \gamma) \wedge (\beta \Rightarrow \gamma)) \Rightarrow \gamma \\
\text{use Eq. (5.8)} : \quad & = \neg\left((\alpha \vee \beta) \wedge (\neg\alpha \vee \gamma) \wedge (\neg\beta \vee \gamma)\right) \vee \gamma \\
\text{de Morgan's laws} : \quad & = (\neg\alpha \wedge \neg\beta) \vee \underline{(\alpha \wedge \neg\gamma)} \vee \underline{(\beta \wedge \neg\gamma)} \vee \gamma \\
\text{identity } p \vee (\neg p \wedge q) = p \vee q : \quad & = \underline{(\neg\alpha \wedge \neg\beta) \vee \alpha} \vee \beta \vee \gamma \\
\text{identity } p \vee (\neg p \wedge q) = p \vee q : \quad & = \underline{\neg\alpha \vee \alpha} \vee \beta \vee \gamma \\
\text{axiom “use arg”} : \quad & = True \quad .
\end{aligned}$$

Since each proof rule of the constructive logic is translated into a true formula in Boolean logic, it follows that a proof tree in the constructive logic will be translated into a tree of Boolean formulas that have value $True$ for each axiom or proof rule. The result is that any constructive proof for a sequent such as $\emptyset \vdash f(\alpha, \beta, \gamma)$ is translated into a chain of Boolean implications that look like this:

$$True = (...) \Rightarrow (...) \Rightarrow ... \Rightarrow f(\alpha, \beta, \gamma) \quad .$$

Since $(True \Rightarrow \alpha) = \alpha$, this chain proves the Boolean formula $f(\alpha, \beta, \gamma)$.

For example, the proof tree shown in Figure 5.1 is translated into:

$$\begin{aligned}
\text{axiom “use arg”} : \quad & True = (\neg((\alpha \Rightarrow \alpha) \Rightarrow \beta) \vee \neg\alpha \vee \alpha) \\
\text{rule “create function”} : \quad & \quad \Rightarrow \neg((\alpha \Rightarrow \alpha) \Rightarrow \beta) \vee (\alpha \Rightarrow \alpha) \quad . \\
\text{axiom “use arg”} : \quad & True = \neg((\alpha \Rightarrow \alpha) \Rightarrow \beta) \vee ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \quad . \\
\text{rule “use function”} : \quad & True \Rightarrow (\neg((\alpha \Rightarrow \alpha) \Rightarrow \beta) \vee \beta) \\
\text{rule “create function”} : \quad & \quad \Rightarrow (((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta) \quad .
\end{aligned}$$

It is easier to check Boolean truth tables than to find a proof tree in constructive logic (or to establish that no proof tree exists). If we find that a formula is *not* true in Boolean logic, we know it is also not true in constructive logic. This gives us a quick way of proving that some type signatures are *not* implementable as fully parametric functions. However, if a formula is true in Boolean logic, it does not follow that the formula is also true in the constructive logic.

In addition to formulas shown in Table 5.4 (Section 5.2.1), here are three more examples of formulas that are *not* true in Boolean logic:

$$\forall\alpha.\,\alpha \quad , \qquad \forall(\alpha, \beta).\,\alpha \Rightarrow \beta \quad , \qquad \forall(\alpha, \beta).\,(\alpha \Rightarrow \beta) \Rightarrow \beta \quad .$$

These formulas are also *not* true in the constructive logic.

5.5.5 The constructive logic and the law of excluded middle

Computations in the Boolean logic are often performed using truth tables. It is perhaps surprising that the proof rules of the constructive logic are *not* equivalent to checking whether some propositions are $True$ or $False$ via a truth table. A general form of this statement was proved by K. Gödel in 1932.[12] In this sense, constructive logic does *not* imply that every proposition is either $True$ or $False$. This is not intuitive and requires getting used to. Reasoning in the constructive logic must use the axioms and derivation rules directly, instead of truth tables.

The reason Boolean logic can use truth tables is that every Boolean proposition is either $True$ or $False$. This can be written as the formula $\forall\alpha.\,(\neg\alpha \vee \alpha = True)$. Table 5.7 uses the Boolean identity $(\alpha \Rightarrow \beta) = (\neg\alpha \vee \beta)$, which does not hold in the constructive logic, to translate the constructive axiom

[12] See plato.stanford.edu/entries/intuitionistic-logic-development/

"use arg" into the Boolean axiom $\neg\alpha \vee \alpha = True$. The formula $\forall\alpha.\,\neg\alpha \vee \alpha = True$ is known as the **law of excluded middle**.[13] It is remarkable that the constructive logic *does not have* the law of excluded middle. It is neither an axiom nor a derived theorem in constructive logic.

To see why, consider how the constructive logic could possibly have $\forall\alpha.\,\neg\alpha \vee \alpha = True$. The negation operation ($\neg\alpha$) is defined as the implication $\alpha \Rightarrow False$. So, the formula $\forall\alpha.\,\neg\alpha \vee \alpha$ corresponds to the type $\forall A.\,(A \to \mathbb{0}) + A$. Can we compute a value of this type via fully parametric code? For that, we need to compute either a value of type $A \to \mathbb{0}$ or a value of type A. This decision needs to be made in advance independently of A, because the code of a fully parametric function must operate in the same way for all types. As we have seen in Example 5.3.4.3, a value of type $A \to \mathbb{0}$ exists if the type A is itself $\mathbb{0}$. But we do not know in advance whether $A = \mathbb{0}$. Since there are no values of type $\mathbb{0}$, and the type parameter A could be, say, `Int`, we cannot compute a value of type $A \to \mathbb{0}$.

Why is it impossible to implement a value of the type $(A \to \mathbb{0}) + A$? Surely, the type A is either void or not void. If A is void then $(A \to \mathbb{0}) \cong \mathbb{1}$ is not void (as Example 5.3.4.3 shows). So, one of the types in the disjunction $(A \to \mathbb{0}) + A$ should be non-void and have values that we can compute.

While this argument is true, it does not help implementing a value of type $(A \to \mathbb{0}) + A$ via fully parametric code. It is not enough to know that one of the two values "should exist". We need to know *which* of the two values exists, and we need to write code that computes that value. That code may not decide what to do depending on whether the type A is void, because the code must work in the same way for all types A (void or not). As we have seen, that code is impossible to write.

In Boolean logic, one may prove that a value "should exist" by showing that the non-existence of a value is contradictory in some way. However, any practically useful program needs to "construct" (i.e., to compute) actual values. The "constructive" logic got its name from this requirement. So, it is the constructive logic (not the Boolean logic) that provides correct reasoning about the types of values computable by fully parametric functional programs.

If we drop the requirement of full parametricity, we *could* implement the law of excluded middle. Special features of Scala (reflection, type tags, and type casts) allow programmers to compare types as values and to determine what type was given to a type parameter when a function is applied:

```
import scala.reflect.runtime.universe._
def getType[T: TypeTag]: Type = weakTypeOf[T]   // Convert the type parameter T into a special value.
def equalTypes[A: TypeTag, B: TypeTag]: Boolean = getType[A] =:= getType[B] // Compare types A and B.

def excludedMiddle[A: TypeTag]: Either[A, A => Nothing] =  // excludedMiddle has type ∀A.(A → 0) + A.
   if (equalTypes[A, Nothing]) Right((identity _).asInstanceOf[A => Nothing])    // Return id : 0 → 0.
   else if (equalTypes[A, Int]) Left(123.asInstanceOf[A])          // Produce some value of type Int.
   else if (equalTypes[A, Boolean]) Left(true.asInstanceOf[A]) // Produce some value of type Boolean.
   else ???                  // Need to write many more definitions to support all other Scala types.

scala> excludedMiddle[Int]
res0: Either[Int,Int => Nothing] = Left(123)

scala> excludedMiddle[Nothing]
res1: Either[Nothing,Nothing => Nothing] = Right(<function1>)
```

In this code, we check whether $A = \mathbb{0}$. If so, we can implement $A \to \mathbb{0}$ as an identity function of type $\mathbb{0} \to \mathbb{0}$. Otherwise, we know that A is one of the existing Scala types (`Int`, `Boolean`, etc.), which are not void and have values that we can simply write down one by one in the subsequent code.

Explicit **type casts**, such as `123.asInstanceOf[A]`, are needed because the Scala compiler cannot know that `A` is `Int` in the scope where we return `Left(123)`. Without a type cast, the compiler will not accept `123` as a value of type `A` in that scope.

The method `asInstanceOf` is dangerous because the code `x.asInstanceOf[T]` disables the type checking for the value `x`. This tells the Scala compiler to believe that `x` has type `T` even when the type `T` is inconsistent with the actually given code of `x`. The resulting programs compile but may give unexpected results or crash. These errors would have been prevented if we did not disable the type checking. In this book, we will avoid writing such code whenever possible.

[13] https://en.wikipedia.org/wiki/Law_of_excluded_middle

6 Functors and contrafunctors

Type constructors such as `Seq[A]` or `Array[A]` are data structures that hold or "wrap" zero or more values of a given type `A`. These data structures are fully parametric: they work in the same way for every type `A`. Working with parametric "data wrappers" or "data containers" turns out to be a powerful design pattern of functional programming. To realize all its benefits, we will formalize the concept of data wrapping through a set of mathematical laws. We will then extend that design pattern to all data types for which the laws hold.

6.1 Practical use

6.1.1 Motivation. Type constructors that wrap data

How to formalize the idea of wrapped data? An intuitive view is that the data is "still there", i.e., we should be able to manipulate the data held within the wrapper. In functional programming, to manipulate means to apply functions to data. So, if an integer value 123 is "wrapped", we should be able somehow to apply a function such as `{x => x * 2}` and obtain a "wrapped" value 246.

Let us look at some often used type constructors defined in the Scala standard library, such as `Seq[A]`, `Try[A]`, and `Future[A]`. We notice the common features:

- There are some methods for creating a data structure that wraps zero or more values of a given type. For example, the Scala code `List.fill(10)(0)` creates a list of ten zeros of type `Int`.
- There are some methods for reading the wrapped values, if they exist. For example, the `List` class has the method `headOption` that returns a non-empty `Option` when the first element exists.
- There are some methods for manipulating the wrapped values while *keeping* them wrapped. For example, `List(10, 20, 30).map(_ + 5)` evaluates to `List(15, 25, 35)`.

The data types `Seq[A]`, `Try[A]`, and `Future[A]` express quite different kinds of wrapping. The data structure implementing `Seq[A]` can hold a variable number of values of type `A`. The data structure `Try[A]` holds either a successfully computed value of type `A` or a failure. The data structure `Future[A]` implements a computation that has been scheduled to run but may not have finished yet, and may compute a value of type `A` (or fail) at a later time.

Since the "wrappers" `Seq`, `Try`, and `Future` are so different, the methods for creating and reading the wrapped values have different type signatures for each wrapper. However, the method `map` is similar in all three examples. We can say generally that the `map` method will apply a given function `f: A => B` to data of type `A` stored inside the wrapper, such that new data (of type `B`) will remain within a wrapper of the same type:

```
val a = List(x,y,z).map(f) // Result is List(f(x), f(y), f(z)).
val b = Try(x).map(f)      // Result is Try(f(x)).
val c = Future(x).map(f)   // Result is Future(f(x)).
```

This motivates us to use the `map` method as a requirement for the wrapping functionality. We will consider a type constructor `F[A]` a "wrapper" if there exists a function `map` with the type signature:

```
def map[A, B]: F[A] => (A => B) => F[B]
```

We can see that `Seq`, `Try`, and `Future` are "wrappers" because they have a suitable `map` method. This chapter focuses on the properties of `map` that are common to *all* wrapper types. We will ignore all

other features — reading data out of the wrapper, inserting or deleting data, waiting until data becomes available, etc., — as they are implemented by methods specific to each wrapper type.

6.1.2 Extended example: `Option` and the identity law

As another example of a "data wrapper", consider the type constructor `Option[A]`, which is written in the type notation as:

$$\text{Opt}^A \stackrel{\text{def}}{=} \mathbb{1} + A \quad .$$

The type signature of its `map` function is:

$$\text{map}^{A,B} : \mathbb{1} + A \to (A \to B) \to \mathbb{1} + B \quad .$$

This function produces a new `Option[B]` value that wraps transformed data. We will now use this example to develop intuition about manipulating data in a wrapper.

Two possible implementations of `map` fit the type signature:

```
def mapX[A, B](oa: Option[A])(f: A => B): Option[B] = None

def mapY[A, B](oa: Option[A])(f: A => B): Option[B] =
  oa match {
    case None       => None
    case Some(x)    => Some(f(x))
  }
```

The code of `mapX` loses information since it always returns `None` and ignores all input. The implementation `mapY` is more useful since it preserves information.

How can we formulate this property of `mapY` in a rigorous way? The trick is to choose the argument $f^{:A\to B}$ in the expression `map(oa)(f)` to be the identity function $\text{id}^{:A\to A}$ (setting `map`'s type parameters as $A = B$, so that the types match). Applying an identity function to a value wrapped in an `Option[A]` should not change that value. To verify that, substitute the identity function instead of `f` into `mapY` and compute:

```
mapY[A, A](x: Option[A])(identity[A]: A => A): Option[A]
  == x match {
        case None       => None         // No change.
        case Some(x)    => Some(x)      // No change.
     } == x
```

The result is always equal to `x`. We can write that fact as an equation:

$$\forall x^{:\text{Opt}^A} .\ \text{map}\,(x)(\text{id}) = x \quad .$$

This equation is called the **identity law** of `map`. The identity law is a formal way of expressing the information-preserving property of the `map` function. The implementation `mapX` violates the identity law since it always returns `None` and so `mapX(oa)(id) == None` and not equal to `oa` for arbitrary values of `oa`. A data wrapper should not unexpectedly lose information when we manipulate the wrapped data. So, the correct implementation of `map` is `mapY`. The code notation for `map` is:

$$\text{map}^{A,B} \stackrel{\text{def}}{=} p^{:\mathbb{1}+A} \to f^{:A\to B} \to p \triangleright \begin{array}{|c||cc|} & \mathbb{1} & B \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & f \end{array} \quad .$$

When writing code, it is convenient to use the `map` method defined in the Scala library. However, when reasoning about the properties of `map`, it turns out to be more convenient to flip the order of the curried arguments and to use the equivalent function, called `fmap`, with the type signature:

$$\text{fmap}^{A,B} : (A \to B) \to \mathbb{1} + A \to \mathbb{1} + B \quad .$$

The Scala implementation and the code notation for `fmap` are shorter than those for `map`:

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  case None       => None
  case Some(x)    => Some(f(x))
}
```

$$\text{fmap}\,(f^{:A\to B}) \stackrel{\text{def}}{=} \begin{array}{|c||cc|} & \mathbb{1} & B \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & f \end{array} \quad . \tag{6.1}$$

The identity law also looks simpler if expressed in terms of `fmap`, namely fmap (id) = id. In writing fmap (id) = id, we omitted the type parameters A and B, which must be both equal ($A = B$).

Note that the type signature of `fmap` looks like a transformation from functions of type `A => B` to functions of type `Option[A] => Option[B]`. This transformation is called **lifting** because it "lifts" a function $f^{:A\to B}$ operating on simple values into a function operating on `Option`-wrapped values.

So, the identity law may be stated by saying that "a lifted identity function is also an identity function". If we apply a lifted identity function to a data wrapper, we expect the wrapped data to remain unchanged. The identity law expresses this expectation as a mathematical equation.

6.1.3 Motivation for the composition law

The main feature of a "data wrapper" is to allow us to manipulate the data inside it by applying functions to that data. The corresponding Scala code is `p.map(f)`, where `p` is a value of a wrapper type. It is natural to expect that lifted functions behave in the same way as the "unlifted" ones. For example, suppose we need to increment a counter `c` of type `Option[Int]`. The `Option` type means that the counter may be empty or non-empty; if it is non-empty, we increment the integer value wrapped inside the `Option` using the incrementing function:

$$\text{incr} \stackrel{\text{def}}{=} x^{:\text{Int}} \to x + 1 \quad .$$

In order to apply a function to the counter `c`, we need to lift that function. The Scala code is:

```
def incr: Int => Int = x => x + 1
val c: Option[Int] = Some(0)

scala> c.map(incr)
res0: Option[Int] = Some(1)
```

If we apply the lifted function twice, we expect that the counter will be incremented twice:

```
scala> c.map(incr).map(incr)
res1: Option[Int] = Some(2)
```

This result is the same as when applying a lifted function $x \to x + 2$:

```
scala> c.map(x => x + 2)
res2: Option[Int] = Some(2)
```

It would be confusing and counter-intuitive if `c.map(x => x + 2)` did not give the same result as `c.map(incr).map(incr)`.

We can formulate this property more generally: liftings should preserve the function composition for arbitrary functions $f^{:A\to B}$ and $g^{:B\to C}$. This is written as:

```
c.map(f).map(g) == c.map(f andThen g) == c.map(x => g(f(x)))
```

$$c^{:F^A} \triangleright \text{fmap}\,(f^{:A\to B}) \triangleright \text{fmap}\,(g^{:B\to C}) = c \triangleright \text{fmap}\,(f) \,⨾\, \text{fmap}\,(g) = c \triangleright \text{fmap}\,(f^{:A\to B} \,⨾\, g^{:B\to C}) \quad .$$

This equation has the form $c \triangleright p = c \triangleright q$ with some functions p and q, or equivalently $p(c) = q(c)$. When we have $p(c) = q(c)$ for all c, it means the functions themselves are equal: $p = q$. So, we may omit the argument c and rewrite the equation in a shorter form:

$$\text{fmap}\,(f) \,⨾\, \text{fmap}\,(g) = \text{fmap}\,(f \,⨾\, g) \quad .$$

This equation is called the **composition law** of functors.

Let us verify the composition law of the `Option` type, whose `fmap` function was shown in Section 6.1.2. For clarity and practice, we will perform the derivations both in the code notation and in the Scala syntax. To evaluate fmap $(f \,⨾\, g)$, we apply `fmap(f andThen g)`, where `f: A => B` and `g: B => C` are arbitrary functions, to an arbitrary value `oa:Option[A]`. In Scala code, it is convenient to use the method `map` and write `oa.map(f)` instead of the equivalent expression `fmap(f)(oa)`:

```
fmap(f andThen g)(oa) == oa.map(f andThen g) == oa match {
  case None      => None
  case Some(x)   => (f andThen g)(x)
}
```

Since `(f andThen g)(x) == g(f(x))`, we rewrite the result as:

```
oa.map(f andThen g) == oa match {
  case None      => None
  case Some(x)   => g(f(x))
}
```

Now we consider the left-hand side of the law, $\text{fmap}\,(f) ⨾ \text{fmap}\,(g)$, and write the Scala expressions:

```
oa.map(f).map(g) == (oa match {
  case None      => None
  case Some(x)   => f(x)
}).map(g) == (oa match {
  case None      => None
  case Some(x)   => f(x)
}) match {
  case None      => None
  case Some(y)   => g(y)
} == oa match {
  case None      => None
  case Some(x)   => g(f(x))
}
```

We find that the two sides of the law have identical code.

The derivation is much shorter in the matrix notation. We use Eq. (6.1) as the definition of `fmap` and omit the types:

$$\text{fmap}\,(f) ⨾ \text{fmap}\,(g) = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & f \end{array} \right| ⨾ \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & g \end{array} \right|$$

$$\text{matrix composition}: \quad = \left\| \begin{array}{cc} \text{id} ⨾ \text{id} & \mathbb{0} \\ \mathbb{0} & f ⨾ g \end{array} \right| = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & f ⨾ g \end{array} \right|$$

$$\text{definition of fmap}: \quad = \text{fmap}\,(f ⨾ g) \quad .$$

These calculations prove that the `map` method of the `Option` type satisfies the composition law. If the composition law did not hold, we would incorrectly imagine the way `map` manipulates data within the `Option` wrapper. Looking at the Scala code example above, we expect `c.map(incr).map(incr)` to increment the data wrapped by `c` two times. If the result of `c.map(incr).map(incr)` were not `Some(2)` but, say, `Some(1)` or `None`, our ordinary intuitions about data transformations would become incorrect. In other words, violations of the composition law prevent us from understanding the code via mathematical reasoning about transformation of data values.

The composition law is a rigorous formulation of the requirement that wrapped data should be transformed (by lifted functions) in the same way as ordinary data. For example, the following associativity property holds for lifted functions:

Statement 6.1.3.1 For arbitrary functions $f^{:A\to B}$, $g^{:B\to C}$, and $h^{:C\to D}$, we have:

$$\text{fmap}\,(f) ⨾ \text{fmap}\,(g ⨾ h) = \text{fmap}\,(f ⨾ g) ⨾ \text{fmap}\,(h) \quad .$$

Proof The left-hand side is rewritten as:

$$\begin{aligned} & \text{fmap}\,(f) ⨾ \underline{\text{fmap}\,(g ⨾ h)} \\ \text{composition law for } (g ⨾ h): \quad & = \text{fmap}\,(f) ⨾ \big(\text{fmap}\,(g) ⨾ \text{fmap}\,(h)\big) \\ \text{associativity law (4.3)}: \quad & = \underline{\big(\text{fmap}\,(f) ⨾ \text{fmap}\,(g)\big)} ⨾ \text{fmap}\,(h) \\ \text{composition law for } (f ⨾ g): \quad & = \text{fmap}\,(f ⨾ g) ⨾ \text{fmap}\,(h) \quad , \end{aligned}$$

which now equals the right-hand side. This proves the statement.

6.1.4 Functors: definition and examples

Separating the functionality of "data wrapper" from any other features of a data type, we obtain:

- A data type with a type parameter, e.g., `L[A]`. We will use the notation $L^\bullet$ (in Scala, `L[_]`) for the type constructor itself when the name of the type parameter is not needed.
- A fully parametric function `fmap` with this type signature:

$$\text{fmap}_L : (A \to B) \to L^A \to L^B \quad .$$

- Two laws that the function `fmap` must satisfy:

$$\text{identity law of } L: \quad \text{fmap}_L(\text{id}^{:A\to A}) = \text{id}^{:L^A\to L^A} \quad , \tag{6.2}$$

$$\text{composition law of } L: \quad \text{fmap}_L(f^{:A\to B} ⨾ g^{:B\to C}) = \text{fmap}_L(f^{:A\to B}) ⨾ \text{fmap}_L(g^{:B\to C}) \quad . \tag{6.3}$$

A type constructor $L^\bullet$ with these properties is called a **functor**. The laws (6.2)–(6.3) are the functor laws of identity and composition.

When a law involves function compositions, it is helpful to draw a type diagram to clarify how the functions transform various types involved in the law. A **type diagram** is a directed graph whose vertices are types and edges are functions mapping one type to another. Function composition corresponds to following a path in the diagram. A type diagram for the composition law (6.3) is shown at left. There are two paths from L^A to L^C; by Eq. (6.3), both paths must give the same result. Mathematicians call such diagrams **commutative**.

$$\begin{tikzcd} & L^B \arrow[dr, "\text{fmap}_L(g^{:B\to C})"] & \\ L^A \arrow[ur, "\text{fmap}_L(f^{:A\to B})"] \arrow[rr, "\text{fmap}_L(f^{:A\to B} ⨾ g^{:B\to C})"'] & & L^C \end{tikzcd}$$

Type diagrams are easier to read when using the *forward* composition $(f ⨾ g)$ because the order of edges is the same as the order of functions in the composition. To see this, compare Eq. (6.3) and the type diagram above with the same law written using the backward composition:

$$\text{fmap}_L(g^{:B\to C} \circ f^{:A\to B}) = \text{fmap}_L(g^{:B\to C}) \circ \text{fmap}_L(f^{:A\to B}) \quad .$$

The function `map` is equivalent to `fmap` and can be defined through `fmap` by:

$$\text{map}_L : L^A \to (A \to B) \to L^B \quad ,$$
$$\text{map}_L(x^{:L^A})(f^{:A\to B}) = \text{fmap}_L(f^{:A\to B})(x^{:L^A}) \quad .$$

Each of the type constructors `Option`, `Seq`, `Try`, and `Future` has its own definition of `map` but the functor laws remain the same. We use the subscript L when writing map_L and fmap_L, in order to indicate clearly the type constructor those functions work with.

We will now look at some examples of type constructors that are functors.

Standard data structures Many type constructors defined in the Scala library have a `map` method, and almost all of them are functors. Some often used functors are:

- The standard disjunctive types `Option`, `Try`, and `Either[A, B]` (where, by default, transformations apply to the type parameter `B`).
- The linear sequence `Seq` and its various subtypes such as `List`, `Range`, `Vector`, `IndexedSeq`, and `Stream`.
- The "task-like" functors: `Future` and its alternatives: `Task` (provided by the `monix` library), `Async` and `Concurrent` (provided by the `cats-effect` library), `ZIO` (provided by the `zio` library).
- Dictionaries: `Map[K, V]` with respect to the type parameter `V`. The method is called `mapValues` instead of `map`: it transforms the values in the dictionary, leaving the keys unchanged.

Application-specific, custom type constructors defined by the programmer, such as case classes with type parameters, are often functors. Their structure is simple and helps build intuition for functors, so let us now consider some examples of case classes that are functors. In this book, they are called "polynomial functors".

Polynomial functors Type constructors built with primitive types, type parameters, products, and disjunctions (or "sums") are often used to represent application-specific data. Consider the code:

```
final case class Counted[A](n: Int, a: A) {
  def map[B](f: A => B): Counted[B] = Counted(n, f(a))
}
```

The data type `Counted[A]` may be used to describe `n` repetitions of a given value `a: A`. The code already defines the method `map` for the `Counted` class, which can be used like this:

```
scala> Counted(10, "abc").map(s => "prefix " + s)
res0: Counted[String] = Counted(10,prefix abc)
```

It is often more convenient to implement `map` as a class method rather than as a function such as:

```
def map[A, B](c: Counted[A])(f: A => B): Counted[B] = c match {
  case Counted(n, a) => Counted(n, f(a))
}
```

The type notation for `Counted` is:

$$\text{Counted}^A \stackrel{\text{def}}{=} \text{Int} \times A \quad ,$$

showing that `Counted[_]` is a polynomial type constructor. The existence of a `map` method suggests that `Counted[_]` is a functor. We still need to check that the functor laws hold for it.

Example 6.1.4.1 Verify that the above implementation of `map` for `Counted` satisfies the functor laws.

Solution The implementation of `map` is fully parametric since it does not perform any type-specific operations; it uses the value `n:Int` as if `Int` were a type parameter. It remains to check that the laws hold. We will first verify the laws using the Scala syntax and then using the code notation.

The identity law means that for all `n: Int` and `a: A` we must have:

```
Counted(n, a).map(identity) == Counted(n, a)
```

To verify this, we substitute the code of `map` and find:

```
Counted(n, a).map(identity) == Counted(n, identity(a)) == Counted(n, a)
```

The composition law means that for all `n: Int`, `a: A`, `f: A => B`, and `g: B => C`, we must have:

```
Counted(n, a).map(f).map(g) == Counted(n, a).map(f andThen g)
```

Substitute the Scala code of `map` into the left-hand side:

```
Counted(n, a).map(f).map(g) == Counted(n, f(a)).map(g) == Counted(n, g(f(a)))
```

The right-hand side can be transformed to the same expression:

```
Counted(n, a).map(f andThen g) == Counted(n, (f andThen g)(a)) == Counted(n, g(f(a)))
```

Let us now write a proof in the code notation, formulating the laws via the `fmap` method:

$$\text{fmap}_{\text{Counted}}(f^{:A\to B}) \stackrel{\text{def}}{=} \left(n^{:\text{Int}} \times a^{:A} \to n \times f(a)\right) \quad .$$

To verify the identity law, we write:

$$\begin{aligned}
\text{expect to equal id}: \quad & \text{fmap}_{\text{Counted}}(\text{id}) \\
\text{definition of fmap}_{\text{Counted}}: \quad & = \left(n \times a \to n \times \underline{\text{id}\,(a)}\right) \\
\text{definition of id}: \quad & = (n \times a \to n \times a) = \text{id} \quad .
\end{aligned}$$

To verify the composition law:

$$\begin{aligned}
\text{expect to equal fmap}_{\text{Counted}}(f \,⨾\, g): \quad & \text{fmap}_{\text{Counted}}(f) \,⨾\, \text{fmap}_{\text{Counted}}(g) \\
\text{definition of fmap}_{\text{Counted}}: \quad & = (n \times a \to n \times f(a)) \,⨾\, (n \times b \to n \times g(b)) \\
\text{compute composition}: \quad & = n \times a \to n \times \underline{g(f(a))} \\
\text{definition of } (f \,⨾\, g): \quad & = (n \times a \to n \times (f \,⨾\, g)(a)) = \text{fmap}_{\text{Counted}}(f \,⨾\, g) \quad .
\end{aligned}$$

We will prove later that all polynomial type constructors have a definition of `map` that satisfies the functor laws. It will be clear without proof that our definition of `map` for `Counted` is correct.

What would be an *incorrect* implementation of `map`? As an example, `map` could transform `Counted(n, a)` as before, except that the value `n` is now used to count the number of times `map` is applied:

```
def map_bad[A, B](c: Counted[A])(f: A => B): Counted[B] = c match {
  case Counted(n, a) => Counted(n + 1, f(a))
}
```

This implementation may appear reasonable. However, it violates both functor laws. For instance:

```
Counter(n, a) != map_bad(Counter(n, a))(identity) == Counter(n + 1, a) // Identity law does not hold.
```

The failure of functor laws leads to surprising behavior because a code refactoring changes the result:

```
Counter(n, a).map(incr).map(incr) != Counter(n, a).map(x => x + 2)
```

Let us look at some other examples of polynomial type constructors.

Example 6.1.4.2 Implement the `fmap` function for the data type `Vec3` defined by:

```
case class Vec3[A](x: A, y: A, z: A)
```

Solution Begin by implementing a fully parametric function in a straightforward way:

```
def fmap[A, B](f: A => B): Vec3[A] => Vec3[B] = {
  case Vec3(x, y, z) => Vec3(f(x), f(y), f(z))  // Apply ‘f‘ to all data of type ‘A‘, keep the order.
}
```

Since all three values `f(x)`, `f(y)`, `f(z)` have type `B`, the code of `fmap` would still satisfy the required type signature if it returned, say, `Vec3(f(z), f(x), f(x))` or some other combination of those values. However, we would then fail to preserve information about the values `x`, `y`, `z` and about their ordering in the original data, `Vec(x, y, z)`. For this reason, we use the implementation of `fmap` shown first.

The type notation for the type constructor `Vec3[_]` is:

$$\text{Vec}_3{}^A \stackrel{\text{def}}{=} A \times A \times A \quad ,$$

and the code notation for `fmap` is:

$$\text{fmap}_{\text{Vec}_3}(f^{:A\to B}) \stackrel{\text{def}}{=} x^{:A} \times y^{:A} \times z^{:A} \to f(x) \times f(y) \times f(z) \quad .$$

Example 6.1.4.3 Implement the `fmap` function for this type constructor:

$$\text{QueryResult}^A \stackrel{\text{def}}{=} \text{String} + \text{String} \times \text{Long} \times A \quad .$$

Solution Begin by implementing the type constructor in Scala:

```
sealed trait QueryResult[A]
case class Error[A](message: String)                      extends QueryResult[A]
case class Success[A](name: String, time: Long, data: A) extends QueryResult[A]
```

Now implement a fully parametric, information-preserving function with the type signature of `fmap` for this type constructor:

```
def fmap[A, B](f: A => B): QueryResult[A] => QueryResult[B] = {
  case Error(message)              => Error(message)
  case Success(name, time, data)   => Success(name, time, f(data))
}
```

As in the previous example, we treat specific types (`Long`, `String`) as if they were type parameters. In this way, we obtain a correct implementation of `fmap` that satisfies the functor laws.

Recursive polynomial functors Recursive disjunctive type constructors shown in Section 3.3, such as lists and trees, are functors. Their `fmap` methods use recursion (the body of `fmap` calls itself at some places). It is often *not* easy to implement those `fmap` methods tail-recursively! Since our main focus is to prove that various laws hold for `fmap`, we will prefer shorter and more straightforward code even if it is not tail-recursive. Once the laws are shown to hold for that code, the programmer is free to use an equivalent but more efficient implementation of the same function.

Proving laws for a recursive function needs mathematical induction. For a first example of such a proof, we use the `List` data type (see Section 3.3.1). We view `List` as a recursive polynomial functor whose `map` method was shown in Section 3.3.2. The equivalent `fmap` method is:

```
def fmap[A, B](f: A => B): List[A] => List[B] = {
  case Nil            => Nil
  case head :: tail   => f(head) :: fmap(f)(tail)
}
```

Example 6.1.4.4 Prove that `List`'s `fmap` method satisfies the functor laws.

Solution We will first write the proof in the Scala syntax and then in the code notation.

To verify the identity law, we need to show that `fmap(identity) == identity`. Both sides of this law are functions, so we need to apply them to an arbitrary value `s` of type `List[A]` and show that `fmap(identity)(s) == s`. The proof goes by induction on the length of `s`. The value `s` must be either `Nil` or `head :: tail`, since these are the only ways of creating a `List` value.

The base case is an empty list (`s == Nil`). To prove the base case, we apply the code of `fmap` to `Nil`. The code goes into the first line: `case Nil => Nil`. So, we have proved that `fmap(identity)(Nil) == Nil`.

To prove the inductive step, we need to show that the law holds for lists of length $n+1$ assuming it holds for lists of length n. This corresponds to `s == head :: tail` where `head` is a value of type `A` and `tail` is a list of length `n`. By the inductive assumption, the law already holds when applied to `tail`, so `fmap(identity)(tail) == tail`. Then we compute `fmap(identity)(s)` as:

```
fmap(identity)(head::tail) == identity(head) :: fmap(identity)(tail) == head :: fmap(identity)(tail)
```

The last result is equal to `head :: tail == s` due to the inductive assumption.

Note that the expression `fmap(f)(tail)` in the code of `fmap` is a *recursive call* to `fmap`. So, the inductive assumption says that the law already holds for any recursive calls of the function `fmap`. We will see this pattern in all proofs of laws for recursive functions.

To prove the composition law of `fmap`, take arbitrary functions `f: A => B` and `g: B => C`, and apply both sides of the law to an arbitrary list `s: List[A]`. We need to show that:

```
fmap(g)(fmap(f)(s)) == fmap(f andThen g)(s)
```

The base case of induction is an empty list, `s == Nil`. In that case, the code of `fmap` always returns an empty list, so the law holds:

```
/* left-hand side of the law:  */   fmap(g)(fmap(f)(Nil)) == fmap(g)(Nil) == Nil
/* right-hand side of the law: */   fmap(f andThen g)(Nil) == Nil
```

The inductive step assumes that the law holds for any list of length n. We need to verify the law for a list `s == head :: tail` of length $n+1$, where `tail` is a list of length n. The left-hand side of the law is:

```
/* left-hand side of the law:  */   fmap(g)(fmap(f)(head :: tail))
/* expand the code of fmap(f): */   == fmap(g)(f(head) :: fmap(f)(tail))
/* expand the code of fmap(g): */   == g(f(head)) :: fmap(g)(fmap(f)(tail))
/* the inductive assumption:   */   == g(f(head)) :: fmap(f andThen g)(tail)
```

Now compute the right-hand side and show that it equals the left-hand side:

```
/* right-hand side of the law: */   fmap(f andThen g)(head :: tail)
/* expand the code of fmap:    */   == (f andThen g)(head) :: fmap(f andThen g)(tail)
/* expand (f andThen g)(head): */   == g(f(head)) :: fmap(f andThen g)(tail)
```

Turning now to the code notation, we begin by writing the definition of the type constructor List^A:

$$\text{List}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times \text{List}^A \quad .$$

The code of the `fmap` method is written in the matrix notation as:

$$\text{fmap}_{\text{List}}(f^{:A\to B}) \stackrel{\text{def}}{=} \begin{array}{|c||cc|} & \mathbb{1} & B \times \text{List}^B \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A \times \text{List}^A & \mathbb{0} & h^{:A} \times t^{:\text{List}^A} \to f(h) \times \overline{\text{fmap}}_{\text{List}}(f)(t) \end{array} \quad .$$

In this book, recursive calls to functions are indicated by an overline, as in $\overline{\text{fmap}}$. Keep in mind that the symbol $\overline{\text{fmap}}_{\text{List}}$ denotes *the same* function as $\text{fmap}_{\text{List}}$. The overline is used only as a reminder that certain parts of code are recursive calls.

To verify the identity law, we apply $\text{fmap}_{\text{List}}$ to an identity function ($\text{id}^{:A\to A}$) and find:

$$\text{fmap}_{\text{List}}(\text{id}) = \left\| \begin{array}{c|c} & \mathbb{1} \quad\quad A\times\text{List}^A \\ \hline \mathbb{1} & \text{id} \quad\quad \mathbb{0} \\ A\times\text{List}^A & \mathbb{0} \quad h\times t \to \text{id}(h)\times\overline{\text{fmap}}_{\text{List}}(\text{id})(t) \end{array} \right\| \quad .$$

The inductive assumption says that any recursive calls to `fmap` already satisfy the identity law. We write this in the code notation as:

$$\overline{\text{fmap}}_{\text{List}}(\text{id}) = \text{id} \quad .$$

Using this assumption, we write (omitting types for brevity):

$$\text{fmap}_{\text{List}}(\text{id}) = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h\times t \to \underline{\text{id}(h)\times\text{id}(t)} \end{array} \right\| = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h\times t \to h\times t \end{array} \right\| \quad .$$

Since $h\times t \to h\times t$ is just an identity function of type $A\times\text{List}^A \to A\times\text{List}^A$, we find:

$$\text{fmap}_{\text{List}}(\text{id}) = \left\| \begin{array}{c|cc} & \mathbb{1} & A\times\text{List}^A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A\times\text{List}^A & \mathbb{0} & \text{id} \end{array} \right\| = \text{id} \quad .$$

A diagonal code matrix whose elements are identity functions will never change any values in any of the disjunctive parts. So, that matrix is equal to the identity function applied to the entire disjunctive type. This concludes the proof of the identity law of $\text{fmap}_{\text{List}}$.

To verify the composition law, we write (omitting types for brevity):

$$\text{fmap}_{\text{List}}(f)\,⨾\,\text{fmap}_{\text{List}}(g) = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h\times t \to f(h)\times\overline{\text{fmap}}_{\text{List}}(f)(t) \end{array} \right\| ⨾ \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h\times t \to g(h)\times\overline{\text{fmap}}_{\text{List}}(g)(t) \end{array} \right\|$$

$$\text{matrix composition}: \quad = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h\times t \to g(f(h))\times\overline{\text{fmap}}_{\text{List}}(g)\big(\overline{\text{fmap}}_{\text{List}}(f)(t)\big) \end{array} \right\| \quad .$$

By the inductive assumption, the law already holds for recursive calls to $\overline{\text{fmap}}_{\text{List}}$:

$$\overline{\text{fmap}}_{\text{List}}(g)\big(\overline{\text{fmap}}_{\text{List}}(f)(t)\big) = t \triangleright \overline{\text{fmap}}_{\text{List}}(f)\,⨾\,\overline{\text{fmap}}_{\text{List}}(g) = t \triangleright \overline{\text{fmap}}_{\text{List}}(f\,⨾\,g) \quad .$$

This allows us to complete the proof of the law:

$$\text{fmap}_{\text{List}}(f)\,⨾\,\text{fmap}_{\text{List}}(g) = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h\times t \to (h \triangleright f\,⨾\,g)\times\overline{\text{fmap}}_{\text{List}}(f\,⨾\,g)(t) \end{array} \right\| = \overline{\text{fmap}}_{\text{List}}(f\,⨾\,g) \quad .$$

Example 6.1.4.5 Define a list of *odd* length as a recursive type constructor $\text{LO}^\bullet$:

$$\begin{aligned} \text{LO}^A &\stackrel{\text{def}}{=} A + A\times A\times\text{LO}^A \\ &\cong A + A\times A\times A + A\times A\times A\times A\times A + \ldots \quad , \end{aligned} \tag{6.4}$$

and implement `fmap` for it (without proving the laws).

Solution The Scala definition of the type constructor `LO[_]` is:

```
sealed trait LO[A]
final case class LO1[A](x: A)                    extends LO[A]
final case class LO2[A](x: A, y: A, tail: LO[A]) extends LO[A]
```

We can implement `fmap` as a recursive function:

```
def fmap[A, B](f: A => B): LO[A] => LO[B] = {
  case LO1(x)            => LO1[B](f(x))
  case LO2(x, y, tail)   => LO2[B](f(x), f(y), fmap(f)(tail))
}
```

This code for `fmap` is *not* tail-recursive because `fmap` is called inside the case class constructor `LO2`.

The type constructor $\text{LO}^\bullet$ is a **recursive polynomial functor** because it is defined by a recursive type equation (6.4) that uses only polynomial type constructions ("sums and products") in its right-hand side. Other examples of recursive polynomial functors are lists and trees. We will prove later (Statement 6.2.3.7) that `LO[_]` (as well as other list-like and tree-like data types) are lawful functors.

6.1.5 Functor block expressions

Computations with wrapped values often require a chain of `map` methods, e.g.:

```
scala> val result = Map(1 -> "one", 2 -> "two", 3 -> "three").
  map { case (i, name) => (i * i, name) }.         // Compute i * i.
  map { case (x, name) => (x, s"$name * $name") }. // Compute product message.
  map { case (x, product) => s"$product is $x" }   // Compute final message.
result: Seq[String] = List(one * one is 1, two * two is 4, three * three is 9)
```

Such code can be rewritten equivalently in the **functor block** syntax:

```
scala> val result = for {
  (i, name) <- Map(1 -> "one", 2 -> "two", 3 -> "three") // For each (i, name)...
  x = i * i                    // define ‘x‘ by computing i * i...
  product = s"$name * $name"   // define ‘product‘...
} yield s"$product is $x"      // and put these expressions into the ‘result‘ sequence.
result: Seq[String] = List(one * one is 1, two * two is 4, three * three is 9)
```

Written in this way, the computations are easier to understand for two main reasons. First, there is less code to read and to write; no `map` or `case` and fewer curly braces. Second, values such as `name` and `x` need to be kept in tuples to pass them from one `map` function to another, but any line in a functor block can directly reuse all values defined in previous lines. Let us look at this in more detail.

The functor block syntax is helpful in functional programming because it replaces a chain of `map` methods (as well as `filter` and `flatMap` methods, as we will see in later chapters) by a visually clearer sequence of definitions and expressions. Scala defines a functor block via the keywords `for` and `yield`. We will see many examples of functor blocks throughout this book. In this chapter, we only consider functor blocks that are equivalent to a chain of `map` operations on a functor value `p: L[A]`. These functor blocks can be recognized because they contain *only one* left arrow (in the first line). Here is how to replace a chain of `map` operations by a functor block:

```
p.map(x => f(x)).map(y => g(y)).map(z => h(z)) == for {
   x <- p           // The first line must contain a left arrow before a functor value ‘p‘.
   y = f(x)         // Some computation involving ‘x‘.
   z = g(y)         // Another computation, uses ‘y‘.
} yield h(z)        // The ‘yield h(z)‘ replaces the inner result of the last ‘map‘.
```

Translating functor blocks back into a chain of `map` operations is straightforward except that in some cases the `map` operations may need to create some intermediate tuples that are not needed in the functor block syntax. Consider this code:

```
val result: L[B] = for {
  x <- p           // The first line must contain a left arrow before a functor value ‘p‘.
  y = f(x)         // Some computation involving ‘x‘.
```

```
  z = g(x, y)      // Another computation, uses ‘x‘ and ‘y‘.
  ...
} yield q(x, y, z) // The ‘yield‘ may use ‘x‘, ‘y‘, ‘z‘, and any other defined variables.
```

The above functor block code assumes that `q(x, y, z)` has type `B`, and is equivalent to:

```
val result: L[B] = p
  .map { x => (x, f(x)) }  // Create a tuple because we need to keep ‘x‘ and ‘f(x)‘.
  .map { case (x, y) => (x, y, g(x, y)) }   // Need to keep ‘x‘, ‘y‘, and ‘g(x, y)‘.
  ...
  .map { case (x, y, z) => q(x, y, z) }       // Here, we can use ‘x‘, ‘y‘, and ‘z‘.
```

This code creates intermediate tuples only because the values `x`, `y`, `z` need to be used in later calculations. The functor block code is easier to read, write, and modify than a chain of `map`s with tuples.

If desired, functor blocks may be written in a single line by using semicolons to separate the individual steps:

```
scala> for { x <- List(1, 2, 3); y = x * x; z = y + 2 } yield z
res0: List[Int] = List(3, 6, 11)
```

A confusing feature of the `for` / `yield` syntax is that, at first sight, functor blocks (such as the code shown at left) appear to compute, or to "yield", the value `expr(x)`. However, this is not so. As the above examples show, if `p` is a sequence then the functor block also computes a *sequence*. In general, the result of a functor block is a "wrapped" value, where the type of the "wrapper" is determined by the first line of the functor block. The first line must have a left arrow followed by a "source", which must be an expression of a functor type, i.e., of type `L[A]` for some functor `L[_]`. The result's type will be `L[B]` where `B` is the type of the expression after the `yield` keyword.

```
for { x <- p; ... } yield expr(x)
```

For instance, the first line of the following functor block contains an `Option` value, `Some(123)`, as the "source". Because of that, the value of the entire functor block expression will also be of type `Option`:

```
scala> for {
  x <- Some(123)   // "Source" is Option[Int].
  y = (x - 3) / 10
} yield { if (y > 0) s"Have $y" else "Error" }
res1: Option[String] = Some(Have 12)
```

In this code, the `yield` keyword is followed by an expression of type `String`. So, the result of the entire functor block is of type `Option[String]`. Note that the expression after the "`yield`" can be a block of arbitrary code containing new `val`s, new `def`s, and/or other `for`/`yield` functor blocks if needed.

Functor blocks can be used with any functor that has a `map` method, not only with library-defined type constructors such as `Seq` or `Option`. Here are some examples of defining the `map` methods and using functor blocks with disjunctive types.

The type constructor `QueryResult[_]` may define the `map` method on the trait itself and split its implementation between the case classes like this:

```
sealed trait QueryResult[A] {
  def map[B](f: A => B): QueryResult[B] // No implementation here.
}
case class Error[A](message: String)                        extends QueryResult[A] {
  def map[B](f: A => B): QueryResult[B] = Error(message)
}
case class Success[A](name: String, time: Long, data: A)   extends QueryResult[A] {
  def map[B](f: A => B): QueryResult[B] = Success(name, time, f(data))
}
```

After these definitions, we can use `QueryResult` in functor blocks:

```
val q: QueryResult[Int] = Success("addresses", 123456L, 10)
scala> val result = for {
  x <- q
  y = x + 2
} yield s"$y addresses instead of $x"
result: QueryResult[String] = Success(addresses,123456,12 addresses instead of 10)
```

As another example, let us define the `map` method on the `L0` trait (a recursive disjunctive type):

```
sealed trait LO[A] {
  def map[B](f: A => B): LO[B]
}
final case class LO1[A](x: A)                        extends LO[A] {
  def map[B](f: A => B): LO[B] = LO1[B](f(x))
}
final case class LO2[A](x: A, y: A, tail: LO[A])   extends LO[A] {
  def map[B](f: A => B): LO[B] = LO2[B](f(x), f(y), tail.map(f))
}
```

After these definitions, we may use values of type `LO[_]` in functor blocks:

```
scala> val result = for {
         x <- LO2("a", "quick", LO2("brown", "fox", LO1("jumped")))
         y = x.capitalize
         z = y + "/"
       } yield (z, z.length)
result: LO[(String, Int)] = LO2((A/,2),(Quick/,6),LO2((Brown/,6),(Fox/,4),LO1((Jumped/,7))))
```

Functor blocks and functor laws There is an important connection between the functor laws and the properties of code in functor blocks. Consider the following code:

```
def f(x: Int) = x * x    // Some computations.
def g(x: Int) = x - 1    // More computations.

scala> for {
  x <- List(10, 20, 30)
  y = x
  z = f(y)   // Perform computations.
} yield g(z)
res0: List[Int] = List(99, 399, 899)
```

The code says that `x = y`, so it appears reasonable to eliminate `y` and simplify this code into:

```
scala> for {
  x <- List(10, 20, 30)    // Eliminated ‘y‘ from the code.
  z = f(x)   // Perform computations.
} yield g(z)
res1: List[Int] = List(99, 399, 899)
```

Another example of refactoring that appears reasonable is to combine transformations:

```
scala> for {
  x <- List(10, 20, 30)
  y = x + 1
  z = f(y)   // Perform computations.
} yield g(z)
res2: List[Int] = List(120, 440, 960)
```

The code says that `y = x + 1`, so we may want to replace `f(y)` by `f(x + 1)`:

```
scala> for {
  x <- List(10, 20, 30)
  z = f(x + 1)   // Eliminated ‘y‘ from the code.
} yield g(z)
res3: List[Int] = List(120, 440, 960)
```

Looking at these code changes, we expect that the computed results will remain the same. Indeed, when the code directly states that `x = y`, it would be confusing and counter-intuitive if the result value changed after replacing `y` by `x`. When the code says that `y = x + 1`, ordinary mathematical reasoning suggests that `f(y)` can be replaced by `f(x + 1)` without affecting the results.

To see the connection with the functor laws, we translate the functor block syntax line by line into chains of `map` methods. The resulting code fragments are shown in Table 6.1. The fragments using `map` methods were split into lines to emphasize their close correspondence to functor blocks.

We find that code fragments `1b` and `2b` are equal only if `.map(x => x)` does not modify the list to

Functor block syntax	Chains of `map` methods

`for { // Code fragment 1a.`
`  x <- List(10, 20, 30)`
`  y = x`
`  z = f(y)`
`} yield g(z)`
	`List(10, 20, 30) // Code fragment 1b.`
	`  .map(x =>`
	`  x).map(y =>`
	`  f(y) ).map(z =>`
	`  g(z) )`

`for { // Code fragment 2a.`
`  x <- List(10, 20, 30)`
`  z = f(x)`
`} yield g(z)`
	`List(10, 20, 30) // Code fragment 2b.`
	`  .map(x =>`
	`  f(x) ).map(z =>`
	`  g(z) )`

`for { // Code fragment 3a.`
`  x <- List(10, 20, 30)`
`  y = x + 1`
`  z = f(y)`
`} yield g(z)`
	`List(10, 20, 30) // Code fragment 3b.`
	`  .map(x =>`
	`  x + 1).map(y =>`
	`  f(y) ).map(z =>`
	`  g(z) )`

`for { // Code fragment 4a.`
`  x <- List(10, 20, 30)`
`  z = f(x + 1)`
`} yield g(z)`
	`List(10, 20, 30) // Code fragment 4b.`
	`  .map(x =>`
	`  f(x + 1) ).map(z =>`
	`  g(z) )`

Table 6.1: Example translations of functor blocks into `map` methods.

which it applies. This holds if the `map` method obeys the functor identity law, `p.map(identity) == p`, for all `p` of the appropriate type. We also find that code fragments `3b` and `4b` are equal if we can replace `.map(x => x + 1).map(f)` by `.map(x => f(x + 1))`. This replacement is justified as long as the `map` method obeys the functor composition law:

```
p.map(h).map(f) == p.map(x => f(h(x)))
```

for all `p` and functions `h` and `f` of appropriate types.

Functor laws guarantee that we can correctly understand and modify code written in functor blocks, reasoning about transformations of values as we do in mathematics.

6.1.6 Examples of non-functors

What properties of a data type make it a functor? To build an intuition, it is helpful to see examples of data types that are *not* functors.

There are several possibilities for a type constructor to fail being a functor:

- A `map` function's type signature cannot be implemented at all.
- A `map` function can be implemented but cannot satisfy the functor laws.
- A given `map` function is incorrect (does not satisfy the laws), although the error could be fixed: a different implementation of `map` satisfies the laws.
- A given `map[A, B]` function satisfies the laws for most types `A` and `B`, but violates the laws for certain specially chosen types.

We will now look at examples illustrating these possibilities.

Cannot implement `map`'s type signature Consider the type constructor $H^\bullet$ defined by:

```
final case class H[A](r: A => Int)
```

$$H^A \stackrel{\text{def}}{=} A \to \text{Int} \quad .$$

The data type `H[A]` does not wrap data of type A; instead, it is a function that *consumes* data of type A. One cannot implement a fully parametric `map` function with the required type signature:

$$\text{map}^{A,B} : (A \to \text{Int}) \to (A \to B) \to (B \to \text{Int}) \quad .$$

To see this, recall that a fully parametric function needs to treat all types as type parameters, including the primitive type `Int`. So the code

```
def map[A, B]: H[A] => (A => B) => H[B] = { r => f => C(_ => 123) }
```

satisfies the type signature of `map` but is not fully parametric because it returns a specific value `123` of type `Int`, which is not allowed. Replacing the type `Int` by a new type parameter N, we obtain the type signature:

$$\text{map}^{A,B,N} : (A \to N) \to (A \to B) \to B \to N \quad .$$

We have seen in Example 5.4.1.8 that this type signature is not implementable. So, the type constructor H is not a functor.

Another important example where the `map`'s type signature cannot be implemented are certain kinds of type constructors called **generalized algebraic data types** (GADTs). In this book, they are called **unfunctors** for brevity. An unfunctor is a type constructor having special values when its type parameter is set to certain specific types. An example of an unfunctor is:

```
sealed trait ServerAction[R]
final case class StoreId(x: Long, y: String)  extends ServerAction[Boolean]
final case class StoreName(name: String)      extends ServerAction[Int]
```

We see that some parts of the disjunctive type `ServerAction[R]` do not carry the type parameter `R` but instead set `R` to specific types, `R = Boolean` and `R = Int`. As a consequence, e.g., the case class `StoreName` has no type parameters and can only represent values of type `ServerAction[Int]` but not, say, `ServerAction[String]`. For this reason, `ServerAction[A]` cannot have a fully parametric `map` function:

```
def map[A, B]: ServerAction[A] => (A => B) => ServerAction[B]
```

To implement `map`, we are required to support any choice of the type parameters `A` and `B`. For example, with `A = Int`, we must be able to transform the value `StoreName("abc")`, which has the type `ServerAction[Int]`, to a value of type `ServerAction[B]` with any given `B`. However, we can only create values of types `ServerAction[Boolean]` or `ServerAction[Int]`, because the disjunctive type `ServerAction[R]` has only two parts with the specified types. So, we cannot create a value of type `ServerAction[B]` with an unknown arbitrary type `B`. For this reason, it is impossible to implement a fully parametric `map` function with the type signature shown above.

We are prevented from implementing `map` because some type parameters are already set in the definition of `ServerAction[R]`. One can say that the unfunctor `ServerAction[_]` fails to be fully parametric *in its type definition*. This behavior of unfunctors is intentional; unfunctors are only used in situations where the lack of `map` does not lead to problems (see Chapter 13).

Cannot implement a lawful `map` An example of a non-functor of this kind is:

$$Q^A \stackrel{\text{def}}{=} (A \to \text{Int}) \times A \quad .$$

Here is the Scala code for this type constructor:

```
final case class Q[A](q: A => Int, a: A)
```

A fully parametric `map` function with the correct type signature *can* be implemented (and there is only one such implementation):

$$\text{map}^{A,B} \stackrel{\text{def}}{=} q^{:A\to\text{Int}} \times a^{:A} \to f^{:A\to B} \to (_ \to q(a))^{:B\to\text{Int}} \times f(a) \quad .$$

The corresponding Scala code is:

```
def map[A, B]: Q[A] => (A => B) => Q[B] = { qa => f =>
  Q[B](_ => qa.q(qa.a), f(qa.a))
}
```

This `map` function is fully parametric (since it treats the type `Int` as a type parameter) and has the right type signature, but the functor laws do not hold. To show that the

identity law fails, we consider an arbitrary value $q^{:A\to\text{Int}} \times a^{:A}$ and compute:

$$\begin{aligned}
\text{expect to equal } q\times a: &\quad \text{map}\,(q\times a)(\text{id})\\
\text{definition of map}: &\quad = (_\to q(a))\times \underline{\text{id}\,(a)}\\
\text{definition of id}: &\quad = (_\to q(a))\times a\\
\text{expanded function, } q=(x\to q(x)): &\quad \neq q\times a = (x\to q(x))\times a \quad .
\end{aligned}$$

The law must hold for arbitrary functions $q^{:A\to\text{Int}}$, but the function $(_\to q(a))$ always returns the same value $q(a)$ and thus is not equal to the original function q. So, the result of evaluating the expression $\text{map}(q\times a)(\text{id})$ is not always equal to the original value $q\times a$.

Since this `map` function is the only available implementation of the required type signature, we conclude that $Q^\bullet$ is not a functor (we cannot implement `map` that satisfies the laws).

Mistakes in implementing `map` Non-functors of the third kind are type constructors with an incorrectly implemented `map`. An example is a type constructor $P^A \stackrel{\text{def}}{=} A\times A$ with the `map` function:

$$\text{map} \stackrel{\text{def}}{=} x^{:A}\times y^{:A}\to f^{:A\to B}\to f(y)\times f(x) \quad .$$

Here is the Scala code corresponding to this code notation:

```
def map[A, B](p: (A, A))(f: A => B): (B, B) = p match { case (x, y) => (f(y), f(x)) }
```

This code swaps the values in the pair `(x, y)`; we could say that it fails to preserve information about the order of those values. The functor identity law does not hold:

$$\begin{aligned}
\text{expect to equal } x\times y: &\quad \text{map}\,(x^{:A}\times y^{:A})(\text{id}^A)\\
\text{definition of map}: &\quad = \underline{\text{id}\,(y)}\times\underline{\text{id}\,(x)}\\
\text{definition of id}: &\quad = y\times x\neq x\times y \quad .
\end{aligned}$$

We should not have swapped the values in the pair. The correct implementation of `map`:

$$\text{map} \stackrel{\text{def}}{=} x^{:A}\times y^{:A}\to f^{:A\to B}\to f(x)\times f(y) \quad ,$$

preserves information and satisfies the functor laws.

Example 6.1.4.2 shows the type constructor $\text{Vec}_3{}^\bullet$ with an incorrect implementation of `map` that reorders some parts of a tuple and duplicates other parts. The correct implementation preserves the order of parts in a tuple and does not duplicate or omit any parts.

Another case of an an incorrect implementation is the following `map` function for `Option[_]`:

```
def map_bad[A, B]: Option[A] => (A => B) => Option[B] = { _ => _ => None }
```

This function always returns `None`, losing information and violating the identity law. However, we have already seen that `Option[_]` has a different implementation of `map` that satisfies the functor laws.

Similarly, one could define `map` for the `List[_]` type constructor to always return an empty list:

```
def map_bad[A, B]: List[A] => (A => B) => List[B] = { _ => _ => List() }
```

This implementation loses information and violates the functor laws. Of course, the Scala library provides a correct implementation of `map` for `List[_]`.

Example 6.1.4.1 is another situation where an incorrectly implemented `map` violates functor laws.

Functor laws will also be violated when `map` is not fully parametric. For instance, consider an implementation of `fmap[A, B](f)` that checks whether the two type parameters A and B are equal to each other *as types*, and if so, applies the function argument `f` twice. We need to use special features of Scala (type reflection and `TypeTag`) for comparing two type parameters at run time:

```
import scala.reflect.runtime.universe._
def getType[T: TypeTag]: Type = weakTypeOf[T]
def equalTypes[A: TypeTag, B: TypeTag]: Boolean = getType[A] =:= getType[B]
```

```
def fmap_bad[A: TypeTag, B: TypeTag](f: A => B)(oa: Option[A]): Option[B] = oa match {
  case None      =>   None
  case Some(x)   =>                    // If A = B, compute f(f(x)), else compute f(x).
    val z: B = if (equalTypes[A, B]) f(f(x).asInstanceOf[A]) else f(x)
    Some(z)
}
```

Testing shows that this function works as designed:

```
scala> fmap_bad[Int, String](_ + " a")(Some(123))       // Appends " a" once.
res0: Option[String] = Some(123 a)

scala> fmap_bad[String, String](_ + " a")(Some("123"))  // Appends " a" twice.
res1: Option[String] = Some(123 a a)
```

The function `fmap_bad[A, B]` satisfies the identity law but violates the composition law when `A = B`:

```
scala> fmap_bad[String, String](_ + " b")(Some("123 a a"))
res2: Option[String] = Some(123 a a b b)

scala> fmap_bad[String, String](_ + " a b")(Some("123"))
res3: Option[String] = Some(123 a b a b)
```

In all these examples, we *could* implement a `map` function that would obey the laws. It is not precise to say that, e.g., the type constructor `Vec3[_]` is *by itself* a functor: being a functor depends on having a lawful `map` function. Keeping that in mind, we will say that the type constructor `Vec3[_]` "is" a functor, meaning that a suitable lawful implementation of `map` can be found.

Laws hold for some types but not for others The Scala standard library contains `map` methods for the type constructors `Set` (transforming the values in a set) and `Map` (transforming both the keys and values in a dictionary). However, `Set[K]` and `Map[K, V]` fail to be lawful functors with respect to the type parameter `K`. The reason for the failure is complicated. A value of type `Set[K]` represents a set of zero or more values of type `K`, and it is enforced that all values in the set are distinct. So, the correct functionality of `Set` requires us to be able to check whether two values of type `K` are equal. A standard way of comparing values for equality is the `equals` method defined in the Scala library:

```
scala>  List(1, 2, 3).equals(List(1, 2, 3))
res0: Boolean = true

scala>  List(1, 2, 3).equals(List(1, 2, 3, 4))
res1: Boolean = false
```

However, an `equals` operation will work as expected only if it obeys the laws of **identity** (if $x = y$ then $f(x) = f(y)$ for any f), **symmetry** (if $x = y$ then $y = x$), **reflexivity** ($x = x$ for any x), and **transitivity** (if $x = y$ and $y = z$ then $x = z$). In most practical applications, the required type `K` (such as `String` or `Int`) will have a lawful `equals` method. In some cases, however, data types could redefine their `equals` method for application-specific purposes and violate some of the required laws.

Here are two examples of law-breaking (but potentially useful) code for `equals`. The first example[1] is a disjunctive type $A + B$ whose `equals` method allows only values of type $A + \mathbb{0}$ to be equal:

```
final case class OnlyA[A, B](eab: Either[A, B]) {
  override def equals(y: Any): Boolean = (eab, y) match {
    case (Left(a1), OnlyA(Left(a2)))   => a1 == a2  // Values Left(a1) and Left(a2) might be equal.
    case _                             => false     // Never equal unless both are ‘Left‘.
  }
}
```

This implementation of `equals` is mathematically invalid: it violates the reflexivity law ($\forall x. x = x$) because values of the form `OnlyA` are never equal to each other:

```
scala> OnlyA(Right(0)) equals OnlyA(Right(0))
res2: Boolean = false
```

[1] This example is based on a comment by Paweł Szulc at https://gist.github.com/tpolecat/7401433

As a result, the library code of `Set[OnlyA]` will fail to detect that, e.g., several values `OnlyA(Right(0))` are equal. The composition law of functors will fail when intermediate values of that type are used:

```
val f: OnlyA[Int, Int] => Int = { case OnlyA(Left(a)) => a; case OnlyA(Right(a)) => a }
val g: Int => OnlyA[Int, Int] = { a => OnlyA(Right(a)) }
val xs = Seq(0, 0, 0).map(g).toSet

scala> xs.map(f andThen g)  // ‘Set‘ fails to detect identical values.
res3: Set[OnlyA[Int,Int]] = Set(OnlyA(Right(0)), OnlyA(Right(0)), OnlyA(Right(0)))

scala> xs.map(f).map(g)     // ‘Set‘ detects identical values.
res4: Set[OnlyA[Int,Int]] = Set(OnlyA(Right(0)))
```

The second example is a product type $A \times B$ whose `equals` method ignores the part of type B:

```
final case class IgnoreB[A, B](a: A, b: B) {
  override def equals(y: Any): Boolean = y match {
    case IgnoreB(a2, b2)   => a == a2  // Equal as long as the parts of type A are equal.
    case _                 => false    // Never equal to a value of another type (not IgnoreB).
  }
}

scala> IgnoreB(123, "abc") == IgnoreB(123, "def")
res5: Boolean = true
```

As a result, Scala's library code of `Set[IgnoreB]` will fail to detect that some values are different. This violates the functor composition law:

```
val f: IgnoreB[Int, Int] => IgnoreB[Int, Int] = { case IgnoreB(x, y) => IgnoreB(y, x) }  //
    f ⨾ f = id
val xs = Set(IgnoreB(0, 0), IgnoreB(1, 0))

scala> xs.map(f andThen f)    // This is equal to ‘xs‘.
res6: Set[IgnoreB[Int,Int]] = Set(IgnoreB(0,0), IgnoreB(1,0))

scala> xs.map(f).map(f)       // This is not equal to ‘xs‘.
res7: Set[IgnoreB[Int,Int]] = Set(IgnoreB(0,0))
```

The functor laws for a type constructor $L^\bullet$ do not require that the types A, B used in the function:

$$\text{fmap}_L : (A \to B) \to L^A \to L^B$$

should have a mathematically lawful definition of the `equals` method (or of any other operation). The `map` method of a functor $L^\bullet$ must be **lawful**, i.e., must satisfy the functor laws (6.2)–(6.3) for all types A, B. The functor laws must hold even if a type A's implementation of some operations violate some other laws. For this reason, `Set[_]` cannot be considered a functor in a rigorous sense.

The `map` method for dictionaries has a similar problem: the keys of a dictionary must be distinct and will be compared using the `equals` method. So, the `map` method for `Map[K, V]` will violate the functor laws unless the type `K` has a lawful `equals` method.

The Scala standard library still provides the `map` and `flatMap` methods for sets `Set[K]` and dictionaries `Map[K, V]` because most applications will use types `K` that have lawful `equals` operations, and the functor laws will hold.

6.1.7 Contrafunctors

As we have seen in Section 6.1.6, the type constructor $H^\bullet$ defined by $H^A \stackrel{\text{def}}{=} A \to \text{Int}$ is not a functor because it is impossible to implement the type signature of `map` as a fully parametric function:

$$\text{map}^{A,B} : (A \to \text{Int}) \to (A \to B) \to B \to \text{Int} \quad .$$

To see why, begin writing the code with a typed hole:

$$\text{map}\,(h^{:A\to\text{Int}})(f^{:A\to B})(b^{:B}) = ???^{:\text{Int}} \quad .$$

The only way of returning an `Int` in fully parametric code is by applying the function $h^{:A\to\text{Int}}$. Since h consumes (rather than wraps) values of type A, we have no values of type A and cannot apply the function $h^{:A\to\text{Int}}$. However, it would be possible to apply a function of type $B \to A$ since a value of type B is given as one of the curried arguments, $b^{:B}$. So, we can implement a function called `contramap` with a different type signature where the function type is $B \to A$ instead of $A \to B$:

$$\text{contramap}^{A,B} : (A \to \text{Int}) \to (B \to A) \to B \to \text{Int} \quad .$$

The implementation of this function is written in the code notation as:

$$\text{contramap} \stackrel{\text{def}}{=} h^{:A\to\text{Int}} \to f^{:B\to A} \to (f \,⨾\, h)^{:B\to\text{Int}} \quad ,$$

and the corresponding Scala code is:

```
def contramap[A, B](h: H[A])(f: B => A): H[B] = { f andThen h }
```

Flipping the order of the curried arguments in `contramap`, we define `cmap` as:

$$\begin{aligned} \text{cmap}^{A,B} &: (B \to A) \to H^A \to H^B \quad , \\ \text{cmap} &\stackrel{\text{def}}{=} f^{:B\to A} \to h^{:A\to\text{Int}} \to (f \,⨾\, h)^{:B\to\text{Int}} \quad . \end{aligned} \tag{6.5}$$

The type signature of `cmap` has the form of a "reverse lifting": functions of type `B => A` are lifted into the type `H[A] => H[B]`. The Scala code for `cmap` is:

```
def cmap[A, B](f: B => A): H[A] => H[B] = { h => f andThen h }
```

We can check that this `cmap` satisfies two laws analogous to the functor laws:

$$\begin{aligned} \text{identity law}: &\quad \text{cmap}^{A,A}(\text{id}^{:A\to A}) = \text{id}^{:H^A\to H^A} \quad , \\ \text{composition law}: &\quad \text{cmap}^{A,B}(f^{:B\to A}) \,⨾\, \text{cmap}^{B,C}(g^{:C\to B}) = \text{cmap}^{A,C}(g \,⨾\, f) \quad . \end{aligned}$$

$$\begin{tikzcd} & H^B \arrow[dr, "\text{cmap}_H(g^{:C\to B})"] & \\ H^A \arrow[ur, "\text{cmap}_H(f^{:B\to A})"] \arrow[rr, "\text{cmap}_H(g^{:C\to B} \,⨾\, f^{:B\to A})"'] & & H^C \end{tikzcd}$$

Since the function argument $f^{:B\to A}$ has the reverse order of types, the composition law needs the reverse order of composition $(g \,⨾\, f)$ on one side; in this way, all types match. To verify the identity law:

$$\begin{aligned} \text{expect to equal id}: &\quad \text{cmap}\,(\text{id}) \\ \text{use Eq. (6.5)}: &\quad = h \to \underline{(\text{id} \,⨾\, h)} \\ \text{definition of id}: &\quad = (h \to h) = \text{id} \quad . \end{aligned}$$

To verify the composition law:

$$\begin{aligned} \text{expect to equal cmap}\,(g \,⨾\, f): &\quad \text{cmap}\,(f) \,⨾\, \text{cmap}\,(g) \\ \text{use Eq. (6.5)}: &\quad = (h \to (f \,⨾\, h)) \,⨾\, (\underline{h} \to (g \,⨾\, \underline{h})) \\ \text{rename } h \text{ to } k \text{ for clarity}: &\quad = (h \to (f \,⨾\, h)) \,⨾\, (k \to (g \,⨾\, k)) \\ \text{compute composition}: &\quad = (h \to g \,⨾\, f \,⨾\, h) \\ \text{use Eq. (6.5)}: &\quad = \text{cmap}\,(g \,⨾\, f) \quad . \end{aligned}$$

A type constructor with a fully parametric `cmap` is called a **contrafunctor** if the identity and the composition laws are satisfied.

Example 6.1.7.1 Show that the type constructor $D^A \stackrel{\text{def}}{=} A \to A \to \text{Int}$ is a contrafunctor.

Solution The required type signature for `contramap` is:

```
def contramap[A, B](d: A => A => Int)(f: B => A): B => B => Int = ???
```

We begin implementing `contramap` by writing code with a typed hole:

$$\text{contramap}^{A,B} \stackrel{\text{def}}{=} d^{:A\to A\to \text{Int}} \to f^{:B\to A} \to b_1^{:B} \to b_2^{:B} \to ???^{:\text{Int}} \quad .$$

To fill the typed hole, we need to compute a value of type `Int`. The only possibility is to apply d to two curried arguments of type A. We have two curried arguments of type B. So we apply $f^{:B\to A}$ to those arguments, obtaining two values of type A. To avoid information loss, we need to preserve the order of the curried arguments. So, the resulting expression is:

$$\text{contramap}^{A,B} \stackrel{\text{def}}{=} d^{:A\to A\to \text{Int}} \to f^{:B\to A} \to b_1^{:B} \to b_2^{:B} \to d\,(f(b_1))\,(f(b_2)) \quad .$$

The corresponding Scala code is

```
def contramap[A, B](d: A => A => Int)(f: B => A): B => B => Int = { b1 => b2 => d(f(b1))(f(b2)) }
```

To verify the laws, it is easier to use the equivalent `cmap` defined by:

$$\text{cmap}^{A,B}(f^{:B\to A}) \stackrel{\text{def}}{=} d^{:A\to A\to \text{Int}} \to b_1^{:B} \to b_2^{:B} \to d\,(f(b_1))\,(f(b_2)) \quad . \tag{6.6}$$

To verify the identity law:

$$\begin{aligned}
\text{expect to equal id}: &\quad \text{cmap}\,(\text{id}) \\
\text{use Eq. (6.6)}: &\quad = d \to b_1 \to b_2 \to d\underline{(\text{id}\,(b_1))(\text{id}\,(b_2))} \\
\text{definition of id}: &\quad = d \to \underline{b_1 \to b_2 \to d(b_1)(b_2)} \\
\text{simplify curried function}: &\quad = (d \to d) = \text{id} \quad .
\end{aligned}$$

To verify the composition law, we rewrite its left-hand side into the right-hand side:

$$\begin{aligned}
& \text{cmap}\,(f) \,⨾\, \text{cmap}\,(g) \\
\text{use Eq. (6.6)}: &\quad = (d \to b_1 \to b_2 \to d\,(f(b_1))\,(f(b_2))) \,⨾\, (\underline{d} \to b_1 \to b_2 \to \underline{d}\,(g(b_1))\,(g(b_2))) \\
\text{rename } d \text{ to } e: &\quad = (d \to b_1 \to b_2 \to d\,(f(b_1))\,(f(b_2))) \,⨾\, (e \to b_1 \to b_2 \to e\,(g(b_1))\,(g(b_2))) \\
\text{compute composition}: &\quad = d \to b_1 \to b_2 \to d\,(f(g(b_1)))\,(f(g(b_2))) \\
\text{use Eq. (6.6)}: &\quad = \text{cmap}\,(b \to f(g(b))) \\
\text{definition of } (g ⨾ f): &\quad = \text{cmap}\,(g ⨾ f) \quad .
\end{aligned}$$

The type H^A represents a function that consumes a value of type A to produce an integer; the type D^A represents a curried function consuming *two* values of type A. These examples suggest the heuristic view that contrafunctors "consume" data while functors "wrap" data. By looking at the position of a given type parameter in a type expression such as $A \times \text{Int}$ or $A \to A \to \text{Int}$, we can see whether the type parameter is "consumed" or "wrapped": A type parameter to the left of a function arrow is being "consumed"; a type parameter to the right of a function arrow (or used without a function arrow) is being "wrapped". We will make this intuition precise in Section 6.2.

Type constructors that are not contrafunctors A type constructor that both consumes *and* wraps data is neither a functor nor a contrafunctor. An example of such a type constructor is:

$$N^A \stackrel{\text{def}}{=} (A \to \text{Int}) \times (\mathbb{1} + A) \quad .$$

We can implement neither `map` nor `contramap` for $N^\bullet$. Intuitively, the type parameter A is used both to the left of a function arrow (being "consumed") and outside of a function (being "wrapped").

Unfunctors (type constructors that lack full parametricity) also cannot be contrafunctors because the required type signature for `contramap` cannot be implemented by a fully parametric function. To show that `ServerAction[_]` cannot be a contrafunctor, we can straightforwardly adapt the reasoning used in Section 6.1.6 when we showed that `ServerAction[_]` cannot be a functor.

6.1.8 Subtyping, covariance, and contravariance

Ordinarily, applying a function of type $Q \to R$ to a value of type P is an error:

```
def h(q: Q): R = ???
val p: P = ???
h(p)          // Type error: the argument of h must have type Q, not P.
```

However, the Scala compiler will admit this kind of code when P is a "subtype" of Q. We may imagine that the compiler automatically converts values of type P into values of type Q using a fixed "conversion function" (of type $P \to Q$) that is somehow already available.

It is convenient to *define* subtyping through the existence of a conversion function:

Definition 6.1.8.1 A type P is called a **subtype** of a type Q (denoted $P \lesssim Q$) if there exists a designated **subtype conversion function** of type $P \to Q$. The compiler must automatically insert that conversion function whenever a value of type P is used instead of a required value of type Q.

Programming languages that support subtyping usually do not define the subtype conversion functions explicitly. Nevertheless, whenever $P \lesssim Q$ we can always recover the underlying conversion function $c^{:P\to Q}$. We just apply an identity function $\text{id}^{:Q\to Q}$ to a value of type P (since we may use values of type P instead of Q in any expression). Then we find: $c^{:P\to Q} \stackrel{\text{def}}{=} p^{:P} \to \text{id}^{:Q\to Q}(p)$.

In most programming languages, the allowed subtype conversion functions are chosen to be *identity functions* that only reassign the types. Let us look at some examples of subtyping of that kind.

The first example uses disjunctive types. Consider this type definition:

```
sealed trait AtMostTwo
final case class Zero()                extends AtMostTwo
final case class One(x: Int)           extends AtMostTwo
final case class Two(x: Int, y: Int)   extends AtMostTwo
```

$$\text{AtMostTwo} \stackrel{\text{def}}{=} \mathbb{1} + \text{Int} + \text{Int} \times \text{Int} \quad .$$

Each of the case classes (`Zero`, `One`, and `Two`) defines a type that is a subtype of `AtMostTwo` whose conversion function is an identity function. To see that, let us implement the conversion functions from each of the three case classes to `AtMostTwo`. The required functions reassign the types but perform no transformations on the data:

```
def f0: Zero => AtMostTwo  = { case Zero()      => Zero()    }
def f1: One  => AtMostTwo  = { case One(x)      => One(x)    }
def f2: Two  => AtMostTwo  = { case Two(x, y)   => Two(x, y) }
```

The implementations of these subtype conversions indeed look like the code of identity functions. In the matrix code notation, we write:

$$f_0 \stackrel{\text{def}}{=} \begin{array}{|c||ccc|} & \text{Zero} & \text{One} & \text{Two} \\ \hline \text{Zero} & \text{id} & \mathbb{0} & \mathbb{0} \end{array} \quad , \quad \text{so} \quad f_0(1^{:\text{Zero}}) \stackrel{\text{def}}{=} 1^{:\text{Zero}} + \mathbb{0}^{:\text{One}} + \mathbb{0}^{:\text{Two}} \quad ,$$

$$f_1 \stackrel{\text{def}}{=} \begin{array}{|c||ccc|} & \text{Zero} & \text{One} & \text{Two} \\ \hline \text{One} & \mathbb{0} & \text{id} & \mathbb{0} \end{array} \quad , \quad \text{so} \quad f_1(x^{:\text{One}}) \stackrel{\text{def}}{=} \mathbb{0}^{:\text{Zero}} + x^{:\text{One}} + \mathbb{0}^{:\text{Two}} \quad ,$$

$$f_2 \stackrel{\text{def}}{=} \begin{array}{|c||ccc|} & \text{Zero} & \text{One} & \text{Two} \\ \hline \text{Two} & \mathbb{0} & \mathbb{0} & \text{id} \end{array} \quad , \quad \text{so} \quad f_2\left((x \times y)^{:\text{Two}}\right) \stackrel{\text{def}}{=} \mathbb{0}^{:\text{Zero}} + \mathbb{0}^{:\text{One}} + (x \times y)^{:\text{Two}} \quad .$$

The matrix notation emphasizes that the code consists of identity functions with reassigned types.

Another example is subtyping between *function* types. Consider these types P and Q:

```
type P = (AtMostTwo => Int)
type Q = (Two => Int)
```

We can convert a function f of type P into a function g of type Q because f includes all the information necessary to define g. The Scala code for that subtype conversion is:

```
def p2q(f: P): Q = { t: Two => f(t) }
```

This is written in the code notation as:

$$\text{p2q}\,(f^{:\text{AtMostTwo}\to\text{Int}}) \stackrel{\text{def}}{=} t^{:\text{Two}} \to f(t) \quad .$$

The code $\text{p2q}\,(f) = t^{:\text{Two}} \to f(t)$ is almost the same as $\text{p2q}\,(f) = f$, except that f is applied to values of type `Two`. So, the implementation of `p2q` is an identity function that reassigns a type.

In these cases, it is easy for the compiler to insert the appropriate conversion functions automatically whenever necessary. Any function that consumes an argument of type Q will then automatically apply to arguments of type P. The compiler does not actually need to insert any new code, since the identity functions do not perform any data transformations.

To achieve this, we need to declare to the Scala compiler that a certain type is a subtype of another. This can be done in three ways depending on the situation: First, one may declare a class that "`extends`" another class. Second, one may declare type parameters with a "variance annotation" such as `L[+A]` or `L[-B]`. Third, one may declare type parameters with a "subtyping annotation" (`A <: B`).

Parts of a disjunctive type are subtypes of that type since they are declared via the keyword "`extends`". A function with argument of type `AtMostTwo` can be applied to a value of type `Two` with no additional code written by the programmer:

```
def head: AtMostTwo => Option[Int] = {
  case Zero()       => None
  case One(x)       => Some(x)
  case Two(x, y)    => Some(x)
}

scala> head(Two(10, 20))
res0: Option[Int] = Some(10)
```

We may imagine that the compiler automatically uses the subtype conversion function `f2` shown above to convert a value of the type `Two` into a value of the type `AtMostTwo`. Since the code of `f2` is equivalent to an identity function, the subtype conversion does not change any data and only reassigns some types. So, the compiler does not need to insert any additional machine code, and the subtype conversion does not lead to any decrease in performance.

The void type (`Nothing`, denoted by $\mathbb{0}$ in the type notation) is special: any type A is a subtype of $\mathbb{0}$. The reason is that the type $\mathbb{0}$ has *no* values, so a conversion function $\texttt{absurd} : \mathbb{0} \to A$ (shown in Example 5.3.4.2) never actually needs to be applied. The Scala compiler recognizes this automatically.

Let us now consider subtyping for type constructors. If a type constructor L^A is a functor, we can use its fmap_L method to lift a subtype conversion function $f : P \to Q$ into:

$$\text{fmap}_L(f) : L^P \to L^Q \quad ,$$

which gives a subtype conversion function from L^P to L^Q. So, it makes sense to define subtyping as `L[P] <: L[Q]` between the types L^P and L^Q. If f is an identity function then $\text{fmap}_L(f)$ is also an identity function because $\text{fmap}_L(\text{id}) = \text{id}$ (the functor L's identity law).

Note that, for any functor F, the type F^A is a subtype of $F^{\mathbb{0}}$ because any type A is a subtype of $\mathbb{0}$.

If a type constructor H^A is a contrafunctor, a subtype conversion function $f^{:P\to Q}$ is lifted to:

$$\text{cmap}_H(f) : H^Q \to H^P \quad ,$$

showing that H^Q is a subtype of H^P. The identity law of the contrafunctor H,

$$\text{cmap}_H(\text{id}) = \text{id} \quad ,$$

shows that the lifted conversion function is again an identity function (with reassigned types).

A type constructor F is called **covariant** if F^A is a subtype of F^B whenever A is a subtype of B. A **contravariant** type constructor H performs subtyping in the opposite direction: H^B is a subtype of H^A. As we just showed, all functors are covariant type constructors, and all contrafunctors are contravariant type constructors.[2] However, the Scala compiler does not automatically determine whether a given type constructor `F[A]` is covariant with respect to a given type parameter `A`. To indicate the covariance property, the programmer needs to use a **variance annotation**, which looks

[2]The name "**contrafunctor**" is used in this book as a shortened form of "contravariant functor".

like `F[+A]`, on the relevant type parameters. For example, the type constructor `Counted[A]` defined in Section 6.1.4 is a functor and so is covariant in its type parameter `A`. If we use the variance annotation `Counted[+A]` in the definition, Scala will automatically consider the type `Counted[Two]` as a subtype of `Counted[AtMostTwo]`. Then we will be able to apply any function to a value of type `Counted[Two]` as if it had type `Counted[AtMostTwo]`:

```
final case class Counted[+A](n: Int, a: A)

def total(c: Counted[AtMostTwo]): Int = c match {
  case Counted(n, Zero())      => 0
  case Counted(n, One(_))      => n
  case Counted(n, Two(_, _))   => n * 2
}

scala> total(Counted(2, Two(10, 20)))
res1: Int = 4
```

The contravariance property for contrafunctors can be annotated using the syntax `F[-A]`.

A type constructor with several type parameters may be covariant with respect to some of them and contravariant with respect to others. Intuitively, the data type "wraps" covariant types but "consumes" contravariant ones. So far, we have seen examples suggesting that a type parameter to the right of function arrows, or without function arrows, is "wrapped" (used in a **covariant position**). Type parameters to the left of function arrows are "consumed" (used in a **contravariant position**). The next examples confirm this intuition, which will be made rigorous in Section 6.2.5.

6.1.9 Solved examples: implementation of functors and contrafunctors

Example 6.1.9.1 Consider this implementation of `map` for the type constructor `Option[_]`:

```
def map[A, B](oa: Option[A])(f: A => B): Option[B] = oa match {
  case None            => None
  case Some(x: Int)    => Some(f((x+1).asInstanceOf[A]))
  case Some(x)         => Some(f(x))
}
```

This code performs a non-standard computation if the type parameter `A` is set to `Int`. Show that this implementation of `map` violates the functor laws.

Solution If the type parameter `A` is not `Int`, or if the argument `oa` is `None`, the given code is the same as the standard (correct) implementation of `map` for `Option`. The function does something non-standard when, e.g., `oa == Some(123)`. Substitute this value of `oa` into the identity law, `map(oa)(identity) == oa`, and compute symbolically (using Scala syntax):

```
map(oa)(identity) == Some(identity((123+1).asInstanceOf[Int])) == Some(124) != oa
```

This shows a violation of the functor identity law.

Example 6.1.9.2 Define case classes and implement `fmap` for the given type constructors:

(a) $\text{Data}^A \stackrel{\text{def}}{=} \text{String} + A \times \text{Int} + A \times A \times A$.

(b) $\text{Data}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times (\text{Int} \times \text{String} + A)$.

(c) $\text{Data}^A \stackrel{\text{def}}{=} (\text{String} \to \text{Int} \to A) \times A + (\text{Boolean} \to \text{Double} \to A) \times A$.

Solution **(a)** Begin by defining a case class for each part of the disjunctive type:

```
sealed trait Data[A]
final case class Message[A](message: String)  extends Data[A]
final case class Have1[A](x: A, n: Int)       extends Data[A]
final case class Have3[A](x: A, y: A, z: A)   extends Data[A]
```

The names `Message`, `Have1`, `Have3`, `n`, `x`, `y`, `z` are chosen arbitrarily.

The function `fmap` must have this type signature:

$$\text{fmap}^{A,B} : (A \to B) \to \text{Data}^A \to \text{Data}^B \quad , \qquad \text{fmap}\,(f^{:A\to B}) = ??? \quad .$$

To implement `fmap` correctly, we need to transform each part of the disjunctive type `Data[A]` into the corresponding part of `Data[B]` without loss of information. To clarify where the transformation $f^{:A\to B}$

need to be applied, let us write the type notation for Data^A and Data^B side by side:

$$\text{Data}^A \stackrel{\text{def}}{=} \text{String} + A \times \text{Int} + A \times A \times A \quad ,$$
$$\text{Data}^B \stackrel{\text{def}}{=} \text{String} + B \times \text{Int} + B \times B \times B \quad .$$

Now it is clear that we need to apply f to each value of type A present in Data^A, preserving the order of values. The corresponding Scala code is:

```
def fmap[A, B](f: A => B): Data[A] => Data[B] = {
  case Message(message)   => Message(message)
  case Have1(x, n)        => Have1(f(x), n)
  case Have3(x, y, z)     => Have3(f(x), f(y), f(z))
}
```

(b) It is convenient to define the disjunctive type $\text{Int} \times \text{String} + A$ separately as P^A:

```
sealed trait P[A]
final case class Message[A](code: Int, message: String)  extends P[A]
final case class Value[A](x: A)                          extends P[A]
```

Now we notice that the type expression $(\mathbb{1} + ...)$ can be encoded via the standard `Option` type. So, the Scala code for Data^A is:

```
final case class Data[A](d: Option[(A, P[A])])
```

To make sure we implement `fmap` correctly, let us write out the type expressions:

$$\text{Data}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times (\text{Int} \times \text{String} + A) \quad ,$$
$$\text{Data}^B \stackrel{\text{def}}{=} \mathbb{1} + B \times (\text{Int} \times \text{String} + B) \quad .$$

Now we can transform Data^A into Data^B by applying $f^{:A\to B}$ at the correct places:

```
def fmap[A, B](f: A => B): Data[A] => Data[B] = {
  case Data(None)                                  => Data(None)
  case Data(Some((x, Message(code, message))))     => Data(Some((f(x), Message(code, message))))
  case Data(Some((x, Value(y))))                   => Data(Some((f(x), Value(f(y)))))
}
```

When deeply nested patterns become hard to read, we may handle the nested structure separately:

```
def fmap[A, B](f: A => B): Data[A] => Data[B] = {
  case Data(None)           => Data(None)
  case Data(Some((x, p)))   =>
      val newP: P[B] = p match {
        case Message(code, message)   => Message(code, message)
        case Value(x)                 => Value(f(x))
      }
      Data(Some((f(x), newP)))
}
```

(c) Since the type structures $(\text{String} \to \text{Int} \to A) \times A$ and $(\text{Boolean} \to \text{Double} \to A) \times A$ have a similar pattern, let us define a new parameterized type:

$$Q^{X,Y,A} \stackrel{\text{def}}{=} (X \to Y \to A) \times A \quad ,$$

and rewrite the given type expression as:

$$\text{Data}^A \stackrel{\text{def}}{=} Q^{\text{String},\text{Int},A} + Q^{\text{Boolean},\text{Double},A} \quad .$$

It is then convenient to define `Data[A]` using the standard disjunctive type `Either`:

```
type Q[X, Y, A] = (X => Y => A, A)
type Data[A] = Either[Q[String, Int, A], Q[Boolean, Double, A]]
```

The code will be clearer if we implement `fmap` separately for $Q^\bullet$ and $\text{Data}^\bullet$.

To derive the code of `fmap` for $Q^\bullet$, we begin with the type signature:

$$\text{fmap}_Q^{A,B} : (A \to B) \to (X \to Y \to A) \times A \to (X \to Y \to B) \times B \quad ,$$

and start writing the code using typed holes:

$$\text{fmap}_Q(f^{:A\to B}) \stackrel{\text{def}}{=} g^{:X\to Y\to A} \times a^{:A} \to ???^{:X\to Y\to B} \times ???^{:B} \quad .$$

The typed hole $???^{:B}$ is filled by $f(a)$. To fill the remaining type hole, we write:

$$\begin{aligned} & ???^{:X\to Y\to B} \\ & = x^{:X} \to y^{:Y} \to \underline{???^{:B}} \\ & = x^{:X} \to y^{:Y} \to f(???^{:A}) \quad . \end{aligned}$$

It would be wrong to fill the typed hole $???^{:A}$ by $a^{:A}$ because, to preserve information, a value of type $X \to Y \to B$ should be computed using the given data g of type $X \to Y \to A$. So, we write:

$$???^{:X\to Y\to B} = x^{:X} \to y^{:Y} \to f(g(x)(y)) \quad .$$

The corresponding Scala code is:

```
def fmap_Q[A, B, X, Y](f: A => B): Q[X, Y, A] => Q[X, Y, B] = {
  case (g, a) => (x => y => f(g(x)(y)), f(a))
  // Could also write the code as
  //   case (g, a) => (x => g(x) andThen f, f(a))
}
```

Finally, we can write the code for $\text{fmap}_{\text{Data}}$:

```
def fmap_Data[A, B](f: A => B): Data[A] => Data[B] = {
  case Left(q)    => Left(fmap_Q(f)(q))
  case Right(q)   => Right(fmap_Q(f)(q))
}
```

The Scala compiler will automatically infer the type parameters required by `fmap_Q` and check that all types match. With all inferred types written out, the code above becomes:

```
def fmap_Data[A, B](f: A => B): Data[A] => Data[B] = {
  case Left(q: Q[String, Int, A])            =>
        Left[Q[String, Int, B]](fmap_Q[A, B, String, Int](f)(q))
  case Right(q: Q[Boolean, Double, A])       =>
        Right[Q[Boolean, Double, B]](fmap_Q[A, B, Boolean, Double](f)(q))
}
```

When types become complicated, it may help to write out some of the type parameters in the code.

Example 6.1.9.3 Decide whether these types are functors or contrafunctors, and implement `fmap` or `cmap` as appropriate:

(a) $\text{Data}^A \stackrel{\text{def}}{=} (\mathbb{1} + A \to \text{Int}) + (A \to A \times A \to \text{String}) \quad .$

(b) $\text{Data}^{A,B} \stackrel{\text{def}}{=} (A + B) \times ((A \to \text{Int}) \to B) \quad .$

Solution **(a)** The type constructor Data^A is defined in Scala as:

```
type Data[A] = Either[Option[A] => Int, A => ((A, A)) => String]
```

The type parameter A is always located to the left of function arrows. So, Data^A *consumes* values of type A, and we expect that Data^A is a contrafunctor. Indeed, we can implement `cmap`:

```
def cmap[A, B](f: B => A): Data[A] => Data[B] = {
  case Left(oa2Int)        => Left(b => oa2Int(b.map(f)))
  case Right(a2aa2Str)     => Right( b1 => { case (b2, b3) => a2aa2Str(f(b1))((f(b2), f(b3))) } )
}
```

(b) The type constructor $\text{Data}^{A,B}$ has two type parameters, and so we need to answer the question separately for each of them. Write the Scala type definition as:

```
type Data[A, B] = (Either[A, B], (A => Int) => B)
```

Begin with the type parameter A and notice that a value of type $\text{Data}^{A,B}$ possibly contains a value of type A within `Either[A, B]`. In other words, A is "wrapped", i.e., it is in a covariant position within the first part of the tuple. It remains to check the second part of the tuple, which is a higher-order function of type $(A \to \text{Int}) \to B$. That function consumes a function of type $A \to \text{Int}$, which in turn consumes a value of type A. Consumers of A are contravariant in A, but it turns out that a "consumer of a consumer of A" is *covariant* in A. So we expect to be able to implement `fmap` that applies to the type parameter A of $\text{Data}^{A,B}$. Renaming the type parameter B to Z for clarity, we write the type signature for `fmap` like this:

$$\text{fmap}^{A,C,Z} : (A \to C) \to (A + Z) \times ((A \to \text{Int}) \to Z) \to (C + Z) \times ((C \to \text{Int}) \to Z) \quad .$$

We need to transform each part of the tuple separately. Transforming $A + Z$ into $C + Z$ is straightforward via the function:

$$\begin{array}{c||cc|} & C & Z \\ \hline A & f & \mathbb{0} \\ Z & \mathbb{0} & \text{id} \end{array} \quad .$$

This code notation corresponds to the following Scala code:

```
{
  case Left(x)     => Left(f(x))
  case Right(z)    => Right(z)
}
```

To derive code transforming $(A \to \text{Int}) \to Z$ into $(C \to \text{Int}) \to Z$, we use typed holes:

$$\begin{aligned}
& f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to \underline{???^{:(C\to\text{Int})\to Z}} \\
\text{nameless function}: \quad & = f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to p^{:C\to\text{Int}} \to \underline{???^{:Z}} \\
\text{get a } Z \text{ by applying } g: \quad & = f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to p^{:C\to\text{Int}} \to g(\underline{???^{:A\to\text{Int}}}) \\
\text{nameless function}: \quad & = f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to p^{:C\to\text{Int}} \to g(a^{:A} \to \underline{???^{:\text{Int}}}) \\
\text{get an Int by applying } p: \quad & = f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to p^{:C\to\text{Int}} \to g(a^{:A} \to p(\underline{???^{:C}})) \\
\text{get a } C \text{ by applying } f: \quad & = f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to p^{:C\to\text{Int}} \to g(a^{:A} \to p(f(\underline{???^{:A}}))) \\
\text{use argument } a^{:A}: \quad & = f \to g \to p \to g(a \to p(f(a)) \quad .
\end{aligned}$$

In the resulting Scala code for `fmap`, we write out some types for clarity:

```
def fmapA[A, Z, C](f: A => C): Data[A, Z] => Data[C, Z] = {
  case (e: Either[A, Z], g: ((A => Int) => Z)) =>
    val newE: Either[C, Z] = e match {
      case Left(x)     => Left(f(x))
      case Right(z)    => Right(z)
    }
    val newG: (C => Int) => Z = { p => g(a => p(f(a))) }
    (newE, newG) // This has type Data[C, Z].
}
```

This suggests that $\text{Data}^{A,Z}$ is covariant with respect to the type parameter A. The results of Section 6.2 will show rigorously that the functor laws hold for this implementation of `fmap`.

The analysis is simpler for the type parameter B because it is only used in covariant positions, never to the left of function arrows. So we expect $\text{Data}^{A,B}$ to be a functor with respect to B. Implementing the corresponding `fmap` is straightforward:

```
def fmapB[Z, B, C](f: B => C): Data[Z, A] => Data[Z, B] = {
  case (e: Either[Z, B], g: ((Z => Int) => B)) =>
    val newE: Either[Z, B] = e match {
      case Left(x)    => Left(f(x))
      case Right(z)   => Right(z)
    }
    val newG: (C => Int) => Z = { p => g(a => p(f(a))) }
    (newE, newG) // This has type Data[C, Z].
}
```

The code indicates that $\text{Data}^{A,B}$ is a functor with respect to both A and B.

Example 6.1.9.4 Rewrite the following code in the type notation; identify covariant and contravariant type usages; verify with the Scala compiler that the variance annotations are correct:

```
sealed trait Coi[A, B]
final case class Pa[A, B](b: (A, B), c: B => Int)     extends Coi[A, B]
final case class Re[A, B](d: A, e: B, c: Int)         extends Coi[A, B]
final case class Ci[A, B](f: String => A, g: B => A)  extends Coi[A, B]
```

Solution The type notation puts all parts of the disjunctive type into a single type expression:

$$\text{Coi}^{A,B} \stackrel{\text{def}}{=} A \times B \times (B \to \text{Int}) + A \times B \times \text{Int} + (\text{String} \to A) \times (B \to A) \quad .$$

Now find which types are wrapped and which are consumed in this type expression. The type parameter A is wrapped and never consumed, but B is both wrapped and consumed (in $B \to A$). So, the type constructor `Coi` is covariant in A but neither covariant nor contravariant in B. We can check this by compiling the corresponding Scala code with variance annotations:

```
sealed trait Coi[+A, B]
case class Pa[+A, B](b: (A, B), c: B => Int)     extends Coi[A, B]
case class Re[+A, B](d: A, e: B, c: Int)         extends Coi[A, B]
case class Ci[+A, B](f: String => A, g: B => A)  extends Coi[A, B]
```

We could also replace the fixed types `Int` and `String` by type parameters `N` and `S`. A similar analysis shows that `N` is in covariant positions while `S` is in a contravariant position. We can then check that the Scala compiler accepts the following type definition with variance annotations:

```
sealed trait Coi2[+A, B, +N, -S]
case class Pa2[+A, B, +N, -S](b: (A, B), c: B => N)  extends Coi[A, B, N, S]
case class Re2[+A, B, +N, -S](d: A, e: B, c: N)      extends Coi[A, B, N, S]
case class Ci2[+A, B, +N, -S](f: S => A, g: B => A)  extends Coi[A, B, N, S]
```

6.1.10 Exercises: implementation of functors and contrafunctors

Exercise 6.1.10.1 An implementation of `fmap` for the type constructor `Either[A, A]` is given as:

```
def fmap[A, B](f: A => B): Either[A, A] => Either[B, B] = {
  case Left(a)         => Right(f(a))
  case Right(a: Int)   => Left(f(a + 1))
  case Right(a)        => Left(f(a))
}
```

Show that this implementation of `fmap` violates the functor laws. Implement `fmap` correctly for this type constructor and the given type signature.

Exercise 6.1.10.2 Define these type constructors in Scala, decide whether they are covariant or contravariant, and implement `fmap` or `cmap` as appropriate:

(a) $\text{Data}^A \stackrel{\text{def}}{=} (\mathbb{1} + A) \times (\mathbb{1} + A) \times \text{String} \quad .$

(b) $\text{Data}^A \stackrel{\text{def}}{=} (A \to \text{Boolean}) \to A \times (\text{Int} + A) \quad .$

(c) $\text{Data}^{A,B} \stackrel{\text{def}}{=} (A \to \text{Boolean}) \times (A + B \to \text{Int}) \quad .$

(d) $\text{Data}^A \stackrel{\text{def}}{=} (\mathbb{1} + (A \to \text{Boolean})) \to \mathbb{1} + (A \to \text{Int}) \to \text{Int} \quad .$

(e) $\text{Data}^B \stackrel{\text{def}}{=} (B + (\text{Int} \to B)) \times (B + (\text{String} \to B)) \quad .$

Exercise 6.1.10.3 Rewrite the following code in the type notation; find covariant and contravariant positions of type parameters; add variance annotations and verify that the revised code compiles:

```
sealed trait S[A, B]
final case class P[A, B](a: A, b: B, c: Int)        extends S[A, B]
final case class Q[A, B](d: Int => A, e: Int => B)  extends S[A, B]
final case class R[A, B](f: A => A, g: A => B)      extends S[A, B]
```

6.2 Laws and structure

A type constructor is a functor if it admits a lawful `map` function. How can we recognize quickly whether a given type constructor is a functor or a contrafunctor? For example, consider the type constructor $Z^{A,R}$ defined by:

$$Z^{A,R} \stackrel{\text{def}}{=} ((A \to A \to R) \to R) \times A + (\mathbb{1} + R \to A + \text{Int}) + A \times A \times \text{Int} \times \text{Int} \quad . \tag{6.7}$$

Is $Z^{A,R}$ a functor with respect to A, or perhaps with respect to R? To answer these questions, we will systematically build up various type expressions for which the functor or contrafunctor laws hold.

6.2.1 Functor laws in the lifting notation

We begin by introducing a more convenient notation for the functor laws. The laws (6.2)–(6.3) were defined in terms of the function `fmap`. When written in terms of the curried function `map`, the structure of the laws becomes less clear:

$$\text{map}_L(x^{:L^A})(\text{id}^{:A\to A}) = x \quad ,$$
$$\text{map}_L(x^{:L^A})(f^{:A\to B} \fatsemi g^{:B\to C}) = \text{map}_L\big(\text{map}_L(x)(f)\big)(g) \quad .$$

The laws are easier to read when using `map` as a class method:

```
x.map(identity) == x
x.map(f).map(g) == x.map(f andThen g)
```

To take advantage of this syntax, we use the **pipe notation** with the symbol $\triangleright$ ("pipe"). The notation $x \triangleright \text{fmap}(f)$ means `x.map(f)`. We then write the functor laws as:

$$x \triangleright \text{fmap}_L(\text{id}) = x \quad ,$$
$$x \triangleright \text{fmap}_L(f) \triangleright \text{fmap}_L(g) = x \triangleright \text{fmap}_L(f \fatsemi g) \quad .$$

In later chapters of this book, we will find that the `map` method (equivalently, the `fmap` function) is used so often in different contexts that the notation $\text{fmap}_L(f)$ becomes too verbose. To make code expressions visually easy to manipulate, we need a shorter notation. At the same time, it is important to show clearly the relevant type constructor L. Dropping the symbol L can lead to errors, since it will be sometimes unclear what type constructors are involved in an expression such as `x.map(f).map(g)` and whether we are justified in replacing that expression with `x.map(f andThen g)`.

For these reasons, we introduce the superscript notation $^{\uparrow L}$ (pronounced "lifted to L"). For any function f of type $A \to B$, we define the function $f^{\uparrow L}$ of type $L^A \to L^B$:

$$(f^{:A\to B})^{\uparrow L} : L^A \to L^B \quad , \qquad f^{\uparrow L} \stackrel{\text{def}}{=} \text{fmap}_L(f) \quad .$$

Now we can choose the notation for lifted functions according to convenience:

$$\text{map}_L(x)(f) = \text{fmap}_L(f)(x) = x \triangleright \text{fmap}_L(f) = x \triangleright f^{\uparrow L} = f^{\uparrow L}(x) \quad .$$

In the lifting notation, the identity and composition laws for a functor L are especially easy to use:

$$\text{id}^{\uparrow L} = \text{id} \quad , \qquad (f \fatsemi g)^{\uparrow L} = f^{\uparrow L} \fatsemi g^{\uparrow L} \quad .$$

Applying a composition of lifted functions to a value looks like this:

$$x \triangleright (f \fatsemi g)^{\uparrow L} = x \triangleright f^{\uparrow L} \fatsemi g^{\uparrow L} = x \triangleright f^{\uparrow L} \triangleright g^{\uparrow L} \quad .$$

This equation directly represents the Scala code syntax:

```
x.map(f andThen g) == x.map(f).map(g)
```

if we make the pipe symbol ($\triangleright$) group weaker than the composition symbol ($\fatsemi$).

Written in the *backward* notation ($f \circ g$), the functor composition law is:

$$(g \circ f)^{\uparrow L} = g^{\uparrow L} \circ f^{\uparrow L} \quad .$$

The analogous notation for a contrafunctor $C^\bullet$ is:

$$f^{\downarrow C} \stackrel{\text{def}}{=} \text{cmap}_C(f) \quad .$$

The contrafunctor laws are then written as:

$$\text{id}^{\downarrow C} = \text{id} \quad , \qquad (f \fatsemi g)^{\downarrow C} = g^{\downarrow C} \fatsemi f^{\downarrow C} \quad , \qquad (g \circ f)^{\downarrow C} = f^{\downarrow C} \circ g^{\downarrow C} \quad .$$

We will mostly use the forward composition $f \fatsemi g$ in this book, keeping in mind that one can straightforwardly and mechanically translate between forward and backward notations:

$$f \fatsemi g = g \circ f \quad , \qquad x \triangleright f = f(x) \quad .$$

6.2.2 Bifunctors

A type constructor can be a functor with respect to several type parameters. A **bifunctor** is a type constructor with *two* type parameters that satisfies the functor laws with respect to both parameters.

As an example, consider the type constructor F defined by:

$$F^{A,B} \stackrel{\text{def}}{=} A \times B \times B \quad .$$

If we fix the type parameter B but let the parameter A vary, we get a type constructor that we denote by $F^{\bullet,B}$. We see that the type constructor $F^{\bullet,B}$ is a functor with the `fmap` function:

$$\text{fmap}_{F^{\bullet,B}}(f^{:A\to C}) \stackrel{\text{def}}{=} a^{:A} \times b_1^{:B} \times b_2^{:B} \to f(a) \times b_1 \times b_2 \quad .$$

Instead of saying that $F^{\bullet,B}$ is a functor, we can also say more verbosely that $F^{A,B}$ is a functor with respect to A when B is held fixed.

If we now fix the type parameter A, we find that the type constructor $F^{A,\bullet}$ is a functor with the following `fmap` method:

$$\text{fmap}_{F^{A,\bullet}}(g^{:B\to D}) \stackrel{\text{def}}{=} a^{:A} \times b_1^{:B} \times b_2^{:B} \to a \times g(b_1) \times g(b_2) \quad .$$

Since the bifunctor $F^{\bullet,\bullet}$ is a functor with respect to each type parameter separately, we can transform a value of type $F^{A,B}$ to a value of type $F^{C,D}$ by applying the two `fmap` functions one after another. It is convenient to denote this transformation by a single operation called `bimap` that uses two functions $f^{:A\to C}$ and $g^{:B\to D}$ as arguments:

$$\begin{aligned} &\text{bimap}_F(f^{:A\to C})(g^{:B\to D}) : F^{A,B} \to F^{C,D} \quad , \\ &\text{bimap}_F(f^{:A\to C})(g^{:B\to D}) \stackrel{\text{def}}{=} \text{fmap}_{F^{\bullet,B}}(f^{:A\to C}) \fatsemi \text{fmap}_{F^{C,\bullet}}(g^{:B\to D}) \quad . \end{aligned} \tag{6.8}$$

In the condensed notation, this is written as:

$$\text{bimap}_F(f^{:A\to C})(g^{:B\to D}) \stackrel{\text{def}}{=} f^{\uparrow F^{\bullet,B}} \fatsemi g^{\uparrow F^{C,\bullet}} \quad ,$$

although in this case the longer notation in Eq. (6.8) may be easier to reason about.

What if we apply the two `fmap` functions in the opposite order? Since these functions work with different type parameters, it is reasonable to expect that the transformation $F^{A,B} \to F^{C,D}$ should be independent of the order of application:

$$\text{fmap}_{F^{\bullet,B}}(f^{:A\to C}) \,\text{⨾}\, \text{fmap}_{F^{C,\bullet}}(g^{:B\to D}) = \text{fmap}_{F^{A,\bullet}}(g^{:B\to D}) \,\text{⨾}\, \text{fmap}_{F^{\bullet,D}}(f^{:A\to C}) \quad . \tag{6.9}$$

This equation is illustrated by the type diagram below.

$$\begin{tikzcd}
 & F^{C,B} \arrow[dr, "\text{fmap}_{F^{C,\bullet}}(g^{:B\to D})"] & \\
F^{A,B} \arrow[ur, "\text{fmap}_{F^{\bullet,B}}(f^{:A\to C})"] \arrow[rr, "\text{bimap}_F(f^{:A\to C})(g^{:B\to D})"] \arrow[dr, "\text{fmap}_{F^{A,\bullet}}(g^{:B\to D})"'] & & F^{C,D} \\
 & F^{A,D} \arrow[ur, "\text{fmap}_{F^{\bullet,D}}(f^{:A\to C})"'] &
\end{tikzcd}$$

Different paths in this diagram give the same results if they arrive at the same vertex (as mathematicians say, "the diagram is commutative"). In this way, the diagram illustrates at once the commutativity law (6.9) and the definition (6.8) of bimap_F.

Let us verify the commutativity law for the bifunctor $F^{A,B} \stackrel{\text{def}}{=} A \times B \times B$:

$$\begin{aligned}
&\text{left-hand side :} && \text{fmap}_{F^{\bullet,B}}(f^{:A\to C}) \,\text{⨾}\, \text{fmap}_{F^{C,\bullet}}(g^{:B\to D}) \\
&\text{definitions of } \text{fmap}_{F^{\bullet,\bullet}} : && = (a^{:A} \times b_1^{:B} \times b_2^{:B} \to f(a) \times b_1 \times b_2) \,\text{⨾}\, (c^{:C} \times b_1^{:B} \times b_2^{:B} \to c \times g(b_1) \times g(b_2)) \\
&\text{compute composition :} && = a^{:A} \times b_1^{:B} \times b_2^{:B} \to f(a) \times g(b_1) \times g(b_2) \quad , \\
&\text{right-hand side :} && \text{fmap}_{F^{A,\bullet}}(g^{:B\to D}) \,\text{⨾}\, \text{fmap}_{F^{\bullet,D}}(f^{:A\to C}) \\
&\text{definitions of } \text{fmap}_{F^{\bullet,\bullet}} : && = (a^{:A} \times b_1^{:B} \times b_2^{:B} \to a \times g(b_1) \times g(b_2)) \,\text{⨾}\, (a^{:A} \times d_1^{:D} \times d_2^{:D} \to f(a) \times d_1 \times d_2) \\
&\text{compute composition :} && = a^{:A} \times b_1^{:B} \times b_2^{:B} \to f(a) \times g(b_1) \times g(b_2) \quad .
\end{aligned}$$

Both sides of the law are equal.

The commutativity law (6.9) leads to the composition law of `bimap`:

$$\text{bimap}_F(f_1^{:A\to C})(g_1^{:B\to D}) \,\text{⨾}\, \text{bimap}_F(f_2^{:C\to E})(g_2^{:D\to G}) = \text{bimap}_F(f_1 \,\text{⨾}\, f_2)(g_1 \,\text{⨾}\, g_2) \quad . \tag{6.10}$$

The following type diagram shows the relationships between various `bimap` and `fmap` functions:

$$\begin{tikzcd}
F^{A,B} \arrow[r, "\text{fmap}_{F^{\bullet,B}}(f_1^{:A\to C})"] \arrow[d, "\text{fmap}_{F^{A,\bullet}}(g_1^{:B\to D})"] \arrow[dr, "\text{bimap}_F(f_1)(g_1)"] & F^{C,B} \arrow[r, "\text{fmap}_{F^{\bullet,B}}(f_2^{:C\to E})"] \arrow[d, "\text{fmap}_{F^{C,\bullet}}(g_1^{:B\to D})"] \arrow[dr, "\text{bimap}_F(f_2)(g_1)"] & F^{E,B} \arrow[d, "\text{fmap}_{F^{E,\bullet}}(g_1^{:B\to D})"] \\
F^{A,D} \arrow[r, "\text{fmap}_{F^{\bullet,D}}(f_1^{:A\to C})"] \arrow[d, "\text{fmap}_{F^{A,\bullet}}(g_2^{:D\to G})"] \arrow[dr, "\text{bimap}_F(f_1)(g_2)"] & F^{C,D} \arrow[r, "\text{fmap}_{F^{\bullet,D}}(f_2^{:C\to E})"] \arrow[d, "\text{fmap}_{F^{C,\bullet}}(g_2^{:D\to G})"] \arrow[dr, "\text{bimap}_F(f_2)(g_2)"] & F^{E,D} \arrow[d, "\text{fmap}_{F^{E,\bullet}}(g_2^{:D\to G})"] \\
F^{A,G} \arrow[r, "\text{fmap}_{F^{\bullet,G}}(f_1^{:A\to C})"] & F^{C,G} \arrow[r, "\text{fmap}_{F^{\bullet,G}}(f_2^{:C\to E})"] & F^{E,G}
\end{tikzcd}$$

To derive the composition law from Eq. (6.9), write:

$$\begin{aligned}
& && \text{bimap}_F(f_1)(g_1) \,\text{⨾}\, \text{bimap}_F(f_2)(g_2) \\
&\text{use Eq. (6.8) :} && = \text{fmap}_{F^{\bullet,B}}(f_1) \,\text{⨾}\, \underline{\text{fmap}_{F^{C,\bullet}}(g_1) \,\text{⨾}\, \text{fmap}_{F^{\bullet,D}}(f_2)} \,\text{⨾}\, \text{fmap}_{F^{E,\bullet}}(g_2) \\
&\text{commutativity law (6.9) :} && = \text{fmap}_{F^{\bullet,B}}(f_1) \,\underline{\text{⨾}}\, \text{fmap}_{F^{\bullet,B}}(f_2) \,\text{⨾}\, \text{fmap}_{F^{E,\bullet}}(g_1) \,\underline{\text{⨾}}\, \text{fmap}_{F^{E,\bullet}}(g_2) \\
&\text{composition laws :} && = \text{fmap}_{F^{\bullet,B}}(f_1 \,\text{⨾}\, f_2) \,\text{⨾}\, \text{fmap}_{F^{E,\bullet}}(g_1 \,\text{⨾}\, g_2) \\
&\text{use Eq. (6.8) :} && = \text{bimap}_F(f_1 \,\text{⨾}\, f_2)(g_1 \,\text{⨾}\, g_2) \quad .
\end{aligned}$$

Conversely, we can derive Eq. (6.9) from the composition law (6.10). We write the composition law with specially chosen functions:

$$\text{bimap}_F(f^{:A\to C})(g^{:B\to D}) = \text{bimap}_F(\text{id}^{:A\to A})(g^{:B\to D}) \,⨾\, \text{bimap}_F(f^{:A\to C})(\text{id}^{:D\to D}) \quad . \tag{6.11}$$

Using Eq. (6.8), we find:

$$\begin{aligned}
\text{expect } \text{fmap}_{F^{A,\bullet}}(g) \,⨾\, \text{fmap}_{F^{\bullet,D}}(f) : \quad & \text{fmap}_{F^{\bullet,B}}(f^{:A\to C}) \,⨾\, \text{fmap}_{F^{C,\bullet}}(g^{:B\to D}) \\
\text{use Eq. (6.8)} : \quad & = \text{bimap}_F(f^{:A\to C})(g^{:B\to D}) \\
\text{use Eq. (6.11)} : \quad & = \text{bimap}_F(\text{id}^{:A\to A})(g^{:B\to D}) \,⨾\, \text{bimap}_F(f^{:A\to C})(\text{id}^{:D\to D}) \\
\text{use Eq. (6.8)} : \quad & = \underline{\text{fmap}_{F^{\bullet,B}}(\text{id})} \,⨾\, \text{fmap}_{F^{A,\bullet}}(g) \,⨾\, \text{fmap}_{F^{\bullet,D}}(f) \,⨾\, \underline{\text{fmap}_{F^{C,\bullet}}(\text{id})} \\
\text{identity laws for } F : \quad & = \text{fmap}_{F^{A,\bullet}}(g) \,⨾\, \text{fmap}_{F^{\bullet,D}}(f) \quad .
\end{aligned}$$

The identity law of `bimap` holds as well:

$$\begin{aligned}
\text{expect to equal id} : \quad & \text{bimap}_F(\text{id}^{:A\to A})(\text{id}^{:B\to B}) \\
\text{use Eq. (6.8)} : \quad & = \underline{\text{fmap}_{F^{\bullet,B}}(\text{id})} \,⨾\, \underline{\text{fmap}_{F^{C,\bullet}}(\text{id})} \\
\text{identity laws for } F : \quad & = \text{id} \,⨾\, \text{id} = \text{id} \quad .
\end{aligned}$$

If $F^{A,B}$ is known to be a functor separately with respect to A and B, will the commutativity law (6.9) always hold? The calculation for the example $F^{A,B} \stackrel{\text{def}}{=} A \times B \times B$ shows that the two `fmap` functions commute because they work on different parts of the data structure $F^{A,B}$. This turns out[3] to be true in general: the commutativity law follows from the parametricity of the `fmap` functions. Because of that, we do not need to verify the `bimap` laws as long as $F^{\bullet,B}$ and $F^{A,\bullet}$ are lawful functors.

Type constructors with more than two type parameters have similar properties. It is sufficient to check the functor laws with respect to each type parameter separately.

In general, a type constructor may be a functor with respect to some type parameters and a contrafunctor with respect to others. We will see examples of such type constructors later.

6.2.3 Constructions of functors

What type expressions will produce a functor? Functional programming languages support the six standard type constructions (see Section 5.1.2). This section will check whether each construction produces a new type that obeys the functor laws. The results are summarized in Table 6.2.

In each of these constructions, the `fmap` function for a new functor is defined either from scratch or by using the known `fmap` functions for previously defined type constructors. The rest of this section will derive the code for these constructions and prove their laws. We will use the code notation for brevity, occasionally showing the translation into the Scala syntax.

Statement 6.2.3.1 The type constructor $\text{Id}^A \stackrel{\text{def}}{=} A$ is a lawful functor (the **identity functor**).

Proof The `fmap` function is defined by:

$$\begin{aligned}
& \text{fmap}_{\text{Id}} : (A \to B) \to \text{Id}^A \to \text{Id}^B \cong (A \to B) \to A \to B \quad , \\
& \text{fmap}_{\text{Id}} \stackrel{\text{def}}{=} (f^{:A\to B} \to f) = \text{id}^{:(A\to B)\to A\to B} \quad .
\end{aligned}$$

The identity function is the only fully parametric implementation of the type signature $(A \to B) \to A \to B$. Since the code of `fmap` is the identity function, the laws are satisfied automatically:

$$\begin{aligned}
\text{identity law} : \quad & \text{fmap}_{\text{Id}}(\text{id}) = \text{id}(\text{id}) = \text{id} \quad , \\
\text{composition law} : \quad & \text{fmap}_{\text{Id}}(f \,⨾\, g) = f \,⨾\, g = \text{fmap}_{\text{Id}}(f) \,⨾\, \text{fmap}_{\text{Id}}(g) \quad .
\end{aligned}$$

[3] A proof is given in Section D.2 of Appendix D. It assumes that the bifunctor F is fully parametric.

Construction	Type notation	Comment
type parameter	$L^A \stackrel{\text{def}}{=} A$	the identity functor
product type	$L^A \stackrel{\text{def}}{=} P^A \times Q^A$	the functor product; P and Q must be functors
disjunctive type	$L^A \stackrel{\text{def}}{=} P^A + Q^A$	the functor co-product; P and Q must be functors
function type	$L^A \stackrel{\text{def}}{=} C^A \to P^A$	the functor exponential; P is a functor and C a contrafunctor
type parameter	$L^A \stackrel{\text{def}}{=} Z$	the constant functor; Z is a fixed type
type constructor	$L^A \stackrel{\text{def}}{=} P^{Q^A}$	functor composition; P and Q are both functors or both contrafunctors
recursive type	$L^A \stackrel{\text{def}}{=} S^{A,L^A}$	recursive functor; $S^{A,B}$ must be a functor w.r.t. both A and B

Table 6.2: Type constructions defining a functor L^A.

Statement 6.2.3.2 The type constructor $\text{Const}^{Z,A} \stackrel{\text{def}}{=} Z$ is a lawful functor (a **constant functor**) with respect to the type parameter A.

Proof The `fmap` function is defined by:

$$\text{fmap}_{\text{Const}} : (A \to B) \to \text{Const}^{Z,A} \to \text{Const}^{Z,B} \cong (A \to B) \to Z \to Z \quad ,$$
$$\text{fmap}_{\text{Const}}(f^{:A\to B}) \stackrel{\text{def}}{=} (z^{:Z} \to z) = \text{id}^{:Z\to Z} \quad .$$

It is a constant function that ignores f and returns the identity $\text{id}^{:Z\to Z}$. The laws are satisfied:

$$\text{identity law :} \quad \text{fmap}_{\text{Const}}(\text{id}) = \text{id} \quad ,$$
$$\text{composition law :} \quad \text{fmap}_{\text{Const}}(f \,⨾\, g) = \text{id} = \text{fmap}_{\text{Const}}(f) \,⨾\, \text{fmap}_{\text{Const}}(g) = \text{id} \,⨾\, \text{id} \quad .$$

The corresponding Scala code is:

```
type Const[Z, A] = Z
def fmap[A, B](f: A => B): Const[Z, A] => Const[Z, B] = identity[Z]
```

The identity functor $\text{Id}^\bullet$ and the constant functor $\text{Const}^{Z,\bullet}$ are not often used: their `fmap` implementations are identity functions, and so they rarely provide useful functionality.

We have seen that type constructors with product types, such as $L^A \stackrel{\text{def}}{=} A \times A \times A$, are functors. The next construction (the **functor product**) explains why.

Statement 6.2.3.3 If $L^\bullet$ and $M^\bullet$ are two functors then the product $P^A \stackrel{\text{def}}{=} L^A \times M^A$ is also a functor.

Proof The `fmap` function for P is defined by:

```
def fmap[A, B](f: A => B): (L[A], M[A]) => (L[B], M[B]) = {
  case (la, ma) => (la.map(f), ma.map(f))
}
```

The corresponding code notation is:

$$f^{\uparrow P} \stackrel{\text{def}}{=} l^{:L^A} \times m^{:M^A} \to f^{\uparrow L}(l) \times f^{\uparrow M}(m) \quad .$$

Writing this code using the pipe (▷) operation makes it somewhat closer to the Scala syntax:

$$(l^{:L^A} \times m^{:M^A}) \triangleright f^{\uparrow P} \stackrel{\text{def}}{=} (l \triangleright f^{\uparrow L}) \times (m \triangleright f^{\uparrow M}) \quad . \tag{6.12}$$

An alternative notation uses the **pair product** symbol ⊠ defined by:

$$p^{:A\to B} \boxtimes q^{:C\to D} : A \times C \to B \times D \quad ,$$
$$p \boxtimes q \stackrel{\text{def}}{=} a \times c \to p(a) \times q(c) \quad ,$$
$$\text{or equivalently :} \quad (a \times c) \triangleright (p \boxtimes q) = (a \triangleright p) \times (b \triangleright q) \quad .$$

In this notation, the lifting for P is defined more concisely:

$$f^{\uparrow P} = f^{\uparrow L\times M} \stackrel{\text{def}}{=} f^{\uparrow L} \boxtimes f^{\uparrow M} \quad . \tag{6.13}$$

We need to verify the identity law and the composition law.

To verify the identity law of P, pipe an arbitrary value of type $L^A \times M^A$ into both sides of the law:

$$\begin{aligned}
\text{expect to equal } l \times m: \quad & (l^{:L^A} \times m^{:M^A}) \triangleright \text{id}^{\uparrow P} \\
\text{definition of } f^{\uparrow P}: \quad & = (l \triangleright \underline{\text{id}^{\uparrow L}}) \times (m \triangleright \underline{\text{id}^{\uparrow M}}) \\
\text{identity laws of } L \text{ and } M: \quad & = (\underline{l \triangleright \text{id}}) \times (\underline{m \triangleright \text{id}}) \\
\text{definition of id}: \quad & = l \times m \quad .
\end{aligned}$$

To verify the composition law of P, we need to show that:

$$f^{\uparrow P} \fatsemi g^{\uparrow P} = (f \fatsemi g)^{\uparrow P} \quad .$$

Apply both sides of this equation to an arbitrary value of type $L^A \times M^A$:

$$\begin{aligned}
\text{expect to equal } (l \times m) \triangleright (f \fatsemi g)^{\uparrow P}: \quad & (l^{:L^A} \times m^{:M^A}) \triangleright f^{\uparrow P} \underline{\fatsemi} g^{\uparrow P} \\
\triangleright \text{ notation}: \quad & = \underline{(}l^{:L^A} \times m^{:M^A}\underline{)} \triangleright \underline{f^{\uparrow P}} \triangleright g^{\uparrow P} \\
\text{use Eq. (6.12)}: \quad & = \underline{\big(}(l \triangleright f^{\uparrow L}) \times (m \triangleright f^{\uparrow M})\underline{\big)} \triangleright \underline{g^{\uparrow P}} \\
\text{use Eq. (6.12)}: \quad & = (l \triangleright f^{\uparrow L} \underline{\triangleright} g^{\uparrow L}) \times (m \triangleright f^{\uparrow M} \underline{\triangleright} g^{\uparrow M}) \\
\triangleright \text{ notation}: \quad & = (l \triangleright \underline{f^{\uparrow L} \fatsemi g^{\uparrow L}}) \times (m \triangleright \underline{f^{\uparrow M} \fatsemi g^{\uparrow M}}) \\
\text{composition laws of } L \text{ and } M: \quad & = (l \triangleright (f \fatsemi g)^{\uparrow L}) \times (m \triangleright (f \fatsemi g)^{\uparrow M}) \\
\text{use Eq. (6.12)}: \quad & = (l \times m) \triangleright (f \fatsemi g)^{\uparrow P} \quad .
\end{aligned}$$

The calculations are shorter if we use the pair product operation:

$$\begin{aligned}
\text{expect to equal } (f \fatsemi g)^{\uparrow P}: \quad & f^{\uparrow P} \fatsemi g^{\uparrow P} = (f^{\uparrow L} \boxtimes f^{\uparrow M}) \fatsemi (g^{\uparrow L} \boxtimes g^{\uparrow M}) \\
\text{composition of functions under } \boxtimes: \quad & = (\underline{f^{\uparrow L} \fatsemi g^{\uparrow L}}) \boxtimes (\underline{f^{\uparrow M} \fatsemi g^{\uparrow M}}) \\
\text{composition laws of } L \text{ and } M: \quad & = (f \fatsemi g)^{\uparrow L} \boxtimes (f \fatsemi g)^{\uparrow M} = (f \fatsemi g)^{\uparrow P} \quad .
\end{aligned}$$

For comparison, the same derivation using the Scala code syntax looks like this:

```
(( l, m )).map(f).map(g) == (( l.map(f), m.map(f) )).map(g)
  == (( l.map(f).map(g), m.map(f).map(g) ))
  == (( l.map(f andThen g), m.map(f andThen g) ))
```

assuming that the `map` method is defined on pairs by Eq. (6.12):

```
(( l, m )).map(f) == (( l.map(f), m.map(f) ))
```

The proof written in the Scala syntax does not show the type constructors whose `map` methods are used in each expression. For instance, it is not indicated that the two `map` methods used in the expression `m.map(f).map(g)` belong to the *same* type constructor M and thus obey M's composition law. The code notation shows this more concisely and more clearly, helping us in reasoning:

$$m \triangleright f^{\uparrow M} \triangleright g^{\uparrow M} = m \triangleright f^{\uparrow M} \fatsemi g^{\uparrow M} = m \triangleright (f \fatsemi g)^{\uparrow M} \quad .$$

By the convention of the pipe notation, it groups to the left, so we have:

$$(x \triangleright f) \triangleright g = x \triangleright f \triangleright g = x \triangleright f \fatsemi g = x \triangleright (f \fatsemi g) = (f \fatsemi g)(x) = g(f(x)) \quad .$$

We will often use this notation in derivations. (Chapter 7 gives an overview of the derivation techniques, including some more details about the pipe notation.)

Statement 6.2.3.4 If P^A and Q^A are functors then $L^A \stackrel{\text{def}}{=} P^A + Q^A$ is a functor, with `fmap` defined by:

```
def fmap[A, B](f: A => B): Either[P[A], Q[A]] => Either[P[B], Q[B]] = {
  case Left(pa)    => Left(fmap_P(f)(pa))   // Use fmap for P.
  case Right(qa)   => Right(fmap_Q(f)(qa))  // Use fmap for Q.
}
```

The functor $L^\bullet$ is the **functor co-product** of $P^\bullet$ and $Q^\bullet$. The code notation for the `fmap` function is:

$$\text{fmap}_L(f^{:A\to B}) = f^{\uparrow L} \stackrel{\text{def}}{=} \left\| \begin{array}{c|cc} & P^B & Q^B \\ \hline P^A & f^{\uparrow P} & \mathbb{0} \\ Q^A & \mathbb{0} & f^{\uparrow Q} \end{array} \right| \quad .$$

Here we assume that lawful `fmap` functions are given for the functors P and Q.

Proof Omitting the type annotations, we write the code of $\text{fmap}_L(f)$ as:

$$\text{fmap}_L(f) = f^{\uparrow L} = \left\| \begin{array}{cc} f^{\uparrow P} & \mathbb{0} \\ \mathbb{0} & f^{\uparrow Q} \end{array} \right| \quad . \tag{6.14}$$

To verify the identity law, use Eq. (6.14) and the identity laws for P and Q:

$$\text{expect to equal id}: \quad \text{id}^{\uparrow L} = \left\| \begin{array}{cc} \text{id}^{\uparrow P} & \mathbb{0} \\ \mathbb{0} & \text{id}^{\uparrow Q} \end{array} \right| = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \end{array} \right|$$

$$\text{identity function in matrix notation}: \quad = \text{id} \quad .$$

To verify the composition law:

$$\text{expect to equal } (f \fatsemi g)^{\uparrow L}: \quad f^{\uparrow L} \fatsemi g^{\uparrow L} = \left\| \begin{array}{cc} f^{\uparrow P} & \mathbb{0} \\ \mathbb{0} & f^{\uparrow Q} \end{array} \right| \fatsemi \left\| \begin{array}{cc} g^{\uparrow P} & \mathbb{0} \\ \mathbb{0} & g^{\uparrow Q} \end{array} \right|$$

$$\text{matrix composition}: \quad = \left\| \begin{array}{cc} \underline{f^{\uparrow P} \fatsemi g^{\uparrow P}} & \mathbb{0} \\ \mathbb{0} & \underline{f^{\uparrow Q} \fatsemi g^{\uparrow Q}} \end{array} \right|$$

$$\text{composition laws of } P \text{ and } Q: \quad = \left\| \begin{array}{cc} (f \fatsemi g)^{\uparrow P} & \mathbb{0} \\ \mathbb{0} & (f \fatsemi g)^{\uparrow Q} \end{array} \right| = (f \fatsemi g)^{\uparrow L} \quad .$$

The last two statements show that any type constructor built up via primitive types, type parameters, products and co-products, such as $L^A \stackrel{\text{def}}{=} \mathbb{1} + (\text{String} + A) \times A \times \text{Int} + A$, is a functor. Functors of this kind are called **polynomial functors** because of the analogy to ordinary arithmetic polynomial functions of a variable A. The type notation with its symbols $(+, \times)$ makes this analogy visually clear.

Implementing `fmap` for a polynomial functor is straightforward: `fmap` replaces each occurrence of the a value of type A by the corresponding value of type B, leaving constant types unchanged and keeping the order of parts in all products and disjunctive types. Previously, our implementations of `fmap` for various type constructors (such as shown in Example 6.1.9.2) were guided by the idea of preserving information. Statements 6.2.3.3–6.2.3.4 explain why those implementations of the `fmap` are correct (i.e., obey the functor laws).

The next construction shows when a function type is a functor: the argument of the function must be a contrafunctor.

Statement 6.2.3.5 If C is a contrafunctor and P is a functor then $L^A \stackrel{\text{def}}{=} C^A \to P^A$ is a functor, called a **functor exponential**, with `fmap` defined by:

$$\text{fmap}_L^{A,B}(f^{:A\to B}) : (C^A \to P^A) \to C^B \to P^B \quad ,$$

$$\text{fmap}_L(f^{:A\to B}) = f^{\uparrow L} \stackrel{\text{def}}{=} h^{:C^A \to P^A} \to f^{\downarrow C} \fatsemi h \fatsemi f^{\uparrow P} \quad . \tag{6.15}$$

The corresponding Scala code is:

```
def fmap_L[A, B](f: A => B)(h: C[A] => P[A]): C[B] => P[B] = {
  cmap_C(f) andThen h andThen fmap_P(f)
}
```

A type diagram for fmap_L can be drawn as:

$$\begin{tikzcd}
 & C^A \arrow[r, "h"] & P^A \arrow[dr, "\text{fmap}_P(f^{:A\to B})"] & \\
C^B \arrow[ur, "\text{cmap}_C(f^{:A\to B})"] \arrow[rrr, "\text{fmap}_L(f^{:A\to B})(h^{:C^A\to P^A})"'] & & & P^B
\end{tikzcd}$$

Proof Since the types are already checked, we can use Eq. (6.15) without type annotations:

$$h \triangleright f^{\uparrow L} = f^{\downarrow C} ⨾ h ⨾ f^{\uparrow P} \quad . \tag{6.16}$$

To verify the identity law of L, show that $\text{id}^{\uparrow L}(h) = h$:

$$\begin{aligned}
\text{expect to equal } h: \quad & h \triangleright \text{id}^{\uparrow L} \\
\text{definition (6.16) of } {}^{\uparrow L}: \quad & = \underline{\text{id}^{\downarrow C}} ⨾ h ⨾ \underline{\text{id}^{\uparrow P}} \\
\text{identity laws of } C \text{ and } P: \quad & = \underline{\text{id}} ⨾ h ⨾ \underline{\text{id}} \\
\text{definition of id}: \quad & = h \quad .
\end{aligned}$$

To verify the composition law of L, it helps to apply both sides of the law to an arbitrary $h^{:L^A}$:

$$\begin{aligned}
\text{expect to equal } h \triangleright f^{\uparrow L} ⨾ g^{\uparrow L}: \quad & h \triangleright (f ⨾ g)^{\uparrow L} \\
\text{definition (6.16) of } {}^{\uparrow L}: \quad & = (\underline{f ⨾ g})^{\downarrow C} ⨾ h ⨾ (\underline{f ⨾ g})^{\uparrow P} \\
\text{composition laws of } C \text{ and } P: \quad & = g^{\downarrow C} ⨾ \underline{f^{\downarrow C}} ⨾ h ⨾ \underline{f^{\uparrow P}} ⨾ g^{\uparrow P} \\
\text{definition (6.16) of } {}^{\uparrow L}: \quad & = \underline{g^{\downarrow C}} ⨾ (h \triangleright f^{\uparrow L}) ⨾ \underline{g^{\uparrow P}} \\
\text{definition (6.16) of } {}^{\uparrow L}: \quad & = (h \triangleright f^{\uparrow L}) \triangleright g^{\uparrow L} = h \triangleright f^{\uparrow L} ⨾ g^{\uparrow L} \quad .
\end{aligned}$$

It is important for this proof that the order of function compositions is reversed when lifting to a contrafunctor C: $(f ⨾ g)^{\downarrow C} = g^{\downarrow C} ⨾ f^{\downarrow C}$. If C were a functor, the proof would not work because we would have obtained $f^{\uparrow C} ⨾ g^{\uparrow C}$ instead of $g^{\downarrow C} ⨾ f^{\downarrow C}$. The order of composition cannot be permuted for arbitrary functions f, g. So, we would not be able to group $f^{\downarrow C} ⨾ h ⨾ f^{\uparrow P}$ together. □

Examples of functors obtained via the exponential construction are $L^A \stackrel{\text{def}}{=} Z \to A$ (with the contrafunctor C^A chosen as the constant contrafunctor Z, where Z is a fixed type) and $L^A \stackrel{\text{def}}{=} (A \to Z) \to A$ (with the contrafunctor $C^A \stackrel{\text{def}}{=} A \to Z$). Statement 6.2.3.5 generalizes those examples to arbitrary contrafunctors C^A used as arguments of function types.

Similarly, one can prove that $P^A \to C^A$ is a contrafunctor (Exercise 6.3.1.2). Together with Statements 6.2.3.3–6.2.3.5, this gives us the rules of reasoning about covariance and contravariance of type parameters in arbitrary type expressions. Every function arrow ($\to$) flips the variance from covariant to contravariant and back. For instance, the identity functor $L^A \stackrel{\text{def}}{=} A$ is covariant in A, while $A \to Z$ is contravariant in A, and $(A \to Z) \to Z$ is again covariant in A. As we have seen, $A \to A \to Z$ is contravariant in A, so any number of curried arrows count as one in this reasoning (and, in any case, $A \to A \to Z \cong A \times A \to Z$). Products and disjunctions do not change variance, so $(A \to Z_1) \times (A \to Z_2) + (A \to Z_3)$ is contravariant in A. This is shown in more detail in Section 6.2.5.

The remaining constructions set a type parameter to another type. The **functor composition** P^{Q^A}, written in Scala as `P[Q[A]]`, is analogous to a function composition such as $f(g(x))$ except for using type constructors. Viewed in this way, type constructors are **type-level functions** (i.e., mappings of types). So, functor composition may be denoted by $P \circ Q$, like the function composition $f \circ g$.

An example of functor composition in Scala is `List[Option[A]]`. Since both `List` and `Option` have a `map` method, we may write code such as:

```
val p: List[Option[Int]] = List(Some(1), None, Some(2), None, Some(3))

scala> p.map(_.map(x => x + 10))
res0: List[Option[Int]] = List(Some(11), None, Some(12), None, Some(13))
```

The code `p.map(_.map(f))` lifts an $f^{:A\to B}$ into a function of type `List[Option[A]] => List[Option[B]]`. In this way, we may perform the `map` operation on the nested data type `List[Option[_]]`.

The next statement shows that this code always produces a lawful `map` function. In other words, the composition of functors is always a functor.

Statement 6.2.3.6 If P^A and Q^A are functors then $L^A \stackrel{\text{def}}{=} P^{Q^A}$ is also a functor, with `fmap` defined by:

```
def fmap_L[A, B](f: A => B): P[Q[A]] => P[Q[B]] = fmap_P(fmap_Q(f))
```

Here we assumed that the functions fmap_P and fmap_Q are known and satisfy the functor laws.

In the code notation, fmap_L is written equivalently as:

$$\begin{aligned}
\text{type signature}: &\quad \text{fmap}_L : f^{:A\to B} \to P^{Q^A} \to P^{Q^B} \quad , \\
\text{implementation}: &\quad \text{fmap}_L(f) \stackrel{\text{def}}{=} \text{fmap}_P(\text{fmap}_Q(f)) \quad , \\
\text{equivalent code}: &\quad \text{fmap}_L \stackrel{\text{def}}{=} \text{fmap}_Q \,{}^{\circ}\!{}_{\circ}\, \text{fmap}_P \quad , \qquad (6.17) \\
\text{in a shorter notation}: &\quad f^{\uparrow L} \stackrel{\text{def}}{=} (f^{\uparrow Q})^{\uparrow P} \stackrel{\text{def}}{=} f^{\uparrow Q \uparrow P} \quad . \qquad (6.18)
\end{aligned}$$

Proof To verify the identity law of L, use the identity laws for P and Q:

$$\text{id}^{\uparrow L} = (\underline{\text{id}^{\uparrow Q}})^{\uparrow P} = \underline{\text{id}^{\uparrow P}} = \text{id} \quad .$$

To verify the composition law of L, use the composition laws for P and Q:

$$(f \,{}^{\circ}\!{}_{\circ}\, g)^{\uparrow L} = \left(\underline{(f \,{}^{\circ}\!{}_{\circ}\, g)}^{\uparrow Q}\right)^{\uparrow P} = \left(\underline{f^{\uparrow Q} \,{}^{\circ}\!{}_{\circ}\, g^{\uparrow Q}}\right)^{\uparrow P} = f^{\uparrow Q \uparrow P} \,{}^{\circ}\!{}_{\circ}\, g^{\uparrow Q \uparrow P} \quad .$$

Finally, we consider recursive data types such as lists and trees (Section 3.3). It is helpful to use the type notation for reasoning about those types. The list type:

```
sealed trait List[A]
final case class Empty()                          extends List[A]
final case class Head[A](head: A, tail: List[A]) extends List[A]
```

is written in type notation as:

$$\text{List}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times \text{List}^A \quad .$$

The binary tree type:

```
sealed trait Tree2[A]
final case class Leaf[A](a: A)                        extends Tree2[A]
final case class Branch[A](x: Tree2[A], y: Tree2[A]) extends Tree2[A]
```

is defined by $\text{Tree}_2^A \stackrel{\text{def}}{=} A + \text{Tree}_2^A \times \text{Tree}_2^A$. Such definitions of recursive types look like "type equations". We can generalize these examples to a recursive definition:

$$L^A \stackrel{\text{def}}{=} S^{A, L^A} \quad , \qquad (6.19)$$

where $S^{A,R}$ is a suitably chosen type constructor (called a **recursion scheme**) with two type parameters A, R. If a recursion scheme $S^{\bullet,\bullet}$ is given, the Scala code defining $L^\bullet$ can be written as:

```
type S[A, R] = ...            // Must be defined previously as type alias, class, or trait.
final case class L[A](x: S[A, L[A]])    // Define recursive type L via recursion scheme S.
```

Description	Type definition	Bifunctor $S^{A,R}$
list	$L^A \stackrel{\text{def}}{=} \mathbb{1} + A \times L^A$	$S^{A,R} \stackrel{\text{def}}{=} \mathbb{1} + A \times R$
non-empty list	$\text{NEL}^A \stackrel{\text{def}}{=} A + A \times \text{NEL}^A$	$S^{A,R} \stackrel{\text{def}}{=} A + A \times R$
list of odd length	$L^A \stackrel{\text{def}}{=} A + A \times A \times L^A$	$S^{A,R} \stackrel{\text{def}}{=} A + A \times A \times R$
binary tree	$L^A \stackrel{\text{def}}{=} A + L^A \times L^A$	$S^{A,R} \stackrel{\text{def}}{=} A + R \times R$
rose tree	$L^A \stackrel{\text{def}}{=} A + \text{NEL}^{L^A}$	$S^{A,R} \stackrel{\text{def}}{=} A + \text{NEL}^R$
perfect-shaped binary tree	$L^A \stackrel{\text{def}}{=} A + L^{A \times A}$	not possible
abstract syntax tree	$L^A \stackrel{\text{def}}{=} P^A + Q^{L^A}$	$S^{A,R} \stackrel{\text{def}}{=} P^A + Q^R$

Table 6.3: Recursive disjunctive types defined using type equations.

We must use a *class* to define `L` because Scala does not support recursive type aliases:

```
scala> type L[A] = Either[A, L[A]]
<console>:14: error: illegal cyclic reference involving type L

scala> final case class L[A](x: Either[A, L[A]])
defined class L
```

Table 6.3 summarizes our previous examples of recursive disjunctive types and shows the relevant choices of $S^{A,R}$, which turns out to be always a bifunctor. For abstract syntax trees, the functors $P^\bullet$ and $Q^\bullet$ must be given; they specify the available shapes of leaves and branches respectively.

We will now prove that Eq. (6.19) always defines a functor when $S^{\bullet,\bullet}$ is a bifunctor. The proof is by induction with an inductive assumption that the laws already hold for all recursive calls of `fmap`. The technique is similar to what we used in the proof for the `List` functor (Example 6.1.4.4).

Statement 6.2.3.7 If $S^{A,B}$ is a bifunctor (a functor with respect to both type parameters A and B) then the recursively defined type constructor L^A is a lawful functor:

$$L^A \stackrel{\text{def}}{=} S^{A,L^A} \quad .$$

The `fmap` method for L is a recursive function implemented as:

$$\text{fmap}_L(f^{:A\to B}) \stackrel{\text{def}}{=} \text{bimap}_S(f)(\text{fmap}_L(f)) \quad . \tag{6.20}$$

The corresponding Scala code is

```
final case class L[A](x: S[A, L[A]]) // The type constructor S[_, _] must be defined previously.

def bimap_S[A, B, C, D](f: A => C)(g: B => D): S[A, B] => S[C, D] = ???      // Must be defined.

def fmap_L[A, B](f: A => B): L[A] => L[B] = { case L(x) =>
    val newX: S[B, L[B]] = bimap_S(f)(fmap_L(f))(x)                 // Recursive call to fmap_L.
    L(newX)            // Need to wrap the value of type S[B, L[B]] into the type constructor L.
}
```

Proof Usually, laws for a recursive function (such as fmap_L) must be proved by induction. In the recursive implementation of fmap_L, its code calls itself in some cases but returns without recursive calls in other cases. So, the base case of induction corresponds to the non-recursive evaluations in the code of fmap_L, and we need to prove that the law is then satisfied. The inductive step must prove that the code of fmap_L obeys the law under the inductive assumption that all recursive calls to fmap_L already obey that law. In the proof, we do not need to separate the base case from the inductive step; we just derive the law using the inductive assumption whenever needed.

For clarity, we add an overline to recursive calls in the code formula:

$$\text{fmap}_L(f) \stackrel{\text{def}}{=} \text{bimap}_S(f)(\overline{\text{fmap}_L}(f)) \quad .$$

To prove the identity law:

$$\begin{aligned}
\text{expect to equal id}: &\quad \text{fmap}_L(\text{id}) \\
\text{definition of fmap}_L: &\quad = \text{bimap}_S(\text{id})(\underline{\overline{\text{fmap}_L}(\text{id})}) \\
\text{inductive assumption — the law holds for } \overline{\text{fmap}_L}: &\quad = \text{bimap}_S(\text{id})(\text{id}) \\
\text{identity law of } S: &\quad = \text{id} \quad .
\end{aligned}$$

To prove the composition law:

$$\begin{aligned}
\text{expect to equal fmap}_L(f \fatsemi g): &\quad \text{fmap}_L(f) \fatsemi \text{fmap}_L(g) \\
\text{definition of fmap}_L: &\quad = \text{bimap}_S(f)(\overline{\text{fmap}_L}(f)) \fatsemi \text{bimap}_S(g)(\overline{\text{fmap}_L}(g)) \\
\text{composition law of } S: &\quad = \text{bimap}_S(f \fatsemi g)(\underline{\overline{\text{fmap}_L}(f) \fatsemi \overline{\text{fmap}_L}(g)}) \\
\text{inductive assumption}: &\quad = \text{bimap}_S(f \fatsemi g)(\overline{\text{fmap}_L}(f \fatsemi g)) \\
\text{definition of fmap}_L: &\quad = \text{fmap}_L(f \fatsemi g) \quad .
\end{aligned}$$

For the perfect-shaped binary tree, the construction (6.19) is insufficient: no bifunctor S^{A,L^A} can replace the type argument A in L^A to obtain $L^{A\times A}$. To see that, consider that S^{A,L^A} is an application of a type-level function $S^{\bullet,\bullet}$ to its two type parameters, which are set to A and L^A. In Scala syntax, S^{A,L^A} is written as `S[A,L[A]]`. No matter how we define the type constructor S, the resulting type expression `S[A,L[A]]` will always use the type constructor `L` as `L[A]` and not as `L[(A,A)]`.

To describe perfect-shaped trees, we need to modify the construction by adding another arbitrary functor, $P^\bullet$, in the type argument of $L^\bullet$:

$$L^A \stackrel{\text{def}}{=} S^{A,L^{P^A}} \quad . \tag{6.21}$$

Perfect-shaped binary trees are defined by Eq. (6.21) with $S^{A,R} \stackrel{\text{def}}{=} A + R$ and $P^A \stackrel{\text{def}}{=} A \times A$. The Scala code for these definitions is:

```
type S[A, R] = Either[A, R]
type P[A] = (A, A)
final case class L[A](s: S[A, L[P[A]]])          // Or ‘case class L[A](s: Either[A, L[(A, A)]])‘.
```

Different choices of $P^\bullet$ will define perfect-shaped trees with different kinds of branching.

Exercise 6.3.1.8 shows that type constructors defined byEq. (6.21) are always functors.

6.2.4 Constructions of contrafunctors

The previous section performed **structural analysis** for functors: a systematic search for type constructions (product, co-product, etc.) that create new functors. *Mutatis mutandis*, similar constructions work for contrafunctors, as shown in Table 6.4. One difference with respect to Table 6.2 is the absence of the identity type constructor, $L^A \stackrel{\text{def}}{=} A$ (it is a functor, not a contrafunctor). However, the constant type constructor, $L^A \stackrel{\text{def}}{=} Z$, is a functor and a contrafunctor at the same time.

Let us now prove the validity of some of these constructions.

Statement 6.2.4.1 If Z is any fixed type, the constant type constructor $C^A \stackrel{\text{def}}{=} Z$ is a contrafunctor (the **constant contrafunctor**) whose `cmap` returns an identity function of type $Z \to Z$:

```
type Const[Z, A] = Z
def cmap[Z, A, B](f: B => A): Const[Z, A] => Const[Z, B] = identity[Z]
```

Proof All laws hold because `cmap` returns an identity function:

$$\begin{aligned}
\text{identity law}: &\quad \text{cmap}\,(\text{id}) = \text{id} \quad , \\
\text{composition law}: &\quad \text{cmap}\,(f) \fatsemi \text{cmap}\,(g) = \text{id} \fatsemi \text{id} = \text{id} = \text{cmap}\,(g \fatsemi f) \quad .
\end{aligned}$$

Construction	Type notation	Comment
tuple	$C^A \stackrel{\text{def}}{=} P^A \times Q^A$	the product contrafunctor; P and Q must be contrafunctors
disjunctive type	$C^A \stackrel{\text{def}}{=} P^A + Q^A$	the co-product contrafunctor; P and Q must be contrafunctors
function type	$C^A \stackrel{\text{def}}{=} L^A \to H^A$	the exponential contrafunctor; L is a functor and H a contrafunctor
type parameter	$C^A \stackrel{\text{def}}{=} Z$	the constant contrafunctor; Z is a fixed type
type constructor	$C^A \stackrel{\text{def}}{=} P^{Q^A}$	the composition; P is a functor and Q a contrafunctor (or vice versa)
recursive type	$C^A \stackrel{\text{def}}{=} S^{A,C^A}$	$S^{A,B}$ must be a contrafunctor w.r.t. A and functor w.r.t. B

Table 6.4: Type constructions defining a contrafunctor C^A.

Statement 6.2.4.2 If P^A is a functor and Q^A is a contrafunctor then $L^A \stackrel{\text{def}}{=} P^{Q^A}$ is a contrafunctor with `cmap` defined by

```
def cmap[A, B](f: B => A): P[Q[A]] => P[Q[B]] = fmap_P(cmap_Q(f))
```

where lawful implementations of fmap_P and cmap_Q are assumed to be given.

Proof Convert the Scala implementation of cmap_L into the code notation:

$$\text{cmap}_L(f^{:B\to A}) \stackrel{\text{def}}{=} \text{fmap}_P(\text{cmap}_Q(f)) \quad .$$

It is easier to reason about this function if we rewrite it as:

$$f^{\downarrow L} \stackrel{\text{def}}{=} \left(f^{\downarrow Q}\right)^{\uparrow P} \quad .$$

The contrafunctor laws for L are then proved like this:

$$\begin{aligned}
&\text{identity law}: && \text{id}^{\downarrow L} = (\text{id}^{\downarrow Q})^{\uparrow P} = \text{id}^{\uparrow P} = \text{id} \quad . \\
&\text{composition law}: && f^{\downarrow L} \fatsemi g^{\downarrow L} = (f^{\downarrow Q})^{\uparrow P} \underline{\fatsemi} (g^{\downarrow Q})^{\uparrow P} \\
&\text{use } P\text{'s composition law}: && = \left(\underline{f^{\downarrow Q} \fatsemi g^{\downarrow Q}}\right)^{\uparrow P} = \left(\underline{(g \fatsemi f)^{\downarrow Q}}\right)^{\uparrow P} = (g \fatsemi f)^{\downarrow L} \quad .
\end{aligned}$$

Finally, the recursive construction works for contrafunctors, except that the type constructor $S^{A,R}$ must be a contrafunctor in A (but still a functor in R). An example of such a type constructor is:

$$S^{A,R} \stackrel{\text{def}}{=} (A \to \text{Int}) + R \times R \quad . \tag{6.22}$$

The type constructor $S^{\bullet,\bullet}$ is not a bifunctor because it is contravariant in its first type parameter; so we cannot define a `bimap` function for it. However, we can define an analogous function called `xmap` with the type signature:

```
def xmap[A, B, Q, R](f: B => A)(g: Q => R): S[A, Q] => S[B, R]
```

$$\begin{aligned}
&\text{xmap}_S : (B \to A) \to (Q \to R) \to S^{A,Q} \to S^{B,R} \quad , \\
&\text{xmap}_S(f^{:B\to A})(g^{:Q\to R}) \stackrel{\text{def}}{=} \text{fmap}_{S^{A,\bullet}}(g) \fatsemi \text{cmap}_{S^{\bullet,R}}(f) \quad .
\end{aligned}$$

The type constructor S is called a **profunctor** if its `xmap` obeys the laws of identity and composition:

$$\text{identity law}: \quad \text{xmap}_S(\text{id})(\text{id}) = \text{id} \quad , \tag{6.23}$$

$$\text{composition law}: \quad \text{xmap}_S(f_1)(g_1) \fatsemi \text{xmap}_S(f_2)(g_2) = \text{xmap}_S(f_2 \fatsemi f_1)(g_1 \fatsemi g_2) \quad . \tag{6.24}$$

These laws are similar to the identity and composition laws for bifunctors (Section 6.2.2) except for inverting the order of the composition $(f_2 \fatsemi f_1)$. The laws hold automatically whenever the functor

and contrafunctor methods for S ($\text{fmap}_{S^{A,\bullet}}$ and $\text{cmap}_{S^{\bullet,R}}$) are fully parametric. We omit the details since they are quite similar to what we saw in Section 6.2.2 for bifunctors.

If we define a type constructor $L^\bullet$ using the recursive "type equation":

$$L^A \stackrel{\text{def}}{=} S^{A,L^A} \stackrel{\text{def}}{=} (A \to \text{Int}) + L^A \times L^A \quad ,$$

we obtain a contrafunctor in the shape of a binary tree whose leaves are functions of type $A \to \text{Int}$. The next statement shows that recursive type equations of this kind always define contrafunctors.

Statement 6.2.4.3 If $S^{A,R}$ is a contrafunctor with respect to A and a functor with respect to R then the recursively defined type constructor C^A is a contrafunctor:

$$C^A \stackrel{\text{def}}{=} S^{A,C^A} \quad .$$

Given the functions $\text{cmap}_{S^{\bullet,R}}$ and $\text{fmap}_{S^{A,\bullet}}$ for S, we implement cmap_C as

$$\begin{aligned} &\text{cmap}_C(f^{:B\to A}) : C^A \to C^B \cong S^{A,C^A} \to S^{B,C^B} \quad , \\ &\text{cmap}_C(f^{:B\to A}) \stackrel{\text{def}}{=} \text{xmap}_S(f)(\overline{\text{cmap}_C}(f)) \quad . \end{aligned}$$

The corresponding Scala code can be written as:

```
final case class C[A](x: S[A, C[A]]) // The type constructor S[_, _] must be defined previously.

def xmap_S[A,B,Q,R](f: B => A)(g: Q => R): S[A, Q] => S[B, R] = ???          // Must be defined.

def cmap_C[A, B](f: B => A): C[A] => C[B] = { case C(x) =>
  val sbcb: S[B, C[B]] = xmap_S(f)(cmap_C(f))(x)                    // Recursive call to cmap_C.
  C(sbcb)              // Need to wrap the value of type S[B, C[B]] into the type constructor C.
}
```

Proof The code of `cmap` is recursive, and the recursive call is marked by an overline:

$$\text{cmap}_C(f) \stackrel{\text{def}}{=} f^{\downarrow C} \stackrel{\text{def}}{=} \text{xmap}_S(f)(\overline{\text{cmap}_C}(f)) \quad .$$

To verify the identity law:

$$\begin{aligned} \text{expect to equal id}: \quad & \text{cmap}_C(\text{id}) = \text{xmap}_S(\text{id})(\underline{\overline{\text{cmap}_C}(\text{id})}) \\ \text{inductive assumption}: \quad & = \text{xmap}_S(\text{id})(\text{id}) \\ \text{identity law of xmap}_S: \quad & = \text{id} \quad . \end{aligned}$$

To verify the composition law:

$$\begin{aligned} \text{expect to equal } (g^{\downarrow C} \fatsemi f^{\downarrow C}): \quad & (f^{:D\to B} \fatsemi g^{:B\to A})^{\downarrow C} = \text{xmap}_S(f \fatsemi g)(\underline{\overline{\text{cmap}_C}(f \fatsemi g)}) \\ \text{inductive assumption}: \quad & = \text{xmap}_S(f \fatsemi g)(\overline{\text{cmap}_C}(g) \fatsemi \overline{\text{cmap}_C}(f))) \\ \text{composition law of xmap}_S: \quad & = \text{xmap}_S(g)(\overline{\text{cmap}_C}(g)) \fatsemi \text{xmap}_S(f)(\overline{\text{cmap}_C}(f)) \\ \text{definition of } {}^{\downarrow C}: \quad & = g^{\downarrow C} \fatsemi f^{\downarrow C} \quad . \end{aligned}$$

6.2.5 Solved examples: How to recognize functors and contrafunctors

Sections 6.2.3 and 6.2.4 describe how functors and contrafunctors are built from other type expressions. We can see from Tables 6.2 and 6.4 that *every* one of the six basic type constructions (unit type, type parameters, product types, co-product types, function types, recursive types) gives either a new functor or a new contrafunctor. The six type constructions generate all exponential-polynomial types, including recursive ones. So, we should be able to decide whether any given exponential-polynomial type expression is a functor or a contrafunctor. The decision algorithm is based on the results shown in Tables 6.2 and 6.4:

- Primitive types $\mathbb{1}$, `Int`, `String`, etc., can be viewed both as constant functors and as constant contrafunctors (since they do not contain type parameters).

- Polynomial type expressions (not containing any function arrows) are always functors with respect to every type parameter. Equivalently, we may say that all polynomial type constructors are covariant in every type parameter. For example, the type expression $A \times B + (A + \mathbb{1} + B) \times A \times C$ is covariant in each of the type parameters A, B, C.

- Type parameters to the right of a function arrow are in a covariant position. For example, $F^A \stackrel{\text{def}}{=} \text{Int} \to A$ is covariant in A.

- Each time a type parameter is placed to the left of an *uncurried* function arrow $\to$, the variance is reversed: covariant becomes contravariant and vice versa. For example:

$$\begin{aligned}
\text{this is covariant in } A: &\quad \mathbb{1} + A \times A \quad , \\
\text{this is contravariant in } A: &\quad (\mathbb{1} + A \times A) \to \text{Int} \quad , \\
\text{this is covariant in } A: &\quad ((\mathbb{1} + A \times A) \to \text{Int}) \to \text{Int} \quad , \\
\text{this is contravariant in } A: &\quad (((\mathbb{1} + A \times A) \to \text{Int}) \to \text{Int}) \to \text{Int} \quad .
\end{aligned}$$

  Indeed, Statement 6.2.3.5 proves that any contravariant type expression (i.e., a contrafunctor) placed to the left of a function arrow will create a covariant functor. One can prove in a similar way that a function arrow turns covariance into contravariance (Exercise 6.3.1.1).

- Repeated curried function arrows work as one arrow: $A \to \text{Int}$ is contravariant in A, and $A \to A \to A \to \text{Int}$ is still contravariant in A. This is because the type $A \to A \to A \to \text{Int}$ is equivalent to $A \times A \times A \to \text{Int}$, which is of the form $F^A \to \text{Int}$ with a type constructor $F^A \stackrel{\text{def}}{=} A \times A \times A$. Exercise 6.3.1.1 will show that $F^A \to \text{Int}$ is contravariant in A.

- Nested type constructors combine their variances: e.g., if F^A is contravariant in A then $F^{A \to \text{Int}}$ is covariant in A, while $F^{A \times A \times A}$ is contravariant in A. This is shown in Exercises 6.3.1.2(c), (d).

To see how this works, consider any exponential-polynomial type expression, such as Eq. (6.7):

$$Z^{A,R} \stackrel{\text{def}}{=} ((A \to A \to R) \to R) \times A + (\mathbb{1} + R \to A + \text{Int}) + A \times A \times \text{Int} \times \text{Int} \quad ,$$

Mark the position of each type parameter as either covariant (+) or contravariant (−), according to the number of nested *uncurried* function arrows:

$$\big(\big(\underset{+}{A} \to \underset{+}{A} \to \underset{-}{R}\big) \to \underset{+}{R}\big) \times \underset{+}{A} + \big(\mathbb{1} + \underset{-}{R} \to \underset{+}{A} + \text{Int}\big) + \underset{+}{A} \times \underset{+}{A} \times \text{Int} \times \text{Int} \quad .$$

We find that A is always in covariant positions, while R is sometimes in covariant and sometimes in contravariant positions. So, we expect that $Z^{A,R}$ is a functor with respect to A, but not a functor (nor a contrafunctor) with respect to R.

To show that $Z^{A,R}$ is indeed a functor in the parameter A, we need to implement a suitable `map` method and verify that the functor laws hold. To do that from scratch, we could use the techniques explained in this and the previous chapters: starting from the type signature

$$\text{map}_Z : Z^{A,R} \to (A \to B) \to Z^{B,R} \quad ,$$

we could derive a fully parametric, information-preserving implementation of `map`. We could then look for proofs of the identity and composition laws for that `map` function. This would require a lot of work for a complicated type constructor such as $Z^{A,R}$.

However, that work can be avoided if we find a way of building up $Z^{A,R}$ step by step via the known functor and contrafunctor constructions. Each step automatically provides both a fragment of the code of `map` and a proof that the functor laws hold up to that step. In this way, we will avoid the need to look for an implementation of `map` or prove the laws for each new functor and contrafunctor. The next examples illustrate this procedure on a simpler type constructor.

Example 6.2.5.1 Rewrite these Scala definitions in the type notation and decide whether F and G are covariant or contravariant with respect to each type parameter:

```
final case class F[A, Z](f: Int => Z => (Int, A))
final case class G[A, Z](p: Either[Int, A], q: Option[Z => F[A, Z]])
```

Solution The type notation for F and G is:

$$F^{A,Z} \stackrel{\text{def}}{=} \text{Int} \to Z \to \text{Int} \times A \quad ,$$
$$G^{A,Z} \stackrel{\text{def}}{=} (\text{Int} + A) \times (\mathbb{1} + (Z \to F^{A,Z})) = (\text{Int} + A) \times (\mathbb{1} + (Z \to \text{Int} \to Z \to \text{Int} \times A)) \quad .$$

Mark the covariant and the contravariant positions in the type expressions for F and G:

$$F^{A,Z} = \text{Int} \to \underset{-}{Z} \to \text{Int} \times \underset{+}{A} \quad , \qquad G^{A,Z} = (\text{Int} + \underset{+}{A}) \times (\mathbb{1} + (\underset{-}{Z} \to \text{Int} \to \underset{-}{Z} \to \text{Int} \times \underset{+}{A})) \quad .$$

All Z positions in the sub-expression $Z \to \text{Int} \to Z \to \text{Int} \times A$ are contravariant since the function arrows are curried rather than nested. So, A is always in covariant positions (+) while Z is always in contravariant positions (−). It follows that $F^{A,Z}$ and $G^{A,Z}$ are covariant in A but contravariant in Z.

Example 6.2.5.2 Use known functor constructions to implement the `map` method with respect to `A` for the type `G[A, Z]` from Example 6.2.5.1.

Solution We need to build $G^{A,Z}$ via step-by-step constructions that start from primitive types and type parameters. At the top level of its type expression, $G^{A,Z}$ is a product type. So, we begin by using the "functor product" construction (Statement 6.2.3.3):

$$G^{A,Z} \cong G_1^A \times G_2^{A,Z} \quad ,$$
$$\text{where} \quad G_1^A \stackrel{\text{def}}{=} \text{Int} + A \quad \text{and} \quad G_2^{A,Z} \stackrel{\text{def}}{=} \mathbb{1} + (Z \to \text{Int} \to Z \to \text{Int} \times A) \quad .$$

We continue with G_1^A, which is a co-product of Int (a constant functor) and A (the identity functor). The constant functor and the identity functor have lawful `map` implementations that are already known (Statements 6.2.3.1–6.2.3.2). Now, the "functor co-product" construction (Statement 6.2.3.3) produces a `map` implementation for G_1^A together with a proof that it satisfies the functor laws:

$$\text{fmap}_{G_1}(f^{:A\to B}) = f^{\uparrow G_1} \stackrel{\text{def}}{=} \left| \begin{array}{c||cc} & \text{Int} & B \\ \hline \text{Int} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & f \end{array} \right| \quad .$$

Turning our attention to $G_2^{A,Z}$, we find that it is a disjunctive type containing a curried function type that ultimately returns the product type $\text{Int} \times A$. This tells us to use the functor constructions for "co-product", "exponential", and "product". Write down the functor constructions needed at each step as we decompose $G_2^{A,Z}$:

$$G_2^{A,Z} \stackrel{\text{def}}{=} \mathbb{1} + (Z \to \text{Int} \to Z \to \text{Int} \times A) \quad .$$
$$\text{co-product}: \quad G_2^{A,Z} \cong \mathbb{1} + G_3^{A,Z} \quad \text{where } G_3^{A,Z} \stackrel{\text{def}}{=} Z \to \text{Int} \to Z \to \text{Int} \times A \quad .$$
$$\text{exponential}: \quad G_3^{A,Z} \cong Z \to G_4^{A,Z} \quad \text{where } G_4^{A,Z} \stackrel{\text{def}}{=} \text{Int} \to Z \to \text{Int} \times A \quad .$$
$$\text{exponential}: \quad G_4^{A,Z} \cong \text{Int} \to G_5^{A,Z} \quad \text{where } G_5^{A,Z} \stackrel{\text{def}}{=} Z \to \text{Int} \times A \quad .$$
$$\text{exponential}: \quad G_5^{A,Z} \cong Z \to G_6^{A} \quad \text{where } G_6^{A} \stackrel{\text{def}}{=} \text{Int} \times A \quad .$$
$$\text{product}: \quad G_6^{A} \cong \text{Int} \times A \cong \text{Const}^{\text{Int},A} \times \text{Id}^{A} \quad .$$

Each of the type constructors G_1, ..., G_6 is a functor in A because all of the functor constructions preserve the functor laws. Therefore, $G^{A,Z}$ is a functor in A.

It remains to derive the code for the `fmap` method of G. Each of the functor constructions combines the `fmap` implementations from previously defined functors into a new `map` implementation, so we just need to combine the code fragments in the order of constructions. For brevity, we will use the notations $f^{\uparrow L} \stackrel{\text{def}}{=} \text{fmap}_L(f)$ and $x \triangleright f^{\uparrow L}$ instead of the Scala code `x.map(f)` throughout the derivations:

$$\text{product}: \quad G^{A,Z} \cong G_1^A \times G_2^{A,Z} \quad , \qquad (g_1 \times g_2) \triangleright f^{\uparrow G} = (g_1 \triangleright f^{\uparrow G_1}) \times (g_2 \triangleright f^{\uparrow G_2}) \quad .$$

$$\text{co-product}: \quad G_1^A \stackrel{\text{def}}{=} \text{Int} + A \quad , \qquad f^{\uparrow G_1} = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & f \end{array} \right| \quad .$$

$$\text{co-product}: \quad G_2^{A,Z} \stackrel{\text{def}}{=} \mathbb{1} + G_3^{A,Z} \quad , \qquad f^{\uparrow G_2} = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & f^{\uparrow G_3} \end{array} \right| \quad .$$

$$\text{exponential}: \quad G_3^{A,Z} \stackrel{\text{def}}{=} Z \to G_4^{A,Z} \quad , \qquad g_3 \triangleright f^{\uparrow G_3} = g_3 \fatsemi f^{\uparrow G_4} = z^{:Z} \to z \triangleright g_3 \triangleright f^{\uparrow G_4} \quad .$$

The pipe symbol ($\triangleright$) groups stronger than the function arrow. So, $z \to z \triangleright g \triangleright h$ means $z \to (z \triangleright g \triangleright h)$, or $z \to h(g(z))$. Applying the exponential construction three times, we finally obtain:

$$G_3^{A,Z} \stackrel{\text{def}}{=} Z \to \text{Int} \to Z \to G_6^A \quad , \qquad g_3 \triangleright f^{\uparrow G_3} = z_1^{:Z} \to n^{:\text{Int}} \to z_2^{:Z} \to g_3(z_1)(n)(z_2) \triangleright f^{\uparrow G_6} \quad .$$

$$G_6^A \stackrel{\text{def}}{=} \text{Int} \times A \quad , \qquad (i \times a) \triangleright f^{\uparrow G_6} = i \times f(a) \quad .$$

We can now write the corresponding Scala code for fmap_G:

```
def fmap_G[A, B, Z](f: A => B): G[A, Z] => G[B, Z] = { case G(p, q) =>
  val newP: Either[Int, B] = p.map(f)      // Use the standard map method for Either[Int, A].
  val newQ: Option[Z => Int => Z => (Int, B)]  = q.map { // Use the map method for Option[_].
    (g3: Z => Int => Z => (Int, A)) =>
      z1 => n => z2 =>                     // The code of map for G_3.
        val (i, a) = g3(z1)(n)(z2)
        (i, f(a))                          // The code of map for G_6.
    }
  G(newP, newQ)                            // The code of map for G_1.
}
```

In this way, the correct-by-construction code of fmap_G may be *derived* for any functor G given the type expression of G^A, and similarly the code for cmap_C for any contrafunctor C. The corresponding algorithms could be implemented as a Scala library that derives the code at compile time.

6.3 Summary

What tasks can we perform with the techniques of this chapter?

- Quickly decide if a given type constructor is a functor, a contrafunctor, or neither.
- Implement a `fmap` or a `cmap` function that satisfies the appropriate laws.
- Use constructions to derive the correct code of `fmap` or `cmap` without trial and error.
- Use functor blocks to manipulate data wrapped in functors with more readable code.

6.3.1 Exercises: Functor and contrafunctor constructions

Exercise 6.3.1.1 If H^A is a contrafunctor and L^A is a functor, show that $C \stackrel{\text{def}}{=} L^A \to H^A$ is a contrafunctor with the `cmap` method defined by the following code:

```
def cmap[A, B](f: B => A)(c: L[A] => H[A]): L[B] => H[B] = {
  lb: L[B] => cmap_H(f)(c(fmap_L(f)(lb)))                    // Code notation: f↓C def= c→f↑L ⨾ c ⨾ f↓H
}
```

Here, `cmap_H` and `fmap_L` are the methods already defined for H and L. Prove that the laws hold.

Exercise 6.3.1.2 Implement the required `fmap` or `cmap` function for the given type constructors L and prove that the appropriate laws hold. Write the implementations both in Scala and in the code notation. Assume that the given type constructors F and G already satisfy their respective laws.

(a) $L^A \stackrel{\text{def}}{=} F^A \times G^A$ is a contrafunctor if F^A and G^A are contrafunctors.
(b) $L^A \stackrel{\text{def}}{=} F^A + G^A$ is a contrafunctor if F^A and G^A are contrafunctors.
(c) $L^A \stackrel{\text{def}}{=} F^{G^A}$ is a functor when both F and G are contrafunctors.
(d) $L^A \stackrel{\text{def}}{=} F^{G^A}$ is a contrafunctor when F is a contrafunctor and G is a functor.

Exercise 6.3.1.3 Show that $L^A \stackrel{\text{def}}{=} F^A \to G^A$ is, in general, neither a functor nor a contrafunctor if both F^A and G^A are functors or both are contrafunctors (give an example of suitable F^A and G^A).

Exercise 6.3.1.4 For each of the Scala type constructors defined below, formulate the definition in the type notation and decide whether the type constructors are functors, contrafunctors, or neither:

```
type F[A] = Int => (Option[A], Either[A, Int], Either[A, A])
type G[A] = ((Int, A)) => Either[Int, A]
type H[A] = Either[A, (A, Option[A])] => Int => Int
```

Exercise 6.3.1.5 Using the known constructions, determine which of the following are functors or contrafunctors (or neither) and implement `fmap` or `cmap` if appropriate. Answer this question with respect to each type parameter separately.

(a) $F^A \stackrel{\text{def}}{=} \text{Int} \times A \times A + (\text{String} \to A) \times A$.
(b) $G^{A,B} \stackrel{\text{def}}{=} (A \to \text{Int} \to \mathbb{1} + B) + (A \to \mathbb{1} + A \to \text{Int})$.
(c) $H^{A,B,C} \stackrel{\text{def}}{=} (A \to A \to B \to C) \times C + (B \to A)$.
(d) $P^{A,B} \stackrel{\text{def}}{=} (((A \to B) \to A) \to B) \to A$.

Exercise 6.3.1.6 Show that the recursive type constructor $L^\bullet$ defined by:

$$L^A \stackrel{\text{def}}{=} \mathbb{1} + A + L^A$$

is a functor, and implement a `map` or `fmap` function for L in Scala.

Exercise 6.3.1.7 Show that the perfect-shaped tree $L^\bullet$ defined by:

$$L^A \stackrel{\text{def}}{=} A \times A + L^{A \times A \times A}$$

is a functor, and implement a `map` or `fmap` function for L in Scala.

Exercise 6.3.1.8 Show that the type constructor L defined by Eq. (6.21) is a functor for any given bifunctor S and functor P.

Exercise 6.3.1.9 **(a)** Given any functor F, show that if two types A and B are equivalent then the types $F^A$and F^B are also equivalent. **(b)** Show that the same property holds for any contrafunctor F.

6.4 Further developments

6.4.1 Profunctors

We have seen that some type constructors are neither functors nor contrafunctors because their type parameters appear both in covariant and contravariant positions. An example of such a type constructor is:

$$P^A \stackrel{\text{def}}{=} A + (A \to \text{Int}) \quad .$$

It is not possible to define either a `map` or a `cmap` function for P: the required type signatures cannot be implemented. However, we *can* implement a function called `xmap` with the type signature:

$$\text{xmap}_P : (B \to A) \to (A \to B) \to P^A \to P^B \quad .$$

To see why, let us temporarily rename the contravariant occurrence of A to Z and define a new type constructor $\tilde{P}$ by:

$$\tilde{P}^{Z,A} \stackrel{\text{def}}{=} A + (Z \to \text{Int}) \quad .$$

The original type constructor P^A is expressed as $P^A = \tilde{P}^{A,A}$. Now, $\tilde{P}^{Z,A}$ is covariant in A and contravariant in Z. We can implement $\texttt{xmap}_{\tilde{P}}$ as a composition of `fmap` with respect to A and `cmap` with respect to Z, similarly to what we saw in the proof of Statement 6.2.4.3. The function $\texttt{xmap}_{\tilde{P}}$ will satisfy the identity and composition laws (6.23)–(6.24). Setting the type parameter $Z = A$, we will obtain the $\texttt{xmap}_P$ function for P. The identity and composition laws for $\texttt{xmap}_P$ will hold, since the laws of $\tilde{P}^{Z,A}$ hold for all type parameters:

$$\text{P's identity law}: \quad \text{xmap}_P(\text{id}^{:A\to A})(\text{id}^{:A\to A}) = \text{id} \quad ,$$

$$\text{P's composition law}: \quad \text{xmap}_P(f_1^{:B\to A})(g_1^{:A\to B}) \fatsemi \text{xmap}_P(f_2^{:C\to B})(g_2^{:B\to C}) = \text{xmap}_P(f_2 \fatsemi f_1)(g_1 \fatsemi g_2) \quad .$$

A type constructor $\tilde{P}^{Z,A}$ with a lawful $\texttt{xmap}_{\tilde{P}}$ is called a profunctor. We will sometimes also call the type constructor $P^A \stackrel{\text{def}}{=} \tilde{P}^{A,A}$ a profunctor.

Consider an exponential-polynomial type constructor P^A, no matter how complicated, such as:

$$P^A \stackrel{\text{def}}{=} (\mathbb{1} + A \times A \to A) \times A \to \mathbb{1} + (A \to A + \text{Int}) \quad .$$

Each copy of the type parameter A will occur either in covariant or in a contravariant position because no other possibility is available in exponential-polynomial types. So, we can always rename all contravariant occurrences of the type parameter A to Z and so obtain a new type constructor $\tilde{P}^{Z,A}$, which will be covariant in A and contravariant in Z. Since $\tilde{P}^{A,Z}$ is a functor in A and a contrafunctor in Z, we will be able to define a function $\texttt{xmap}_{\tilde{P}}$ satisfying the identity and composition laws. Setting $Z = A$, we will obtain a lawful $\texttt{xmap}_P$, which makes P a profunctor. Thus, *every* exponential-polynomial type constructor is a profunctor.

An unfunctor (such as the disjunctive type `ServerAction[R]` shown in Section 6.1.6) cannot be made into a profunctor. The type signature of `xmap` cannot be implemented for `ServerAction[R]` because it is not a fully parametric type constructor (and so is not exponential-polynomial).

Profunctors are not often used in practical coding. We will see profunctors occasionally in later chapters where we need to reason about type constructors of arbitrary variance.

6.4.2 Subtyping with injective or surjective conversion functions

When P is a subtype of Q (denoted as $P \lesssim Q$), sometimes the set of values of P is a *subset* of values of Q. In that case, the subtype conversion function $P \to Q$ is injective and embeds all information from a value of type P into a value of type Q. This kind of subtyping works for parts of disjunctive types, such as `Some[A] <: Option[A]` (in the type notation, $\mathbb{0} + A \lesssim \mathbb{1} + A$). The set of all values of type `Some[A]` is a subset of the set of values of type `Option[A]`. Then the conversion function is injective because it is an identity function, $\mathbb{0} + x^{:A} \to \mathbb{0} + x$, that merely reassigns types.

However, subtyping does not always mean that the conversion function is injective. An example of subtyping with a *surjective* conversion function is between the function types $P \stackrel{\text{def}}{=} \mathbb{1} + A \to \text{Int}$ and $Q \stackrel{\text{def}}{=} \mathbb{0} + A \to \text{Int}$ (in Scala, `P = Option[A] => Int` and `Q = Some[A] => Int`). We have $P \lesssim Q$ because $P \cong C^{\mathbb{1}+A}$ and $Q \cong C^{\mathbb{0}+A}$, where $C^X \stackrel{\text{def}}{=} X \to \text{Int}$ is a contrafunctor. The conversion function $P \to Q$ is an identity function that reassigns types:

```
def p2q[A](p: Option[A] => Int): Some[A] => Int = { x: Some[A] => p(x) }
```

In the code notation, $p \to x \to p(x)$ is just an expanded form of the identity function $p \to p$.

Note that functions of type P accept `None` as an argument, while functions of type Q do not. So, there are *more values* of type P than values of type Q. The conversion function $\text{p2q} : P \to Q$ is surjective: for any function f of type `Some[A] => Int`, we can find a function g of type `Option[A] => Int` such that $\text{p2q}(g) = f$. (The function g is defined as `g(Some(x)) = f(x)` and `g(None) = 123`, say.)

Another example of subtyping with a surjective subtype conversion function is found when using object-oriented inheritance. Consider this code where a class inherits a value from a parent class:

```
sealed class HasID(val id: Long)
final case class Person(name: String, override val id: Long) extends HasID(id)
```

The type `Person` is a subtype of `HasID`; the conversion function of type `Person => HasID` simply forgets that `name` is defined. In the type notation, `HasID` $\cong$ Long and `Person` $\cong$ String $\times$ Long. The subtype conversion function is $\pi_2 : \text{String} \times \text{Long} \to \text{Long}$, which is a surjective mapping.

Sometimes, type conversion functions are neither injective nor surjective. As an example, let $P_1 \lesssim Q_1$ and $P_2 \lesssim Q_2$, where the conversion function $r_1 : P_1 \to Q_1$ is injective but $r_2 : P_2 \to Q_2$ is surjective (as we have seen, this can happen). Now consider the product types $P_1 \times P_2$ and $Q_1 \times Q_2$. The product type $A \times B$ is covariant in both type parameters, so $P_1 \times P_2 \lesssim Q_1 \times Q_2$. The conversion function is the pair product $r_1 \boxtimes r_2 : P_1 \times P_2 \to Q_1 \times Q_2$ and is neither injective nor surjective.

Functors preserve subtyping; it turns out that functor liftings also preserve injectivity and surjectivity. If a function $f^{:A\to B}$ is injective, it is lifted to an injective function $f^{\uparrow L} : L^A \to L^B$, and similarly for surjective functions f. We conclude this chapter by proving this property.

Statement 6.4.2.1 If L^A is a lawful functor and $f^{:A\to B}$ is an injective function then $\text{fmap}_L(f)$ is also an injective function of type $L^A \to L^B$. If f is surjective then the function $\text{fmap}_L(f)$ is also surjective.

Proof If the function $f^{:A\to B}$ is injective, it must somehow embed all information from a value of type A into a value of type B. The **image** of f (the set of all values of type B that can be obtained as $f(a)$ for some $a^{:A}$) must contain a distinct value of type B for each distinct value of type A. So, there must exist a function that maps any b from the image of f back to a value $a^{:A}$ it came from; call that function $g^{:B\to A}$. The function g must satisfy the following law:

$$g(f(a)) = a \quad \text{for all } a^{:A},$$

which we can write more concisely as:

$$g \circ f = \text{id} \quad , \quad \text{or equivalently :} \quad f \fatsemi g = \text{id} \quad .$$

We note that g may be a partial function: g needs to be defined only for a subset of values of type B, namely the values within the image of f. Despite the equation $f \fatsemi g = \text{id}$, the function g is not an inverse of f. (An inverse function of f must be a *total* (not a partial) function h satisfying $f \fatsemi h = \text{id}$ and $h \fatsemi f = \text{id}$.) The function g is called a **left inverse** of f.

The fact that f has a left inverse is *equivalent* to the property that f is injective. Indeed, if any function f has a left inverse g, we can show that f is injective. Assume some x and y such that $f(x) = f(y)$; we will prove that f is injective if we show that $x = y$. Applying g to both sides of the equation $f(x) = f(y)$, we get:

$$x = g(f(x)) = g(f(y)) = y \quad .$$

Now we apply this trick to functions lifted to the functor L. To prove that $\text{fmap}_L(f)$ is injective, we need to show that it has a left inverse. We can lift both sides of the equation $f \fatsemi g = \text{id}$ to get:

$$\begin{aligned}
& \text{fmap}_L(f) \fatsemi \text{fmap}_L(g) \\
\text{composition law of } L : \quad & = \text{fmap}_L(f \fatsemi g) \\
\text{use } f \fatsemi g = \text{id} : \quad & = \text{fmap}_L(\text{id}) \\
\text{identity law of } L : \quad & = \text{id} \quad .
\end{aligned}$$

It follows that $\text{fmap}_L(g) \circ \text{fmap}_L(f) = \text{id}$. In other words, $\text{fmap}_L(g)$ is a left inverse for $\text{fmap}_L(f)$. Since $\text{fmap}_L(f)$ has a left inverse, it is injective.

Now consider surjective functions f. By definition, the image of a surjective function $f^{:A\to B}$ is the entire set of values of type B. So, there must exist a (total) function $g^{:B\to A}$ that maps each value $b^{:B}$ to some $a^{:A}$ such that $f(a) = b$. In other words, we must have:

$$f(g(b)) = b \quad \text{for all } b^{:B} \quad , \qquad \text{or equivalently :} \qquad g \fatsemi f = \text{id} \quad .$$

So, g is a **right inverse** for f. The existence of a right inverse is equivalent to the property that f is a surjective function.

Lifting the law $g \fatsemi f = \text{id}$ to the functor L, we obtain:

$$\text{fmap}_L(g) \fatsemi \text{fmap}_L(f) = \text{fmap}_L(g \fatsemi f) = \text{fmap}_L(\text{id}) = \text{id} \quad .$$

It follows that the function $\text{fmap}_L(f)$ has a right inverse. So, $\text{fmap}_L(f)$ is surjective.

7 Reasoning about code. Techniques of symbolic derivation

In previous chapters, we have performed symbolic derivations of some laws. To make the derivations more manageable, we gradually developed special notations and techniques of reasoning. This short chapter is a summary of those notations and techniques.

7.1 This book's notation for code

7.1.1 The nine constructions of fully parametric code

The eight basic constructions introduced in Section 5.2.2, together with recursion, serve as a foundation for **fully parametric** coding style. All major techniques and design patterns of functional programming can be implemented using only these constructions, i.e., by fully parametric code. We will now define the code notation (summarized in Table 7.1) for each of the nine constructions.

1) Use a constant Whenever we need a value of a fixed type, e.g., `Int`, `String`, or `Unit`, we may use a constant value of that type. We may also use a "named unit", e.g., `None` of type `Option[A]` for any type `A`. All named unit values are denoted by 1 and are viewed as having type $\mathbb{1}$. Functions taking `Unit` type as an argument do not actually need to use their argument and may be written as _ → expr.

Functions that ignore their argument are called **constant functions** and are denoted by:

```
def c_1[A](x: A): Int = 123
```

$$c_1(x^{:A}) \stackrel{\text{def}}{=} 123 \quad \text{or} \quad c_1 \stackrel{\text{def}}{=} _^{:A} \to 123 \quad .$$

2) Use a given argument In any expression that has a bound variable (e.g., an argument within a function's body), we may use the bound variable at any place, as many times as we need.

```
def c_2[A](x: String, y: Int): Int = 123 + y + y
```

$$c_2(x^{:\text{String}}, y^{:\text{Int}}) \stackrel{\text{def}}{=} 123 + y + y \quad .$$

3) Create a function We can always make a nameless function `{ x => expr }` out of a variable, say `x`, and any expression `expr` that may use `x` as a free variable (i.e., a variable that should be defined outside that expression). E.g., the expression `123 + x + x` uses `x` as a free variable because `123 + x + x` only makes sense if `x` is already defined. So, we can create a nameless function:

```
{ x: Int => 123 + x + x }
```

$$x^{:\text{Int}} \to 123 + x + x \quad .$$

If the expression `expr` already contains `x` as a bound variable, the function `{ x => expr }` will have a name clash. As an example, consider an expression `expr == { x => x }` that already contains a nameless function with bound variable `x`. If we want to make a function out of that expression, we could write `x => { x => x }`, but such code is confusing. It is helpful to avoid the name clash by renaming the bound variables inside `expr`, e.g., `expr == { z => z }`:

```
val f = { x: Int => { z: Int => z } }
```

$$f \stackrel{\text{def}}{=} x^{:\text{Int}} \to z^{:\text{Int}} \to z \quad .$$

4) Use a function If a function is already defined, we can use it by applying it to an argument.

```
val f = { x: Int => 123 + x + x }
f(100)        // Evaluates to 323.
```

$$f \stackrel{\text{def}}{=} x^{:\text{Int}} \to 123 + x + x \quad , \qquad f(100) = 323 \quad .$$

5) Create a tuple Given two values `a: A` and `b: B`, we can create the tuple `(a, b)` as well as `(b, a)`. In the code notation, those tuples are written as $a \times b$ and $b \times a$.

Constructions	Scala examples	Code notation
use a constant	`()` or `true` or `"abc"` or `123`	1, true, "abc", 123
use a given argument	`def f(x: A) = { ... x ... }`	$f(x^{:A}) \stackrel{\text{def}}{=} \ldots x \ldots$
create a function	`(x: A) => expr(x)`	$x^{:A} \to \text{expr}(x)$
use a function	`f(x)` or `x.pipe(f)` (Scala 2.13)	$f(x)$ or $x \triangleright f$
create a tuple	`val p: (A, B) = (a, b)`	$p^{:A\times B} \stackrel{\text{def}}{=} a \times b$
use a tuple	`p._1` or `p._2`	$p \triangleright \pi_1$ or $p \triangleright \pi_2$
create a disjunctive value	`Left[A, B](x)` or `Right[A, B](y)`	$x^{:A} + \mathbb{0}^{:B}$ or $\mathbb{0}^{:A} + y^{:B}$

use a disjunctive value	`val p: Either[A, B] = ...`
	`val q: C = p match {`
	`  case Left(x) => f(x)`
	`  case Right(y) => g(y)`
	`}`
		$q^{:C} \stackrel{\text{def}}{=} p^{:A+B} \triangleright \left\vert\begin{array}{c\|c} & C \\ \hline A & x^{:A} \to f(x) \\ B & y^{:B} \to g(y) \end{array}\right\vert$

use a recursive call	`def f(x) = { ... f(y) ... }`	$f(x) \stackrel{\text{def}}{=} \ldots \overline{f}(y) \ldots$

Table 7.1: This book's notation for the nine basic code constructions.

6) Use a tuple Given a tuple `p == (a, b)`, we can extract each of the values via `p._1` and `p._2`. The corresponding code notation is $p \triangleright \pi_1$ and $p \triangleright \pi_2$. The auxiliary functions π_i (where $i = 1, 2, \ldots$) may be used for tuples of any size. Example code defining these functions:

```
def pi_1[A, B]: ((A, B)) => A = {
    case (a, b) => a
} // Same as  p => p._1  or  _._1
def pi_2[A, B]: ((A, B)) => B = {
    case (a, b) => b
} // Same as  p => p._2  or  _._2
```

$$\pi_1^{A,B} \stackrel{\text{def}}{=} a^{:A} \times b^{:B} \to a \quad ,$$
$$\pi_2^{A,B} \stackrel{\text{def}}{=} a^{:A} \times b^{:B} \to b \quad .$$

The notation $a \times b$ is used in an *argument* of a function to destructure a tuple.

7) Create a disjunctive value Once a disjunctive type such as $A + B + C$ has been defined in Scala, its named "constructors" (i.e., case classes) are used to create values of that type:

```
sealed trait S
final case class P(w: Int, x: Int)  extends S
final case class Q(y: String)       extends S
final case class R(z: Int)          extends S

val s: S = P(10, 20) // Create a value of type S.
val t: S = R(30)     // Another value of type S.
```

$$S \stackrel{\text{def}}{=} \text{Int} \times \text{Int} + \text{String} + \text{Int} \quad ,$$

$$s^{:S} \stackrel{\text{def}}{=} 10 \times 20 + \mathbb{0}^{:\text{String}} + \mathbb{0}^{:\text{Int}} \quad ,$$
$$t^{:S} \stackrel{\text{def}}{=} \mathbb{0}^{:\text{Int}\times\text{Int}} + \mathbb{0}^{:\text{String}} + 30 \quad .$$

The code notation for disjunctive values, e.g., $\mathbb{0} + \mathbb{0} + x$, is more verbose than the Scala syntax such as `R(x)`. The advantage is that we may explicitly annotate all types and show clearly the part of the disjunction that we are creating. Another advantage is that the notation $\mathbb{0} + \mathbb{0} + x$ is similar to a row vector, $\left\vert \begin{array}{ccc} \mathbb{0} & \mathbb{0} & x \end{array} \right\vert$, which is well adapted to the matrix notation for functions.

8) Use a disjunctive value Once created, disjunctive values can be used as arguments of a pattern-matching expression (Scala's `match`/`case` syntax). Recall that functions that take a disjunctive value as an argument ("**disjunctive functions**") may be also written in Scala *without* the `match` keyword:

```
val compute: Option[Int] => Option[Int] = {
    case None      => Some(100)
    case Some(x)   => Some(x / 2)
}
```

$$\text{compute}^{:\mathbb{1}+\text{Int}\to\mathbb{1}+\text{Int}} \stackrel{\text{def}}{=} \left\vert\begin{array}{c\|cc} & \mathbb{1} & \text{Int} \\ \hline \mathbb{1} & \mathbb{0} & _ \to 100 \\ \text{Int} & \mathbb{0} & x \to \frac{x}{2} \end{array}\right\vert \quad .$$

We will use this example to explain how disjunctive functions are written in the matrix notation.

Each row of a matrix corresponds to a part of the disjunctive type matched by one of the `case` expressions. In this example, the disjunctive type `Option[Int]` has two parts: the named unit `None` (denoted by $\mathbb{1}$) and the case class `Some[Int]`, which is equivalent to the type `Int`. So, the matrix has two rows labeled $\mathbb{1}$ and Int, showing that the function's argument type is $\mathbb{1} + \text{Int}$.

The columns of the matrix correspond to the parts of the disjunctive type *returned* by the function. In this example, the return type is also `Option[Int]`, that is, $\mathbb{1} + \text{Int}$, so the matrix has two columns labeled $\mathbb{1}$ and Int. If the return type is not disjunctive, the matrix will have one column.

What are the matrix elements? The idea of the matrix notation is to translate the `case` expressions line by line from the Scala code. Look at the first `case` line as if it were a standalone partial function,

```
{ case None => Some(100) }
```

Since `None` is a named unit, this function is written in the code notation as $1 \to \mathbb{0}^{:\mathbb{1}} + 100^{:\text{Int}}$.

The second line is written in the form of a partial function as

```
{ case Some(x) => Some(x / 2) }
```

The pattern variable on the left side is `x`, so we can denote that function by $x^{:\text{Int}} \to \mathbb{0}^{:\mathbb{1}} + (x/2)^{:\text{Int}}$.

To obtain the matrix notation, we may simply write the two partial functions in the two rows:

```
val compute: Option[Int] => Option[Int] = {
    case None      => Some(100)
    case Some(x)   => Some(x / 2)
}
```

$$\text{compute}^{:\mathbb{1}+\text{Int}\to\mathbb{1}+\text{Int}} \stackrel{\text{def}}{=} \left|\begin{array}{c||c} & \mathbb{1}+\text{Int} \\ \hline \mathbb{1} & _ \to \mathbb{0}+100 \\ \text{Int} & x \to \mathbb{0}+\frac{x}{2} \end{array}\right| \quad .$$

This is already a valid matrix notation for the function f. So far, the matrix has two rows and one column. However, we notice that each row's return value is *known* to be in a specific part of the disjunctive type $\mathbb{1} + \text{Int}$ (in this example, both rows return values of type $\mathbb{0} + \text{Int}$). So, we can split the column into two columns and obtain a clearer and more useful notation for this function:

$$\text{compute}^{:\mathbb{1}+\text{Int}\to\mathbb{1}+\text{Int}} \stackrel{\text{def}}{=} \left|\begin{array}{c||cc} & \mathbb{1} & \text{Int} \\ \hline \mathbb{1} & \mathbb{0} & _ \to 100 \\ \text{Int} & \mathbb{0} & x^{:\text{Int}} \to \frac{x}{2} \end{array}\right| \quad .$$

The void type $\mathbb{0}$ is written symbolically to indicate that the disjunctive part in that column is not returned. In this way, the matrix displays the parts of disjunctive types that are being returned.

Partial functions are expressed in the matrix notation by writing $\mathbb{0}$ in the missing rows:

```
def get[A]: Option[A] => A = {
    case Some(x) => x
} // Partial function; fails on ‘None‘.
```

$$\text{get}^{:\mathbb{1}+A\to A} \stackrel{\text{def}}{=} \left|\begin{array}{c||c} & A \\ \hline \mathbb{1} & \mathbb{0} \\ A & x^{:A} \to x \end{array}\right| = \left|\begin{array}{c||c} & A \\ \hline \mathbb{1} & \mathbb{0} \\ A & \text{id} \end{array}\right| \quad .$$

Scala's `match` expression is equivalent to an application of a disjunctive function:

```
val p: Option[Int] = Some(64)
val q: Option[Int] = p match {
    case None      => Some(100)
    case Some(x)   => Some(x / 2)
}    // The value of q equals Some(32).
```

$$p \stackrel{\text{def}}{=} \mathbb{0}^{:\mathbb{1}} + 64^{:\text{Int}} \quad , \quad q \stackrel{\text{def}}{=} p \triangleright \left|\begin{array}{c||cc} & \mathbb{1} & \text{Int} \\ \hline \mathbb{1} & \mathbb{0} & _ \to 100 \\ \text{Int} & \mathbb{0} & x \to \frac{x}{2} \end{array}\right| \quad .$$

It is convenient to put the argument p to the *left* of the disjunctive function, as in the Scala code.

Because only one part of a disjunctive type can ever be returned, a row can have at most one non-void value. That value will be in the column corresponding to the part being returned.

The matrix notation allows us to compute such function applications directly. We view the disjunctive value $\mathbb{0} + 64^{:\text{Int}}$ as a "row vector" $\left|\begin{array}{cc} \mathbb{0} & 64 \end{array}\right|$, written with a single left line to distinguish it from a function matrix. Calculations use the standard rules of a vector-matrix product:

$$(\mathbb{0}+64) \triangleright \left\|\begin{array}{cc} \mathbb{0} & _ \to 100 \\ \mathbb{0} & x \to \frac{x}{2} \end{array}\right| = \left|\begin{array}{cc} \mathbb{0} & 64 \end{array}\right| \triangleright \left\|\begin{array}{cc} \mathbb{0} & _ \to 100 \\ \mathbb{0} & x \to \frac{x}{2} \end{array}\right| = \left|\begin{array}{cc} \mathbb{0} & 64 \triangleright (x \to \frac{x}{2}) \end{array}\right| = \left|\begin{array}{cc} \mathbb{0} & 32 \end{array}\right| = (\mathbb{0}+32) \quad .$$

The pipe ($\triangleright$) operation plays the role of the "multiplication" of matrix elements, and we drop any terms containing $\mathbb{0}$. We omitted type annotations since we already checked that the types match.

9) Use a recursive call The last construction is to call a function recursively within its own definition. This construction was not shown in Section 5.2.2 because the constructive propositional logic (which was the main focus in that chapter) cannot represent a recursively defined value. However, this limitation of propositional logic means only that we do not have an algorithm for *automatic* derivation of recursive code. Similarly, no algorithm can automatically derive code that involves type constructors with known methods. Nevertheless, those derivations can be performed by hand.

Recursive code is used often, and we need to get some experience reasoning about it. In derivations, this book denotes recursive calls by an *overline*. For example, the standard `fmap` method for the `List` functor is defined as:

$$\text{fmap}_{\text{List}}(f) \stackrel{\text{def}}{=} \left\| \begin{array}{c|cc} & \mathbb{1} & A \times \text{List}^A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A \times \text{List}^A & \mathbb{0} & h^{:A} \times t^{:\text{List}^A} \to f(h) \times \left(t \triangleright \overline{\text{fmap}_{\text{List}}}(f)\right) \end{array} \right| \quad .$$

The recursive call to $\text{fmap}_{\text{List}}$ is applied to a list's tail (the value t).

In proofs of laws for recursive functions, it is necessary to use induction in the number of recursive self-calls. However, the proof does not need to separate the base case (no recursive calls) from the inductive step. In the proof, we write a symbolic calculation as usual, except that we may assume that the law already holds for any recursive calls to the same function.

For example, a proof of the identity law of $\text{fmap}_{\text{List}}$ (which says that $\text{fmap}_{\text{List}}(\text{id}) = \text{id}$) may proceed by replacing the recursive call $\overline{\text{fmap}_{\text{List}}}(\text{id})$ by id during the calculations (see Example 6.1.4.4):

$$\text{fmap}_{\text{List}}(\text{id}) = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h^{:A} \times t^{:\text{List}^A} \to \text{id}(h) \times \left(t \triangleright \underline{\overline{\text{fmap}_{\text{List}}}(\text{id})}\right) \end{array} \right|$$

$$\text{inductive assumption}: \quad = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h \times t \to \underline{\text{id}(h) \times (t \triangleright \text{id})} \end{array} \right| = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \underline{h \times t \to h \times t} \end{array} \right|$$

$$\text{identity matrix}: \quad = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \end{array} \right| = \text{id} \quad .$$

7.1.2 Function composition and the pipe notation

In addition to the basic code constructions, our derivations will often need to work with function compositions and lifted functions. It is often faster to perform calculations with functions when we do not write all of their arguments explicitly; e.g., writing the right identity law as $f \fatsemi \text{id} = f$ instead of $\text{id}(f(x)) = f(x)$. This is known as calculating in **point-free** style (meaning "argument-free"). Many laws can be formulated and used more easily in the point-free form.

Calculations in point-free style almost always involve composing functions. This book prefers to use the *forward* function composition ($f \fatsemi g$) defined for arbitrary $f^{:A \to B}$ and $g^{:B \to C}$ by:

```
f andThen g == { x => g(f(x)) }
```

$$f \fatsemi g \stackrel{\text{def}}{=} x \to g(f(x)) \quad .$$

A useful tool for calculations is the **pipe** operation, $x \triangleright f$, which places the argument (x) to the *left* of a function (f). It is then natural to apply further functions at *right*, for example $(x \triangleright f) \triangleright g$ meaning $g(f(x))$. In Scala, methods such as `map` and `filter` are often combined in this way:

```
x.map(f).filter(p)
```

$$x \triangleright \text{fmap}(f) \triangleright \text{filt}(p) \quad .$$

To enable this common usage, the $\triangleright$ operation is defined to group towards the left. So, the parentheses in $(x \triangleright f) \triangleright g = x \triangleright f \triangleright g$ are not needed.

Since $x \triangleright f \triangleright g = g(f(x))$ by definition, it follows that the composition $f \fatsemi g$ satisfies the equation:

$$x \triangleright f \triangleright g = x \triangleright (f \fatsemi g) \quad .$$

Such formulas are needed often, so we follow the convention that the pipe operation ($\triangleright$) groups weaker than the composition operation ($\fatsemi$). We can then omit parentheses: $x \triangleright (f \fatsemi g) = x \triangleright f \fatsemi g$.

Another common simplification occurs with function compositions of the form:

$$(x \to t \triangleright f) \fatsemi g = x \to g(t \triangleright f) = x \to (t \triangleright f \triangleright g) = x \to t \triangleright f \fatsemi g \quad .$$

The function arrow groups weaker than the pipe operator: $x \to t \triangleright f \fatsemi g = x \to (t \triangleright f \fatsemi g)$.

How can we verify this and other similar computations where the operations $\triangleright$ and $\fatsemi$ are combined in some way? Instead of memorizing a large set of identities, we can rely on knowing only one rule that says how arguments are symbolically substituted as parameters into functions, for example:

$$\text{substitute } x \text{ instead of } a: \quad \underline{x} \triangleright (\underline{a} \to f(\underline{a})) = f(x) \quad .$$
$$\text{substitute } f(x) \text{ instead of } y: \quad (x \to \underline{f(x)}) \fatsemi (\underline{y} \to g(\underline{y})) = x \to g(f(x)) \quad .$$

Whenever there is a doubt (is $x \triangleright (f \triangleright g)$ or $(x \fatsemi f) \triangleright g$ the correct formula?), one can write functions in an expanded form (writing $x \to f(x)$ instead of f) and perform calculations more verbosely. After getting some experience with the $\triangleright$ and $\fatsemi$ operations, the reader will start using them more freely without writing functions in expanded form.

The matrix notation is adapted to the pipe operation and the forward function composition. As an example, let us write the composition of the functions `compute` and `get[Int]` shown above:

$$\text{compute} \fatsemi \text{get} = \begin{array}{c||cc|} & \mathbb{1} & \text{Int} \\ \hline \mathbb{1} & \mathbb{0} & 1 \to 100 \\ \text{Int} & \mathbb{0} & x \to \frac{x}{2} \end{array} \fatsemi \begin{array}{c||c|} & \text{Int} \\ \hline \mathbb{1} & \mathbb{0} \\ \text{Int} & \text{id} \end{array} = \begin{array}{c||c|} & \text{Int} \\ \hline \mathbb{1} & (1 \to 100) \fatsemi \text{id} \\ \text{Int} & (x \to \frac{x}{2}) \fatsemi \text{id} \end{array} = \begin{array}{c||c|} & \text{Int} \\ \hline \mathbb{1} & 1 \to 100 \\ \text{Int} & x \to \frac{x}{2} \end{array} \quad .$$

In this computation, we used the composition ($\fatsemi$) instead of the "multiplication" of matrix elements.

Why does the rule for matrix multiplication work for function compositions? The reason is the equivalence $x \triangleright f \triangleright g = x \triangleright f \fatsemi g$. We have defined the matrix form of functions to work with the "row-vector" form of disjunctive types, i.e., for the computation $x \triangleright f$ (where x is a row vector representing a value of a disjunctive type). The result of computing $x \triangleright f$ is again a row vector, which we can pipe into another matrix g as $x \triangleright f \triangleright g$. The standard rules of matrix multiplication make it associative; so, the result of $x \triangleright f \triangleright g$ is the same as the result of piping x into the matrix product of f and g. Therefore, the matrix product of f and g must yield the function $f \fatsemi g$.

A "non-disjunctive" function (i.e., one not taking or returning disjunctive types) may be written as a 1×1 matrix, so its composition with disjunctive functions can be computed via the same rules.

7.1.3 Functor and contrafunctor liftings

Functions and function compositions lifted to a functor (or to a contrafunctor) are used in derivations so often that the notation $x \triangleright \text{fmap}_F(f)$ and its Scala analog `x.map(f)` are not short enough. This book uses the notation $x \triangleright f^{\uparrow F}$ for functors F and $x \triangleright f^{\downarrow C}$ for contrafunctors C. This notation graphically emphasizes the function f being lifted and also shows the name of the relevant functors or contrafunctors. Compositions of lifted functions are visually easy to recognize, for example:

$$f^{\downarrow H} \fatsemi g^{\downarrow H} = (g \fatsemi f)^{\downarrow H} \quad , \qquad f^{\uparrow L} \fatsemi g^{\uparrow L} \fatsemi h^{\uparrow L} = (f \fatsemi g \fatsemi h)^{\uparrow L} \quad .$$

In these formulas, the labels $^{\downarrow H}$ and $^{\uparrow L}$ clearly suggest that we may pull several functions under a single lifting. We may also split a lifted composition into a composition of liftings.

The lifting notation helps us recognize that these steps are possible just by looking at the formula. Of course, we still need to find a useful sequence of steps in a given derivation or proof.

7.2 Derivation techniques

7.2.1 Standard functions for working with products

The functions denoted by π_1, π_2, Δ, and $\boxtimes$ proved to be helpful in derivations that involve tuples. (However, the last two functions are unlikely to be frequently used in practical programming.)

We already saw the definition and the implementation of the functions π_1 and π_2.

The "diagonal" function Δ is a right inverse for π_1 and π_2:

```
def delta[A]: A => (A, A) = { x => (x, x) }
```

$$\Delta^A : A \to A \times A \quad , \qquad \Delta \stackrel{\text{def}}{=} a^{:A} \to a \times a \quad .$$

It is clear that extracting any part of a pair `delta(x) == (x, x)` will give back the original `x`. This property can be written as an equation or a "law":

```
delta(x)._1 == x
```

$$\pi_1(\Delta(x)) = x \quad .$$

We can transform this law into a point-free equation by first using the pipe notation:

$$\pi_1(\Delta(x)) = (\Delta(x)) \triangleright \pi_1 = x \triangleright \Delta \triangleright \pi_1 = x \triangleright \Delta \fatsemi \pi_1 \quad ,$$

which gives the equation $x \triangleright \Delta \fatsemi \pi_1 = x = x \triangleright \text{id}$. Now we omit "$x \triangleright$" and obtain a point-free equation:

$$\Delta \text{ is a right inverse of } \pi_1 : \quad \Delta \fatsemi \pi_1 = \text{id} \quad . \tag{7.1}$$

The same property holds for π_2.

The **pair product** operation $f \boxtimes g$ is defined for any functions $f^{:A\to P}$ and $g^{:B\to Q}$ by:

```
def pairProduct[A,B,P,Q](f: A => P, g: B => Q): ((A, B)) => (P, Q) = {
    case (a, b) => (f(a), g(b))
}
```

$$f \boxtimes g : A \times B \to P \times Q \quad , \qquad f \boxtimes g \stackrel{\text{def}}{=} a \times b \to f(a) \times g(b) \quad .$$

Two properties of this operation follow directly from its definition:

$$\text{composition law :} \quad (f^{:A\to P} \boxtimes g^{:B\to Q}) \fatsemi (m^{:P\to X} \boxtimes n^{:Q\to Y}) = (f \fatsemi m) \boxtimes (g \fatsemi n) \quad , \tag{7.2}$$

$$\text{left and right projection laws :} \quad (f^{:A\to P} \boxtimes g^{:B\to Q}) \fatsemi \pi_1 = \pi_1 \fatsemi f \quad , \qquad (f \boxtimes g) \fatsemi \pi_2 = \pi_2 \fatsemi g \quad , \tag{7.3}$$

$$\text{identity law :} \quad \text{id}^A \boxtimes \text{id}^B = \text{id}^{A\times B} \quad .$$

An equivalent way of defining $f \boxtimes g$ is via this Scala code:

```
def pairProduct[A,B,P,Q](f: A => P, g: B => Q)(p: (A, B)): (P, Q)  =  (f(p._1), g(p._2))
```

$$f \boxtimes g = p^{:A\times B} \to f(p \triangleright \pi_1) \times g(p \triangleright \pi_2) = p \to (p \triangleright \pi_1 \triangleright f) \times (p \triangleright \pi_2 \triangleright g) \quad .$$

The pair product notation can shorten calculations with functors that involve product types (tuples). For example, the lifting for the functor $F^A \stackrel{\text{def}}{=} A \times A \times Z$ can be shortened to:

$$f^{\uparrow F} \stackrel{\text{def}}{=} \left(a_1^{:A} \times a_2^{:A} \times z^{:Z} \to f(a_1) \times f(a_2) \times z\right) = f \boxtimes f \boxtimes \text{id} \quad .$$

The last formula is often more convenient in symbolic derivations.

7.2.2 Deriving laws for functions with known implementations

The task is to prove a given law (an equation) for a function whose code is known. An example of such an equation is the naturality law of Δ, which states that for any function $f^{:A\to B}$ we have:

$$f \fatsemi \Delta = \Delta \fatsemi (f \boxtimes f) \quad . \tag{7.4}$$

Laws for fully parametric functions are often written without type annotations. However, it is important to check that types match. So we begin by finding suitable type parameters for Eq. (7.4).

Since it is given that f has type $A \to B$, the function Δ in the left-hand side of Eq. (7.4) must take arguments of type B and thus returns a value of type $B \times B$. We see that the left-hand side must be a function of type $A \to B \times B$. So, the Δ in the right-hand side must take arguments of type A. It then returns a value of type $A \times A$, which is consumed by $f \boxtimes f$. In this way, we see that all types match. We can put the resulting types into a type diagram and write the law with type annotations:

$$\begin{tikzcd} A \arrow[r, "\Delta^A"] \arrow[d, "f"'] & A \times A \arrow[d, "f \boxtimes f"] \\ B \arrow[r, "\Delta^B"] & B \times B \end{tikzcd}$$

$$f^{:A\to B} \fatsemi \Delta^{:B\to B\times B} = \Delta^{:A\to A\times A} \fatsemi (f \boxtimes f) \quad .$$

To prove the law, we need to use the known code of the function Δ. We substitute that code into the left-hand side of the law and into the right-hand side of the law, hoping to transform these two expressions until they are the same.

We will now perform this computation in the Scala syntax and in the code notation:

```
x.pipe(f andThen delta)
  == (f(x)).pipe { a => (a, a) }
  == (f(x), f(x)) // Left-hand side.
x.pipe(delta andThen { case (a, b) => (f(a), f(b)) })
  == (x, x).pipe { case (a, b) => (f(a), f(b)) }
  == (f(x), f(x)) // Right-hand side.
```

$$\begin{aligned} x \triangleright f \fatsemi \Delta &= f(x) \triangleright \underline{(b \to b \times b)} \\ &= f(x) \times f(x) \quad . \\ \underline{x \triangleright \Delta} \fatsemi (f \boxtimes f) & \\ &= (x \times x) \triangleright \underline{(a \times b \to f(a) \times f(b))} \\ &= f(x) \times f(x) \quad . \end{aligned}$$

At each step of the derivation, there is often only one symbolic transformation we can perform. In the example above, each step either substitutes a definition of a known function or applies some function to an argument and computes the result. To help us remember what was done, we use a green underline as a hint indicating a sub-expression to be modified in that step.

We will prefer to derive laws in the code notation rather than in Scala syntax. The code notation covers all fully parametric code, i.e., all programs that use only the nine basic code constructions.

7.2.3 Working with disjunctive types in matrix notation

The matrix notation provides a general way of performing symbolic derivations with disjunctive types in point-free style (the matrix elements are *functions*). Writing all code matrices with type annotations makes it easier to translate between matrices and Scala code.

In many cases, the rules of matrix multiplication and function composition are sufficient for calculating with disjunctive types. For example, consider the following functions `swap[A]` and `merge[A]`:

```
def swap[A]: Either[A, A] => Either[A, A] = {
    case Left(a)    => Right(a)
    case Right(a)   => Left(a)
}
def merge[A]: Either[A, A] => A = {
    case Left(a)    => a
    case Right(a)   => a
}
```

$$\text{swap}^A \stackrel{\text{def}}{=} \left\| \begin{array}{c|cc} & A & A \\ \hline A & \mathbb{0} & \text{id} \\ A & \text{id} & \mathbb{0} \end{array} \right\| \quad , \quad \text{merge}^A \stackrel{\text{def}}{=} \left\| \begin{array}{c|c} & A \\ \hline A & \text{id} \\ A & \text{id} \end{array} \right\| \quad .$$

We can quickly prove by matrix composition that $\text{swap} \fatsemi \text{swap} = \text{id}$ and $\text{swap} \fatsemi \text{merge} = \text{merge}$:

$$\text{swap} \fatsemi \text{swap} = \left\| \begin{array}{cc} \mathbb{0} & \text{id} \\ \text{id} & \mathbb{0} \end{array} \right\| \fatsemi \left\| \begin{array}{cc} \mathbb{0} & \text{id} \\ \text{id} & \mathbb{0} \end{array} \right\| = \left\| \begin{array}{cc} \text{id} \fatsemi \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \fatsemi \text{id} \end{array} \right\| = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \end{array} \right\| = \text{id} \quad ,$$

$$\text{swap} \fatsemi \text{merge} = \left\| \begin{array}{cc} \mathbb{0} & \text{id} \\ \text{id} & \mathbb{0} \end{array} \right\| \fatsemi \left\| \begin{array}{c} \text{id} \\ \text{id} \end{array} \right\| = \left\| \begin{array}{c} \text{id} \fatsemi \text{id} \\ \text{id} \fatsemi \text{id} \end{array} \right\| = \left\| \begin{array}{c} \text{id} \\ \text{id} \end{array} \right\| = \text{merge} \quad .$$

The identity function for any disjunctive type, e.g., $A+B+C$, is the "identity diagonal" matrix:

$$\mathrm{id}^{:A+B+C\to A+B+C} = \begin{array}{c||ccc} & A & B & C \\ \hline A & \mathrm{id} & \mathbb{0} & \mathbb{0} \\ B & \mathbb{0} & \mathrm{id} & \mathbb{0} \\ C & \mathbb{0} & \mathbb{0} & \mathrm{id} \end{array} \quad .$$

As another example, consider the function `fmap` for the functor $E^A \stackrel{\text{def}}{=} A+A$:

```
def fmap[A, B](f: A => B): Either[A, A] => Either[B, B] = {
    case Left(a)    => Left(f(a))
    case Right(a)   => Right(f(a))
}
```

$$(f^{:A\to B})^{\uparrow E} \stackrel{\text{def}}{=} \begin{array}{c||cc} & B & B \\ \hline A & f & \mathbb{0} \\ A & \mathbb{0} & f \end{array} \quad .$$

With this definition, we can formulate a law of `merge` called the "naturality law":

$$\begin{tikzcd} A+A \arrow[r, "\mathrm{merge}^A"'] \arrow[d, "f^{\uparrow E}"'] & A \arrow[d, "f"] \\ B+B \arrow[r, "\mathrm{merge}^B"'] & B \end{tikzcd}$$

$$(f^{:A\to B})^{\uparrow E} ⨾ \mathrm{merge}^B = \mathrm{merge}^A ⨾ f^{:A\to B} \quad .$$

Proving this law is a short matrix calculation:

$$\text{left-hand side}: \quad f^{\uparrow E} ⨾ \mathrm{merge} = \begin{Vmatrix} f & \mathbb{0} \\ \mathbb{0} & f \end{Vmatrix} ⨾ \begin{Vmatrix} \mathrm{id} \\ \mathrm{id} \end{Vmatrix} = \begin{Vmatrix} f ⨾ \mathrm{id} \\ f ⨾ \mathrm{id} \end{Vmatrix} = \begin{Vmatrix} f \\ f \end{Vmatrix} \quad ,$$

$$\text{right-hand side}: \quad \mathrm{merge} ⨾ f = \begin{Vmatrix} \mathrm{id} \\ \mathrm{id} \end{Vmatrix} ⨾ \underline{f} = \begin{Vmatrix} \mathrm{id} \\ \mathrm{id} \end{Vmatrix} ⨾ \begin{Vmatrix} f \end{Vmatrix} = \begin{Vmatrix} \mathrm{id} ⨾ f \\ \mathrm{id} ⨾ f \end{Vmatrix} = \begin{Vmatrix} f \\ f \end{Vmatrix} \quad .$$

In the last line we replaced f by a 1×1 matrix, $\begin{Vmatrix} f \end{Vmatrix}$, in order to apply matrix composition.

Matrix rows and columns can be split or merged when necessary to accommodate various disjunctive types. As an example, let us verify the **associativity law** of `merge`:

$$\begin{tikzcd} E^{A+A} \arrow[r, "\mathrm{merge}^{A+A}"] \arrow[d, "\mathrm{merge}^{\uparrow E}"] & A+A \arrow[d, "\mathrm{merge}^A"'] \\ E^A \arrow[r, "\mathrm{merge}^A"'] & A \end{tikzcd}$$

$$(\mathrm{merge}^A)^{\uparrow E} ⨾ \mathrm{merge}^A = \mathrm{merge}^{A+A} ⨾ \mathrm{merge}^A \quad .$$

Both sides of this law are functions of type $A+A+A+A \to A$. To transform the left-hand side, we use the definition of $^{\uparrow E}$ and write:

$$\mathrm{merge}^{\uparrow E} ⨾ \mathrm{merge} = \begin{array}{c||cc} & A & A \\ \hline A+A & \mathrm{merge} & \mathbb{0} \\ A+A & \mathbb{0} & \mathrm{merge} \end{array} ⨾ \begin{array}{c||c} & A \\ \hline A & \mathrm{id} \\ A & \mathrm{id} \end{array} = \begin{array}{c||c} & A \\ \hline A+A & \mathrm{merge} \\ A+A & \mathrm{merge} \end{array} \quad .$$

However, we have not yet substituted the definition of `merge` into the matrix. To do that, add more rows to the matrix in order to accommodate the disjunctive type $(A+A)+(A+A)$:

$$\mathrm{merge}^{\uparrow E} ⨾ \mathrm{merge} = \begin{array}{c||c} & A \\ \hline A+A & \mathrm{merge} \\ A+A & \mathrm{merge} \end{array} = \begin{array}{c||c} & A \\ \hline A & \mathrm{id} \\ A & \mathrm{id} \\ A & \mathrm{id} \\ A & \mathrm{id} \end{array} \quad .$$

Now we compute the right-hand side of the law by substituting the code of `merge`:

$$\mathrm{merge}^{A+A} ⨾ \mathrm{merge}^A = \begin{array}{c||c} & A+A \\ \hline A+A & \mathrm{id} \\ A+A & \mathrm{id} \end{array} ⨾ \begin{array}{c||c} & A \\ \hline A & \mathrm{id} \\ A & \mathrm{id} \end{array} \quad .$$

We cannot proceed with matrix composition because the dimensions of the matrices do not match. To compute further, we need to expand the rows and the columns of the first matrix:

$$\begin{array}{c||c|}
 & A+A \\ \hline\hline
A+A & \mathrm{id} \\
A+A & \mathrm{id}
\end{array}
\;\fatsemi\;
\begin{array}{c||c|}
 & A \\ \hline\hline
A & \mathrm{id} \\
A & \mathrm{id}
\end{array}
=
\begin{array}{c||cc|}
 & A & A \\ \hline\hline
A & \mathrm{id} & \mathbb{0} \\
A & \mathbb{0} & \mathrm{id} \\
A & \mathrm{id} & \mathbb{0} \\
A & \mathbb{0} & \mathrm{id}
\end{array}
\;\fatsemi\;
\begin{array}{c||c|}
 & A \\ \hline\hline
A & \mathrm{id} \\
A & \mathrm{id}
\end{array}
=
\begin{array}{c||c|}
 & A \\ \hline\hline
A & \mathrm{id} \\
A & \mathrm{id} \\
A & \mathrm{id} \\
A & \mathrm{id}
\end{array}
\quad .$$

This proves the law and also helps visualize how various types are transformed by `merge`.

In some cases, we cannot fully split the rows or the columns of a matrix. For instance, if we are calculating with an arbitrary function $f^{:\mathbb{1}+A\to\mathbb{1}+B}$, we cannot write that function in a form of a 2×2 matrix because we do not know which parts of the disjunction are returned (the code of the function f is arbitrary and unknown). At most, we could split the *rows* by expressing the function f through two unknown functions $g^{:\mathbb{1}\to\mathbb{1}+B}$ and $h^{:A\to\mathbb{1}+B}$:

$$f = \begin{array}{c||c|}
 & \mathbb{1}+B \\ \hline\hline
\mathbb{1} & g \\
A & h
\end{array}
\quad .$$

The single column of this matrix remains unsplit. Either that column will remain unsplit throughout the derivation, or additional information about f, g, or h will allow us to split the column.

Finally, there are two tricks that complement the matrix intuition and may sometimes simplify a disjunctive function.[1]

Ignored arguments If all rows of the disjunctive function ignore their arguments and always return the same results, we may collapse all rows into one, as shown in this example:

```
def same[A]: Either[A, Option[A]] => Option[A] = {
    case Left(a)          => None
    case Right(None)      => None
    case Right(Some(a))   => None
}
```

$$\mathrm{same}^{:A+\mathbb{1}+A\to\mathbb{1}+A} \stackrel{\mathrm{def}}{=} \begin{array}{c||cc|}
 & \mathbb{1} & A \\ \hline\hline
A & _\to 1 & \mathbb{0} \\
\mathbb{1} & _\to 1 & \mathbb{0} \\
A & _\to 1 & \mathbb{0}
\end{array}
= \begin{array}{c||cc|}
 & \mathbb{1} & A \\ \hline\hline
A+\mathbb{1}+A & _\to 1 & \mathbb{0}
\end{array}
\quad .$$

There is a more general formula for arbitrary functions $f^{:X\to C}$ containing this code:

$$\text{the value } p \text{ is ignored}: \quad x^{:X} \to p^{:A+B} \triangleright \begin{array}{c||c|}
 & C \\ \hline\hline
A & _\to f(x) \\
B & _\to f(x)
\end{array} = x^{:X} \to f(x) = f \quad .$$

In this case, we can completely collapse the matrix, getting an ordinary (non-disjunctive) function.

Simplification of diagonal pair products Consider the pair product of two disjunctive functions such as $f^{:A+B\to R}$ and $g^{:P+Q\to S}$. Computing $f\boxtimes g$ in the matrix notation requires, in general, to split the rows and the columns of the matrices because the type of $f\boxtimes g$ is:

$$\begin{aligned}
f\boxtimes g &: (A+B)\times(P+Q)\to R\times S \\
&\cong A\times P + A\times Q + B\times P + B\times Q \to R\times S \quad .
\end{aligned}$$

[1] These tricks are adapted from Section 2.8 of the book "Program design by calculation" (draft from October 2019), see http://www4.di.uminho.pt/~jno/ps/pdbc.pdf for the latest version.

So, the pair product of two 2×1 matrices must be written *in general* as a 4×1 matrix:

$$\text{for any } f \stackrel{\text{def}}{=} \begin{array}{|c||c|} & R \\ \hline A & f_1 \\ B & f_2 \end{array} \quad \text{and} \quad g \stackrel{\text{def}}{=} \begin{array}{|c||c|} & S \\ \hline P & g_1 \\ Q & g_2 \end{array} \quad , \quad \text{we have} \quad f \boxtimes g = \begin{array}{|c||c|} & R \times S \\ \hline A \times P & f_1 \boxtimes g_1 \\ A \times Q & f_1 \boxtimes g_2 \\ B \times P & f_2 \boxtimes g_1 \\ B \times Q & f_2 \boxtimes g_2 \end{array} \quad .$$

A simplification trick exists when the pair product is composed with the diagonal function Δ:

$$\Delta \fatsemi (f \boxtimes g) = \Delta^{:A+B \to (A+B) \times (A+B)} \fatsemi (f^{:A+B \to R} \boxtimes g^{:A+B \to S}) = p \to f(p) \times g(p) \quad .$$

This "diagonal pair product" is well-typed only if f and g have the same argument types. It turns out that the function $\Delta \fatsemi (f \boxtimes g)$ can be written as a 2×1 matrix, i.e., we do not need to split the rows:

$$\text{for any } f \stackrel{\text{def}}{=} \begin{array}{|c||c|} & R \\ \hline A & f_1 \\ B & f_2 \end{array} \quad \text{and} \quad g \stackrel{\text{def}}{=} \begin{array}{|c||c|} & S \\ \hline A & g_1 \\ B & g_2 \end{array} \quad , \quad \text{we have} \quad \Delta \fatsemi (f \boxtimes g) = \begin{array}{|c||c|} & R \times S \\ \hline A & \Delta \fatsemi (f_1 \boxtimes g_1) \\ B & \Delta \fatsemi (f_2 \boxtimes g_2) \end{array} \quad .$$

The rules of matrix multiplication do not help in deriving this law. So, we use a more basic approach: show that both sides are equal when applied to arbitrary values p of type $A + B$:

$$p^{:A+B} \triangleright \Delta \fatsemi (f \boxtimes g) = f(p) \times g(p) \stackrel{?}{=} p \triangleright \begin{array}{|c||c|} & R \times S \\ \hline A & \Delta \fatsemi (f_1 \boxtimes g_1) \\ B & \Delta \fatsemi (f_2 \boxtimes g_2) \end{array} \quad .$$

The type $A + B$ has two cases. Applying the left-hand side to $p \stackrel{\text{def}}{=} a^{:A} + \mathbb{0}^{:B}$, we get:

$$\begin{aligned} f(p) \times g(p) &= \big((a^{:A} + \mathbb{0}^{:B}) \triangleright f \big) \times \big((a^{:A} + \mathbb{0}^{:B}) \triangleright g \big) \\ &= \left(\begin{array}{|cc|} a & \mathbb{0} \end{array} \triangleright \begin{array}{||c|} f_1 \\ f_2 \end{array} \right) \times \left(\begin{array}{|cc|} a & \mathbb{0} \end{array} \triangleright \begin{array}{||c|} g_1 \\ g_2 \end{array} \right) = (a \triangleright f_1) \times (a \triangleright g_1) = f_1(a) \times g_1(a) \quad . \end{aligned}$$

Applying the right-hand side to the same p, we find:

$$\begin{aligned} \text{expect to equal } f_1(a) \times g_1(a) : \quad & \underline{p} \triangleright \begin{array}{||c|} \Delta \fatsemi (f_1 \boxtimes g_1) \\ \Delta \fatsemi (f_2 \boxtimes g_2) \end{array} = \begin{array}{|cc|} a & \mathbb{0} \end{array} \triangleright \begin{array}{||c|} \Delta \fatsemi (f_1 \boxtimes g_1) \\ \Delta \fatsemi (f_2 \boxtimes g_2) \end{array} = \underline{a \triangleright \Delta} \fatsemi (f_1 \boxtimes g_1) \\ \text{definition of } \Delta : \quad & = (a \times a) \triangleright (f_1 \boxtimes g_1) = f_1(a) \times g_1(a) \quad . \end{aligned}$$

A similar calculation with $p \stackrel{\text{def}}{=} \mathbb{0}^{:A} + b^{:B}$ shows that both sides of the law are equal to $f_2(b) \times g_2(b)$.

7.2.4 Derivations involving unknown functions with laws

A more challenging task is to derive an equation that uses arbitrary functions about which we only know that they satisfy certain given laws. Such derivations usually proceed by trying to transform the code until the given laws can be applied.

As an example, let us derive the property that $L^A \stackrel{\text{def}}{=} A \times F^A$ is a functor if $F^\bullet$ is known to be a functor. We are in the situation where we only know that the function fmap_F exists and satisfies the functor law, but we do not know the code of fmap_F. Let us discover the derivation step by step.

First, we need to define fmap_L. We use the lifting notation $^{\uparrow F}$ and write, for any $f^{:A \to B}$:

```
def fmap_L[A, B](f: A => B): ((A, F[A])) => (B, F[B]) = { case (a, p) => (f(a), p.map(f)) }
```

$$f^{\uparrow L} \stackrel{\text{def}}{=} a^{:A} \times p^{:F^A} \to f(a) \times (p \triangleright f^{\uparrow F}) \quad .$$

To verify the identity law of L:

$$\text{expect to equal id}: \quad \text{id}^{\uparrow L} = a^{:A} \times p^{:F^A} \to \text{id}\,(a) \times (p \triangleright \text{id}^{\uparrow F}) = ???$$

At this point, the only things we can simplify are the identity functions applied to arguments. We know that F is a lawful functor; therefore, $\text{id}^{\uparrow F} = \text{id}$. So we continue the derivation, omitting types:

$$\begin{aligned}
\text{expect to equal id}: \quad & \text{id}^{\uparrow L} = a \times p \to \underline{\text{id}\,(a)} \times (p \triangleright \underline{\text{id}^{\uparrow F}}) \\
\text{identity law of } F: \quad & = a \times p \to a \times (\underline{p \triangleright \text{id}}) \\
\text{apply function}: \quad & = a \times p \to a \times p = \text{id} \quad .
\end{aligned}$$

To verify the composition law of L, we assume two arbitrary functions $f^{:A\to B}$ and $g^{:B\to C}$:

$$\text{expect to equal } (f \fatsemi g)^{\uparrow L}: \quad f^{\uparrow L} \fatsemi g^{\uparrow L} = \big(a \times p \to f(a) \times f^{\uparrow F}(p)\big) \fatsemi \big(b \times q \to g(b) \times g^{\uparrow F}(q)\big) \quad .$$

At this point, we pause and try to see how we might proceed. We do not know anything about f and g, so we cannot evaluate $f(a)$ or $f^{\uparrow F}(p)$. We also do not have the code of $^{\uparrow F}$ (i.e., of fmap_F). The only information we have about these functions is that F's composition law holds:

$$f^{\uparrow F} \fatsemi g^{\uparrow F} = (f \fatsemi g)^{\uparrow F} \quad . \tag{7.5}$$

We could use this law only if we somehow bring $f^{\uparrow F}$ and $g^{\uparrow F}$ together in the formula. The only way forward is to compute the function composition of the two functions whose code we *do* have:

$$\begin{aligned}
& \big(a \times p \to f(a) \times f^{\uparrow F}(p)\big) \fatsemi \big(b \times q \to g(b) \times g^{\uparrow F}(q)\big) \\
& = a \times p \to g(f(a)) \times g^{\uparrow F}(f^{\uparrow F}(p)) \quad .
\end{aligned}$$

In order to use the law (7.5), we need to rewrite this code via the composition $f \fatsemi g$. We notice that the formula contains exactly those function compositions:

$$g(f(a)) \times g^{\uparrow F}(f^{\uparrow F}(p)) = (a \triangleright f \fatsemi g) \times (p \triangleright f^{\uparrow F} \fatsemi g^{\uparrow F}) \quad .$$

So, we can now apply the composition law of F and write up the complete derivation, adding hints:

$$\begin{aligned}
\text{expect to equal } (f \fatsemi g)^{\uparrow L}: \quad & f^{\uparrow L} \fatsemi g^{\uparrow L} = \big(a \times p \to f(a) \times f^{\uparrow F}(p)\big) \fatsemi \big(b \times q \to g(b) \times g^{\uparrow F}(q)\big) \\
\text{compute composition}: \quad & = a \times p \to \underline{g(f(a))} \times \underline{g^{\uparrow F}(f^{\uparrow F}(p))} \\
\triangleright\text{-notation}: \quad & = a \times p \to (a \triangleright f \fatsemi g) \times \big(p \triangleright \underline{f^{\uparrow F} \fatsemi g^{\uparrow F}}\big) \\
\text{composition law of } F: \quad & = a \times p \to (a \triangleright \underline{f \fatsemi g}) \times \big(p \triangleright (\underline{f \fatsemi g})^{\uparrow F}\big) \\
\text{definition of } ^{\uparrow L}: \quad & = (f \fatsemi g)^{\uparrow L} \quad .
\end{aligned}$$

The derivation becomes significantly shorter if we use the pair product ($\boxtimes$) to define $^{\uparrow L}$:

$$f^{\uparrow L} \stackrel{\text{def}}{=} \text{id} \boxtimes f^{\uparrow F} \quad .$$

For instance, verifying the identity law then looks like this:

$$\text{id}^{\uparrow L} = \text{id} \boxtimes \text{id}^{\uparrow F} = \text{id} \boxtimes \text{id} = \text{id} \quad .$$

This technique was used in the proof of Statement 6.2.3.3. The cost of having a shorter proof is the need to remember the properties of the pair product ($\boxtimes$), which is not often used in derivations.

7.2.5 Exercises

Exercise 7.2.5.1 Assume given functors F, G, K, L and a natural transformation $\phi : F^A \to G^A$.

(a) Prove that $\phi^{\uparrow K} : K^{F^A} \to K^{G^A}$ is also a natural transformation.

(b) Given another natural transformation $\psi : K^A \to L^A$, prove that the pair product of ϕ and ψ, that is, $\phi \boxtimes \psi : F^A \times K^A \to G^A \times L^A$, as well as the pair co-product $\phi \boxplus \psi : F^A + K^A \to G^A + L^A$, are also natural transformations. The **pair co-product** of two functions ϕ and ψ is defined by:

$$(\phi \boxplus \psi) : F^A + K^A \to G^A + L^A \quad , \qquad \phi \boxplus \psi \stackrel{\text{def}}{=} \begin{array}{c|cc} & G^A & L^A \\ \hline F^A & \phi & \mathbb{0} \\ K^A & \mathbb{0} & \psi \end{array} \quad .$$

Exercise 7.2.5.2 Show using matrix calculations that $\text{swap} ⨾ \text{swap} = \text{id}$, where `swap` is the function defined in Section 7.2.3.

Exercise 7.2.5.3 Now consider a different function `swap[A, B]` defined as:

```
def swap[A, B]: ((A, B)) => (B, A) = { case (a, b) => (b, a) }
```

$$\text{swap}^{A,B} \stackrel{\text{def}}{=} a^{:A} \times b^{:B} \to b \times a \quad .$$

Show that $\Delta ⨾ \text{swap} = \Delta$. Write out all types in this law and draw a type diagram.

Exercise 7.2.5.4 Given an arbitrary functor F, define the functor $L^A \stackrel{\text{def}}{=} F^A \times F^A$ and prove, for an arbitrary function $f^{:A\to B}$, the "lifted naturality" law:

$$f^{\uparrow F} ⨾ \Delta = \Delta ⨾ f^{\uparrow L} \quad .$$

Write out all types in this law and draw a type diagram.

Exercise 7.2.5.5 Show that the types $(\mathbb{1} + \mathbb{1}) \times A$ and $A + A$ are equivalent. One direction of this equivalence is given by a function `two[A]` with the type signature:

```
def two[A]: ((Either[Unit, Unit], A)) => Either[A, A] = ???
```

$$\text{two}^A : (\mathbb{1} + \mathbb{1}) \times A \to A + A \quad .$$

Implement that function and prove that it satisfies the "naturality law": for any $f^{:A\to B}$,

$$(\text{id} \boxtimes f) ⨾ \text{two} = \text{two} ⨾ f^{\uparrow E} \quad ,$$

where $E^A \stackrel{\text{def}}{=} A + A$ is the functor whose lifting $^{\uparrow E}$ was defined in Section 7.2.3. Write out the types in this law and draw a type diagram.

Exercise 7.2.5.6 Prove that the following laws hold for arbitrary $f^{:A\to B}$ and $g^{:C\to D}$:

$$\text{left projection law}: \quad (f \boxtimes g) ⨾ \pi_1 = \pi_1 ⨾ f \quad ,$$
$$\text{right projection law}: \quad (f \boxtimes g) ⨾ \pi_2 = \pi_2 ⨾ g \quad .$$

Exercise 7.2.5.7 Given arbitrary functors F and G, define the functor $L^A \stackrel{\text{def}}{=} F^A \times G^A$ and prove that for arbitrary $f^{:A\to B}$,

$$f^{\uparrow L} ⨾ \pi_1 = \pi_1 ⨾ f^{\uparrow F} \quad .$$

Write out the types in this naturality law and draw a type diagram.

Exercise 7.2.5.8 Consider the functor L^A defined as:

$$L^A \stackrel{\text{def}}{=} \text{Int} \times \text{Int} + A \quad .$$

Implement the functions `fmap` and `flatten` (denoted ftn_L) and write their code in matrix notation:

$$(f^{:A\to B})^{\uparrow L} : \text{Int} \times \text{Int} + A \to \text{Int} \times \text{Int} + B \quad ,$$
$$\text{ftn}_L : \text{Int} \times \text{Int} + \text{Int} \times \text{Int} + A \to \text{Int} \times \text{Int} + A \quad .$$

Exercise 7.2.5.9* Show that `flatten` (denoted ftn_L) from Exercise 7.2.5.8 satisfies the naturality law: for any $f^{:A\to B}$, we have $f^{\uparrow L \uparrow L} ⨾ \text{ftn}_L = \text{ftn}_L ⨾ f^{\uparrow L}$.

8 Typeclasses and functions of types

8.1 Motivation and first examples

8.1.1 Constraining type parameters

The summation method, `sum`, works for any collection of numeric values:

```
scala> Seq(1, 2, 3).sum
res0: Int = 6

scala> Seq(1.0, 2.0, 3.0).sum
res1: Double = 6.0
```

We can use `sum` to compute the average of a sequence of numbers:

```
def avg(s: Seq[Double]): Double = s.sum / s.length
```

Can we generalize the averaging function `avg` from `Double` to other numeric types, e.g.:

```
def avg[T](s: Seq[T]): T = ???
```

This code is impossible because averaging works only for certain types `T`, not for arbitrary types `T` as implied by the type signature above. We will be able to define `avg[T]` only if we constrain the type parameter `T` to be a type representing a suitable numeric value (e.g., `Float`, `Double`, or `BigDecimal`).

Another example of a similar situation is a function with type signature $A \times F^B \to F^{A \times B}$:

```
def inject[F[_], A, B](a: A, f: F[B]): F[(A, B)] = f.map(b => (a, b))     // Must have ‘f.map‘.
```

This function requires the type constructor `F[_]` to have a `map` method, i.e., to be a functor. We can implement `inject` only if we constrain the parameter `F` to be a functor.

What would that constraint be like? Consider an ordinary function with no type parameters, e.g.:

```
def f(x: Int): Int = x + 1
```

In this code, the syntax `x: Int` constrains the value of the argument `x` to be integer. It is a type error to apply `f` to a non-integer argument.

Using a similar syntax for *type parameters*, we write the type signatures for `avg` and `inject` as:

```
def avg[T: Fractional](s: Seq[T]): T
def inject[F[_]: Functor, A, B](a: A, f: F[B]): F[(A, B)]
```

Scala uses the syntax `[T: Fractional]` to constrain the type parameter `T` to "fractional numeric" types. Similarly, `[F[_]: Functor]` requires the type constructor `F[_]` to be a functor. Applying `avg` or `inject` to types that do not obey those constraints will be a type error detected at compile time.

In these examples, we are restricting a type parameter to a subset of possible types, because only types from that subset have certain properties that we need. A subset of types, together with the required properties that those types must satisfy, is called a **typeclass**. The syntax `[T: Fractional]` is a **typeclass constraint** that forces the type `T` to belong to the typeclass `Fractional`.

This chapter focuses on defining and using typeclasses and on understanding their properties. We will see in detail how the syntax such as `[T: Fractional]` is implemented and used.

8.1.2 Functions of types and values

The similarity between the type parameter `T` and the argument `s` is clear in this type signature,

```
def avg[T: Fractional](s: Seq[T]): T
```

We can view `avg` as a function that takes *two* parameters (a type `T` and a value `s`) and returns a value. We can also view `avg` as a function from a *type* `T` to a *value* of type `Seq[T] => T`. We may call functions of this kind **type-to-value** functions (TVF). The syntax for TVFs supported in a future version of Scala 3 will show this more clearly:

```
val avg: [T] => Seq[T] => T = ...   // Scala 3 only.
```

To emphasize that `avg` is a TVF, we may write the type signature of `avg` as:

```
def avg[T: Fractional]: Seq[T] => T // Inconvenient in practice! Use avg[T: Fractional](s: Seq[T]): T
```

A type constructor such as `Seq[_]` can be viewed as a **type-to-type** function (TTF) because it can take any type `T` and return a new type `Seq[T]`.

Functions can map from values or from types and to values or to types, as this table shows:

functions...	from value	from type
to value	(VVF) `def f(x:Int):Int`	(TVF) `def pure[A]: A => List[A]`
to type	(VTF) dependent type	(TTF) `type MyData[A] = Either[Int, A]`

We have already seen examples of VVFs, TVFs, and TTFs. Value-to-type functions (VTFs) are known as **dependent types** (or, more verbosely, as "value-dependent types"). An example in Scala:

```
val x = new { type T = Int }
val y: x.T = 123
```

In this example, `x.T` is a dependent type because it is a type that depends on the *value* `x`. For the value `x` defined in this code, the expression `x.T` evaluates to the type `Int`.

We will not consider dependent types (VTFs) in this chapter because typeclasses only require a combination of a TTF and a TVF.

8.1.3 Partial functions of types and values

We would like to define the function `avg[T: Fractional]` as a TVF that can be applied only to a subset of possible types `T`. This is similar to a **partial function**, i.e., a function defined only for a subset of possible values of its argument's type. We may call such functions partial type-to-value functions (PTVFs), to distinguish them from partial value-to-value functions (PVVFs) we saw before.

In some situations, partial functions are safe to use. For instance, the following partial function `p`,

```
def p: Either[Int, String] => Int = { case Left(x) => x - 1 }
```

can be applied only to values of the form `Left(...)`. Applying `p` to a value `Right(...)` will cause a run-time error. However, consider this code:

```
val x = Seq(Left(1), Right("abc"), Left(2))

scala> x.filter(_.isLeft).map(p)
res0: Seq[Int] = List(0, 1)
```

Although `x.filter(_.isLeft)` has type `Seq[Either[Int, String]]`, all values in that sequence are guaranteed to be of type `Left`. So we know it is safe to apply the partial function `p` in `.map(p)`.

Although safe, this code is brittle: if the `filter` operation were moved to another place, we might by mistake write code equivalent to `x.map(p)`, causing a run-time exception. It is better to refactor the code so that the compile-time type-checking guarantees the safety of all operations at run time. For the example shown above, the `collect` method would make a partial function, such as `p`, safe to use:

```
scala> x.collect { case Left(y) => y - 1 }
res1: Seq[Int] = List(0, 1)
```

The `collect` method is a **total** function because it is defined for all values of its arguments and does not throw exceptions.

Total functions are safer to use than partial functions. The partial function `p` can be converted into a total function by changing its type to `Left[Int, String] => Int`. Another example: applying `head` to a `List` is unsafe, but the non-empty list type guarantees at compile time that the first element exists:

```
val xs: NonEmptyList[Int] = ...
val h = xs.head  // .head is a total function for a NonEmptyList.
```

In these cases, we achieve safety by making types more strictly constrained. Similarly, partial type-to-value functions (PTVFs) become safe to use if we impose suitable typeclass constraints on the type parameters. Typeclasses can be viewed as a systematic way of safely managing PTVFs.

8.2 Implementing typeclasses

A typeclass constraint `[T: Fractional]` will generate a compile-time error when a function such as `avg[T]` is applied to an incorrectly chosen type parameter `T`. If the Scala library did not already implement the `Fractional` typeclass, how could we reproduce that functionality?

8.2.1 Creating a partial function at type level

The code needs to specify that the type parameter must belong to a certain subset of allowed types. To simplify the task, assume that the allowed types are `BigDecimal` and `Double`. One example of a type constraint is shown by unfunctors (see Section 6.1.6), which are type constructors whose type parameters are restricted to specific types. As an example of an unfunctor, consider the code shown at left, where values of type `Frac[A]` can be created only if `A = BigDecimal` or `A = Double`. The keywords `sealed` and `final` guarantee that no further code could extend this definition and allow us to create a value of type, say, `Frac[String]` or `Frac[Boolean]`. The Scala compiler will not detect any errors in the following code:

```
sealed trait Frac[A]                // Unfunctor.
final case class FracBD() extends Frac[BigDecimal]
final case class FracD()  extends Frac[Double]
```

```
type T = Frac[String]
type U = Frac[Boolean]
```

But we will never be able to create and use any values of types `T` or `U`. In other words, the types `Frac[String]` and `Frac[Boolean]` are *void* types. Trying to create and use values of these types will result in type errors, as the following code shows:

```
1 def f[A]: Frac[A] = FracD()     // Type error.
2 val x: U = FracD()              // Type error.
3 val y: U = FracD().asInstanceOf[U]
4 y match { case FracD() => }     // Type error.
```

In line 3, we disabled the type checker and forced the Scala compiler to ignore the type error in the definition of `y`. However, line 4 shows that we are unable to use that value `y` in further computations.

The type `Frac[A]` is non-void (i.e., has values) only for A belonging to the set {`BigDecimal`, `Double`} of types. This set is called the **type domain** of the type function `Frac`. We now need to define the function `avg[T]` with a type parameter T constrained to that type domain.

The type constraint $T \in$ {`BigDecimal`, `Double`} is equivalent to the requirement that a value of type Frac^T should exist. So, we will implement the type constraint if we include an *additional argument* of type Frac^T into the type signature of `avg`:

```
def avg[T](s: Seq[T], frac: Frac[T]): T
```

The value `frac:Frac[T]` is called a **typeclass instance** value. Because that value needs to be passed to every call of `avg[T]`, we will be unable to use types `T` for which `Frac[T]` is void (i.e., has no values).

In this way, we implemented the typeclass constraint for the PTVF `avg[T]`. The main steps were:

1. Define a type constructor `Frac[_]`.
2. Make sure values of type `Frac[A]` exist only when `A = BigDecimal` or `A = Double`.
3. Pass a value of type `Frac[T]` to the function `avg[T]` as an additional argument.

It is not necessary to define the type constructor `Frac` via an unfunctor. The type constructor `Frac` is only needed to define the type domain {`BigDecimal`, `Double`}. We can use a simple case class instead:

```
final case class Frac[T]()
val fracBD: Frac[BigDecimal] = Frac()
val fracD: Frac[Double] = Frac()
```

This code creates a type constructor `Frac` and makes values of type `Frac[T]` available for chosen type parameters `T`. In this way, we implemented the required type domain.

To write the code for `avg[T]`, we need to be able to add numeric values and to divide by an integer value. More precisely, the body of `avg[T]` needs access to two PTVFs that we may call `add` and `intdiv`:

```
def add[T](x: T, y: T): T
def intdiv[T](x: T, n: Int): T
```

Since `avg[T]` now has an additional argument `frac`, we may use that argument to hold the required functions. So, we redefine `Frac` as a named tuple (case class) containing the functions `add` and `intdiv`:

```
final case class Frac[T](add: (T, T) => T, intdiv: (T, Int) => T)
```

Typeclass instances for `BigDecimal` and `Double` are then created by the following code:

```
val fracBD = Frac[BigDecimal]( (x, y) => x + y,  (x, n) => x / n )
val fracD = Frac[Double]( (x, y) => x + y,  (x, n) => x / n )
```

With these definitions, implementing `avg[T]` becomes straightforward:

```
def avg[T](s: Seq[T], frac: Frac[T]): T = {      // Assuming ‘s‘ is a non-empty sequence.
  val sum = s.reduce(frac.add)     // Here, ‘reduce‘ would fail on an empty sequence ‘s‘.
  frac.intdiv(sum, s.length)       // Compute ‘sum/length‘.
}
```

To use this function, we need to pass a typeclass instance corresponding to the type `T`:

```
scala> avg(Seq(1.0, 2.0, 3.0), fracD) // It will be a type error to use fracBD instead of fracD here.
res0: Double = 2.0

scala> avg(Seq(BigDecimal(1.0), BigDecimal(2.0)), fracBD)
res1: BigDecimal = 1.5
```

This is a fully working implementation of the `avg` function with a `Frac` typeclass constraint. We have achieved compile-time safety since `avg[T]` cannot be applied to values of unsupported types `T`. We have also achieved easy extensibility: To implement another function as a PTVF with the same type domain, we need to add an extra argument of type `Frac[T]` to the function. To add another supported type `T` to the type domain, we just write one more line of code similar to `val fracD = ...`

An equivalent implementation of the `Frac` typeclass via a `trait` with methods requires this code:

```
trait Frac[T] {                  // Trait is not ‘sealed‘.
  def add(x: T, y: T): T
  def intdiv(x: T, n: Int): T
}
val fracBD = new Frac[BigDecimal] {
  def add(x: BigDecimal, y: BigDecimal): BigDecimal = x + y
  def intdiv(x: BigDecimal, n: Int): BigDecimal = x / n
}
val fracD = new Frac[Double] {
  def add(x: Double, y: Double): Double = x + y
  def intdiv(x: Double, n: Int): Double = x / n
}
```

The function `avg[T]` will work unchanged with this implementation of the `Frac` typeclass.

The implementation via a `trait` is significantly longer than the code using a case class as shown previously. One advantage of the longer code is the ability to combine different typeclasses by `trait` mixing. We will look at that in more detail below. For now, we note that both implementations will require the programmer to add a significant amount of new code:

- Calls to `func[T](args)` need to be changed to `func[T](args, ti)` with typeclass instances `ti`.
- For each supported type `T`, a corresponding typeclass instance value needs to be created.
- All those values need to be passed to all places in the code where PTVFs are used.

The extra work can be reduced (and sometimes avoided) by using Scala's "implicit value" feature.

8.2.2 Scala's `implicit` values

An **implicit** declaration is a feature of Scala that makes values automatically available to any function that declares an "implicit argument" of the same type. Scala's syntax for implicit values is:

```
implicit val x: Int = 123
```

This declaration introduces an implicit value of type `Int` into the current scope. That value will be automatically passed as an argument to any function declaring an argument of type `Int` as `implicit`:

```
def f(a: String)(implicit n: Int) = s"$a with $n"

scala> f("xyz")
res0: String = xyz with 123
```

We need to declare the arguments as `implicit` in the function's type signature, and the implicit arguments must be in a *separate* argument list.

The simplest useful function with an implicit argument is the identity function. In the Scala library, this function is called `implicitly`. Compare its code with the code of the ordinary identity function:

```
def implicitly[T](implicit t: T): T = t
def identity[T](t: T): T = t
```

What does `implicitly[T]` do? Since its only argument is declared as `implicit`, we can simply write `implicitly[T]` with no arguments to apply that function. (The type parameter usually needs to be specified.) If no implicit value of type `T` is available, a compile-time error will occur. If an implicit value of type `T` is available in the current scope, `implicitly[T]` will return that value:

```
implicit val s: String = "qqq"

scala> implicitly[String]
res1: String = qqq
```

It is an error to declare more than one implicit value of the same type in the same scope, because implicit arguments are specified by type alone. The Scala compiler will not be able to set implicit arguments of functions automatically when the function's outer scope contains more than one implicit value of a required type, as in this code:

```
implicit val x: Int = 1
implicit val y: Int = 2

scala> implicitly[Int]
<console>:14: error: ambiguous implicit values:
 both value x of type => Int
 and value y of type => Int
 match expected type Int
       implicitly[Int]
                  ^
```

But it is not an error to declare several implicit arguments of the *same* type, e.g.:

```
def f(a: String)(implicit x: MyType, y: MyType)
implicit val z: MyType = ???

f("abc") // Same as f("abc")(z, z) since z is the unique implicit value of type MyType.
```

In the example above, the arguments `x` and `y` will be set to the same value, `z`. A compile-time error will occur if no `implicit` value of type `MyType` is visible in the current scope:

```
scala> implicitly[MyType]
<console>:12: error: could not find implicit value for parameter e: MyType
       implicitly[MyType]
                  ^
```

8.2.3 Implementing typeclasses by making instances `implicit`

The idea is to declare typeclass instance values as `implicit`. Typeclass instance arguments of functions are also declared as `implicit`. As a result, typeclass instances will be passed to all PTVFs automatically (as long as the appropriate implicits are visible in the scope of the PTVFs). This makes typeclasses easier to use because instance values need to be written out much less often.

The example with the `Frac` typeclass is implemented using implicit values like this:

```
final case class Frac[T](add: (T, T) => T, intdiv: (T, Int) => T)
implicit val fracBD = Frac[BigDecimal]( (x, y) => x + y,  (x, n) => x / n )
implicit val fracD = Frac[Double]( (x, y) => x + y,  (x, n) => x / n )
```

To define the function `avg[T]`, we declare an implicit argument as a `Frac` typeclass instance for `T`:

```
def avg[T](s: Seq[T])(implicit frac: Frac[T]): T = { // Assuming ‘s‘ is a non-empty sequence.
  val sum = s.reduce(frac.add)     // Here, ‘reduce‘ would fail on an empty sequence ‘s‘.
  frac.intdiv(sum, s.length)       // Compute ‘sum/length‘.
}
```

It is now easier to use the function `avg` because the typeclass instances are inserted automatically:

```
scala> avg(Seq(1.0, 2.0, 3.0))
res0: Double = 2.0

scala> avg(Seq(BigDecimal(1.0), BigDecimal(2.0)))
res1: BigDecimal = 1.5
```

Scala supports the “type constraint” syntax for implicit typeclass instance arguments: the code

```
def f[A, B](args...)(implicit t1: Typeclass1[A], t2: Typeclass2[B])
```

is equivalent to the shorter code:

```
def f[A: Typeclass1, B: Typeclass2](args...)
```

The shorter code omits the names (`t1`, `t2`) of the typeclass instances. These values can be extracted via the standard function `implicitly` because all implicit arguments are automatically made available as implicit values in the scope of a function's body. The code of `avg[T]` can then be written as:

```
def avg[T: Frac](s: Seq[T]): T = {
  val frac = implicitly[Frac[T]]
  val sum = s.reduce(frac.add)
  frac.intdiv(sum, s.length)
}
```

When an implicit argument is required, the Scala compiler will search for implicit values of the required type in different places of the code. If implicit values are declared in another module, they can be made available by using an `import` declaration. In many cases, explicit `import` declarations can be avoided. One way to avoid them is to declare the required implicit values within the **companion object** of the typeclass (i.e., the Scala `object` with the same name as the type constructor):

```
final case class Frac[T](add: (T, T) => T, intdiv: (T, Int) => T)

object Frac {   // The companion object of ‘Frac[T]‘ creates some typeclass instances as ‘implicit‘.
  implicit val fracBD = Frac[BigDecimal]( (x, y) => x + y,  (x, n) => x / n )
  implicit val fracD = Frac[Double]( (x, y) => x + y,  (x, n) => x / n )
}
```

Whenever a function needs an implicit value of type `Frac[T]` for a specific type `T`, the Scala compiler will automatically look within the companion object of `Frac` (as well as within the companion object of the type `T`) for any instances declared there. So, the programmer's code will not need to `import` those typeclass instances explicitly even if the companion object is in a different module:

```
scala> avg(Seq(1.0, 2.0, 3.0))
res0: Double = 2.0
```

8.2.4 Extension methods

In Scala, function applications can use three kinds of syntax:

1. The "function" syntax: arguments are to the right of the function as in `plus(x, y)` or `plus(x)(y)`.
2. The "method" syntax: the first argument is to the left of the function, and all other arguments (if any) are placed to the right of the function, as in `x.plus(y)` or `xs.foldLeft(0)(updater)`.
3. The "infix method" syntax (only applies to functions with two *explicit* arguments): no dot character is used. For example, `x plus y`, or `xs map {x => x + 1}`, or `Set(1, 2, 3) contains 1`.

The last two syntax features are often used when writing chains of function applications, such as `xs.map(f).filter(g)`, because that code is easier to read than `filter(map(xs, f), g)`.

The method syntax is available only for methods defined in a class. A special feature of Scala allows us to add new functions with method syntax to previously defined types. New functions added in this way are called **extension methods**.

Suppose we would like to convert a previously defined function, e.g.:

```
def func(x: X, y: Y): Z = { ... }
```

into an extension method on the type `X` with the syntax `x.func(y)`. To do that, we define a new helper `class` that has a method named `func`. The class's constructor must be declared as an `implicit` function having a *single* argument of type `X`:

```
implicit class FuncSyntax(x: X) {  def func(y: Y): Z = ... }
```

After this declaration, we can write the method syntax `x.func(y)` as well as the infix method syntax `x func y`. The new syntax will work because the compiler automatically rewrites `x.func(y)` into `new FuncSyntax(x).func(y)`, creating a new temporary value `FuncSyntax(x)`. The method `func` will be available since it is defined in the class `FuncSyntax`. (Scala 3 will have a shorter syntax.)

As an example, let us define an extension method `average` for the type `Seq[T]`. Both the type parameter `T` and its typeclass constraint can be written in the constructor of the helper class:

```
implicit class AvgSyntax[T: Frac](xs: Seq[T]) {
  def average: T = avg(xs) // Use a different name, ‘average‘, to avoid name clash with ‘avg‘.
}
```

We can now use the method `average` on numeric sequences:

```
scala> Seq(1.0, 2.0, 3.0).average
res0: Double = 2.0
```

The Scala compiler automatically rewrites the syntax `Seq(1.0, 2.0, 3.0).average` as the expression:

```
new AvgSyntax(Seq(1.0, 2.0, 3.0))(implicitly[Frac[Double]]).average
```

In this way, the method `average` is actually invoked on a temporarily created value of type `AvgSyntax`. These values will be created automatically because the class constructor of `AvgSyntax` is declared as `implicit`. Since the constructor of `AvgSyntax` includes the typeclass constraint `[T: Frac]`, we will not be able to create values of type `AvgSyntax[T]` for types `T` not in the type domain of `Frac`.

This example illustrates the convenience of implementing PTVFs as extension methods. An extension method is defined only once but automatically becomes available for all types in the domain of the typeclass. Because of the typeclass constraint, the new method will be available *only* on values of supported types.

This convenience comes at a cost: helper classes such as `AvgSyntax` need to be explicitly imported into every scope where extension methods are used, with a declaration like this:

```
import some.library.having.a.longwinded.packagename.AvgSyntax
```

If the helper class is defined in some library, the programmer will have to look at the library's source code to determine the full name of the helper class that needs to be imported.

8.2.5 Solved examples: Implementing typeclasses in practice

We will now look at some practical examples of programming tasks implemented via typeclasses.

Example 8.2.5.1 (metadata extractors) An application needs to work with data structures implemented in various external libraries. All those data structures are case classes containing certain metadata: "name" (a `String`) and "count" (a `Long` integer). However, the specific data structures define the metadata differently, each in its own way:

```
final case class Data1(p: String, q: String, r: Long) // "name" = p, "count" = r
final case class Data2(s: String, c: Long, x: Int)    // "name" = s, "count" = c
final case class Data3(x: Int, y: Long, z: String)    // "name" = z, "count" = x * y
```

The task is to implement two functions, `getName[T]` and `getCount[T]`, for extracting the metadata out of the data structures of type `T`, where `T` is one of `Data1`, `Data2`, `Data3`. Type signatures and sample tests:

```
def getName[T: HasMetadata](t: T): String = ???
def getCount[T: HasMetadata](t: T): Long = ???

scala> getName(Data2("abc", 123, 0))
res0: String = abc

scala> getCount(Data3(10, 20, "x"))
res1: Long = 200
```

Solution We will implement a typeclass `HasMetadata` and declare instances only for `Data1`, `Data2`, and `Data3`. The code for extracting the metadata will be contained within the typeclass instances. Since the metadata extractors have types `T => String` and `T => Long`, a simple solution is to define the typeclass as a `case class` containing these two functions:

```
final case class HasMetadata[T](getName: T => String, getCount: T => Long)
```

The required typeclass instances are declared as implicit values within the companion object:

```
object HasMetadata {    // Extract metadata from each type as appropriate.
  implicit val data1 = HasMetadata[Data1](_.p, _.r)
  implicit val data2 = HasMetadata[Data2](_.s, _.c)
  implicit val data3 = HasMetadata[Data3](_.z, data3 => data3.x * data3.y)
}
```

Now we can define `getName` and `getCount` as PTVFs with typeclass constraints. First, let us write the code using an implicit argument to pass the typeclass instance:

```
def getName[T](t: T)(implicit ti: HasMetadata[T]): String = ti.getName(t)
def getCount[T](t: T)(implicit ti: HasMetadata[T]): Long = ti.getCount(t)
```

Or we could use the typeclass constraint syntax, replacing `ti` by calls to `implicitly[...]`:

```
def getName[T: HasMetadata](t: T): String = implicitly[HasMetadata[T]].getName(t)
def getCount[T: HasMetadata](t: T): Long = implicitly[HasMetadata[T]].getCount(t)
```

This code defines PTVFs `getName` and `getCount` with the type domain that contains the three types `Data1`, `Data2`, `Data3`. In order to add a new type, say `Data4`, to the type domain, we will need to declare a new typeclass instance as an implicit value of type `HasMetadata[Data4]`. New implicit values can be defined anywhere in the code, not necessarily within the companion object `HasMetadata`. To avoid extra `import` statements, the implicit value may be defined within the companion object of `Data4`:

```
final case class Data4(x: Int, message: String)
object Data4 {   // Add Data4 to the type domain of HasMetadata:
  implicit val data4 = HasMetadata[Data4](_.message, _.x.toLong)
}

scala> getName(Data4(1, "abc"))
res2: String = abc
```

For convenience, let us declare the metadata extractors as extension methods:

```
implicit class ExtractorsSyntax[T: HasMetadata](t: T) {
  def name: String = getName(t)
  def count: Long = getCount(t)
}
```

With this definition, we can write:

```
scala> Data2("abc", 123, 0).name
res3: String = "abc"

scala> Data3(10, 20, "x").count
res4: Long = 200
```

The code looks as if the new methods `name` and `count` became available for each of the supported data types. It is important that the new methods were added *without* any changes to the implementations of `Data1`, `Data2`, or `Data3`. Those implementations may reside in external libraries whose code we do not need to modify. The typeclass pattern enables us to to add externally defined types to a type domain whenever necessary, and to implement new PTVFs for them.

Example 8.2.5.2 (counters) A certain application needs to count the number of files processed and to send this number to external observability services. The functionality of a counter is provided by an external library as a special class `Counter` with an `inc` method that increments the counter. To test the code, we want to be able to pass a test-only counter provided by another library as a type `TestCounter` with an `incr` method. The task is to implement a function `bump[C]()`, where `C` is a type constrained to be one of the supported types of counters. The type signature and sample code:

```
def bump[C]()(...) = ???

val counter = Counter(...)
val testCounter = TestCounter(...)

bump(counter)       // Should call counter.inc()
bump(testCounter)   // Should call testCounter.incr()
```

Solution We will implement a typeclass `Bumpable` whose type domain contains the types `Counter` and `TestCounter`. Typeclass instances should allow us to increment a counter of any supported type. So, a typeclass instance value of type `Bumpable[C]` needs to contain a function of type `C => Unit` that will increment a counter of type `C` appropriately:

```
final case class Bumpable[C](bump: C => Unit)
```

We can now create the typeclass instances and implement the function `bump[C]`:

```
object Bumpable {
  implicit val b1 = Bumpable[Counter](c => c.inc())
  implicit val b2 = Bumpable[TestCounter](c => c.incr())

  def bump[C](counter: C)(implicit ti: Bumpable[C]): Unit = ti.bump(counter)
}
```

An equivalent implementation with the type constraint syntax looks like this:

```
def bump[C: Bumpable](counter: C): Unit = implicitly[Bumpable[C]].bump(counter)
```

Example 8.2.5.3 (default values) Certain types have naturally chosen "default" values (e.g., integer zero, empty string, empty array, etc.). The task is to implement a function `default[T]` restricted to types `T` for which default values are available. The required type signature and sample tests:

```
def default[T: HasDefault]: T = ???

scala> default[Int]
res0: Int = 0

scala> default[Double]
res1: Double = 0.0
```

Solution We need to define a PTVF `default[T]` with a type domain that contains (at least) the types `Int` and `Double`. For every supported type `T`, we need to store the known default value of that

type. So, the typeclass instance can be defined as a simple wrapper for values of type `T`:

```
final case class HasDefault[T](value: T)
```

Typeclass instances are declared straightforwardly:

```
object HasDefault {
  implicit val defaultInt = HasDefault[Int](0)
  implicit val defaultDouble = HasDefault[Double](0.0)
  implicit val defaultString = HasDefault[String]("")
  implicit val defaultUnit = HasDefault[Unit](())
}
```

The implementation of `default[T]` is written as:

```
def default[T](implicit ti: HasDefault[T]) = ti.value
```

When using the typeclass constraint syntax, the code is:

```
def default[T: HasDefault]: T = implicitly[HasDefault[T]].value
```

How to define an empty list as a default value for lists of *any* chosen type? We cannot write:

```
implicit val defaultList = HasDefault[List[A]](List())     // Error: 'A' is undefined.
```

The type parameter `A` needs to be defined in the left-hand side. Since Scala 2 does not support `val` declarations with type parameters (Scala 3 does), we need to write the typeclass instance as a `def`:

```
implicit def defaultList[A] = HasDefault[List[A]](List())
```

Another example of a `HasDefault` instance with a type parameter is for functions of type $A \to A$:

```
implicit def defaultFunc[A]: HasDefault[A => A] = HasDefault[A => A](identity)
```

Types that have default values are also called **pointed** types. This book defines the typeclass `Pointed` for pointed *functors* (Section 8.3.5) because they are more widely used than pointed types.

Example 8.2.5.4 (semigroups) In many cases, data items can be combined or merged into a larger data item of the same type. For instance, two numbers can be added, two sets combined into one set, two strings concatenated into one string, and two lists into one list. The "merge" operation can be defined as a function `combine[T]` taking two arguments of type `T` and returning a new value of type `T`. We will denote that operation by $\oplus$, e.g., we write $x \oplus y$. In all the examples just mentioned (integers, strings, lists, etc.), that operation is associative:

$$\forall(x, y, z).\ (x \oplus y) \oplus z = x \oplus (y \oplus z) \quad . \tag{8.1}$$

This associativity law makes parentheses in the expression $x \oplus y \oplus z$ unnecessary.

A type `T` with an associative binary operation is called a **semigroup**. The task in this example is to define the semigroup operation for the types `Int`, `String`, and `List[A]`.

Solution For every supported type T, the required data is a function of type $T \times T \to T$. So, we define the typeclass as a wrapper over that type:

```
final case class Semigroup[T](combine: (T, T) => T)
```

The typeclass instances for the supported types are defined using a short syntax as:

```
object Semigroup {
  implicit val semigroupInt = Semigroup[Int](_ + _)
  implicit val semigroupString = Semigroup[String](_ + _)
  implicit def semigroupList[A] = Semigroup[List[A]](_ ++ _)
}
```

The function `combine[T]` is implemented as:

```
def combine[T](x: T, y: T)(implicit ti: Semigroup[T]): T = ti.combine(x, y)
```

Since `combine` is a binary operation, it is convenient to define infix method syntax for it:

```
implicit class SemigroupSyntax[T: Semigroup](x: T) { def |+|(y: T): T = combine(x, y) }
```

After this definition, we may use the infix operation `|+|` like this:

```
scala> List(1, 2, 3) |+| List(4)
res0: List[Int] = List(1, 2, 3, 4)
```

Due to the associativity law (8.1), the result of `x |+| y |+| z` does not depend on the choice of parentheses: `(x |+| y) |+| z == x |+| (y |+| z)`. This makes programs written using the semigroup operation `|+|` easier to understand and reason about.

Semigroup types represent data that can be pairwise "merged" in a certain well-defined way. Using the `Semigroup` typeclass, we can write code that is parameterized by the type of "mergeable" data. As an example, given a `Seq[T]` where the type `T` is a semigroup, we can "merge" all elements to compute a result of type `T`. This computation can be implemented as a function parameterized by `T`:

```
def merge[T: Semigroup](ts: Seq[T]): T = ts.reduce(combine[T])
```

This function assumes a non-empty input sequence `ts` whose elements are of a semigroup type `T`. We can also implement the same function as an extension method:

```
implicit class SumSyntax[T: Semigroup](ts: Seq[T]) { def merge: T = ts.reduce(combine[T]) }
```

With the previous definitions, we can now evaluate expressions such as:

```
scala> Seq(1, 2, 3).merge
res1: Int = 6

scala> Seq(List(), List(true), List(), List(true, false)).merge
res2: List[Boolean] = List(true, true, false)
```

It is important that the associativity law (8.1) should hold for each of the supported types. If that is not so, programs written with `merge` will not work as expected. (For instance, programmers would certainly expect that `xs.merge |+| ys.merge == (xs ++ ys).merge` for any sequences `xs` and `ys`.) However, the code of the typeclass *does not* check the associativity law. It is the responsibility of the programmer to verify that the implementation of each typeclass instance is lawful.

The associativity law of integer addition is a standard arithmetic identity:

$$(x + y) + z = x + (y + z) \quad .$$

Verifying associativity for lists and strings (which are lists of characters) is intuitively simple because concatenation preserves the order of elements. If `x`, `y`, and `z` are lists, the concatenation `(x ++ y) ++ z` is a list containing all elements from `x`, `y`, and `z` in their original order. It is evident that the concatenation `x ++ (y ++ z)` is a list with the same elements in the same order. However, a rigorous proof of the associativity law for lists, starting from the code of the `concat` function, requires significant work (see Section 8.5.2).

Example 8.2.5.5 (alternative semigroup implementations) The definitions of the semigroup operation $\oplus$ as concatenation for strings and as addition for integers may appear to be "natural". However, alternative implementations are useful in certain applications. As long as the associativity law holds, *any* function of type $T \times T \to T$ may be used as the semigroup operation. The task of this example is to show that the following implementations of the semigroup operation are lawful and to implement the corresponding typeclass instances in Scala.

(a) For any type T, define $x \oplus y \stackrel{\text{def}}{=} x$ (ignoring the value of y).

(b) For pair types $T \stackrel{\text{def}}{=} A \times B$, define the operation $\oplus$ by $(a_1 \times b_1) \oplus (a_2 \times b_2) \stackrel{\text{def}}{=} a_1 \times b_2$.

(c) For $T \stackrel{\text{def}}{=} \text{String}$, define $x \oplus y$ as the longer of the strings x and y.

(d) For $T \stackrel{\text{def}}{=} S \to S$ (the type S is fixed), define $x \oplus y \stackrel{\text{def}}{=} x \,\text{⨾}\, y$ (the forward function composition) or $x \oplus y \stackrel{\text{def}}{=} x \circ y$ (the backward function composition).

Solution **(a)** To verify the associativity law, use the definition $x \oplus y \stackrel{\text{def}}{=} x$ to compute:

$$(x \oplus y) \oplus z = x \oplus z = x \quad , \quad x \oplus (y \oplus z) = x \quad .$$

So the associativity law holds: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ for any x, y, z.

It is clear that $x \oplus y \oplus z \oplus ... = x$ for any number of values; the binary operation keeps the first value and ignores all further values. We can implement this semigroup instance at once for all types T:

```
implicit def semigroup1[T] = Semigroup[T]{ (x, y) => x }
```

Similarly, the definition $x \oplus y \stackrel{\text{def}}{=} y$ gives an associative binary operation for a (different) semigroup.

(b) To verify the associativity law:

$$((a_1 \times b_1) \oplus (a_2 \times b_2)) \oplus (a_3 \times b_3) = (a_1 \times b_2) \oplus (a_3 \times b_3) = a_1 \times b_3 \quad ,$$
$$(a_1 \times b_1) \oplus ((a_2 \times b_2) \oplus (a_3 \times b_3)) = (a_1 \times b_1) \oplus (a_2 \times b_3) = a_1 \times b_3 \quad .$$

The implementation is possible for any types A, B:

```
implicit def semigroup2[A, B] = Semigroup[(A, B)]{ case ((a1, b1), (a2, b2)) => (a1, b2) }
```

One use case for this semigroup is to maintain a pair of timestamps for the first and the last events in a temporally ordered series. Merging two such pairs for consecutive events means to keep the first value from the first pair and the second value from the second pair.

(c) It is clear that $x \oplus y \oplus z$ is the longest of the strings x, y, and z. Since the definition of "longest" does not depend on the order in which we select pairs for comparison, the operation is associative. (For the same reason, any "maximum" or "minimum" operation is associative.) Implementation:

```
implicit val semigroup3 = Semigroup[String]{ (x, y) => if (x.length > y.length) x else y }
```

(d) The composition of functions is associative (see proofs in Section 4.2.2). Whether we choose to define $x \oplus y \oplus z = x ; y ; z$ or $x \oplus y \oplus z = x \circ y \circ z = z ; y ; x$, the results do not depend on inserting parentheses. The code for these two typeclass instances is:

```
implicit def semigroup4[S] = Semigroup[S => S]{ (x, y) => x andThen y }
implicit def semigroup5[S] = Semigroup[S => S]{ (x, y) => x compose y }
```

Example 8.2.5.6 (monoids) When a data type is a semigroup, it is often possible to find a special value that acts as an "default" value with respect to the semigroup operation. Merging with the default value will not change any other value. For instance, concatenating with an empty list does not change any other list; so the empty list plays the role of the default value for lists. Merging an empty set with any other set does not change the other set; so the empty set is the default for sets.

A semigroup with such a default value is called a **monoid**. Formally, a type T is a monoid when there is an associative binary operation $\oplus_T$ and a chosen default or "empty" value ($e^{:T}$) such that:

$$\text{for all } x^{:T} : \quad e \oplus_T x = x \quad , \qquad x \oplus_T e = x \quad . \tag{8.2}$$

The laws (8.2) are called the **identity laws** of a monoid.

The task in this example is to define a typeclass describing monoids.

Solution The typeclass instances should contain the same information as semigroups and additionally the default value of type T. So, we write:

```
final case class Monoid[T](combine: (T, T) => T, empty: T)
```

Let us define some typeclass instances for illustration:

```
object Monoid {
  implicit val monoidInt = Monoid[Int](_ + _, 0)
  implicit val monoidString = Monoid[String](_ + _, "")
  implicit def monoidList[A] = Monoid[List[A]](_ ++ _, List())
  implicit def monoidFunc[A] = Monoid[A => A](_ andThen _, identity)
}
```

Monoids formalize the general properties of data aggregation. Section 8.3.4 will study monoids in more detail and show further examples of their use. At this point, we look at one more example that defines a `Monoid` instance using two previously defined typeclasses, `Semigroup` and `HasDefault`.

Example 8.2.5.7 (monoids as semigroups with default) A monoid combines the properties of a semigroup and a type with a default value. If a type `T` is a semigroup and has a default value, it is likely that `T` is a monoid. The task is to define a `Monoid` typeclass instance given `Semigroup` and `HasDefault` instances for a type `T`.

Solution We need to define a function `monoidOf[T]` that returns the required monoid typeclass instance for `T`. The typeclass constraints for this function are `Semigroup` and `HasDefault`. So the type signature must be:

```
def monoidOf[T](implicit ti1: Semigroup[T], ti2: HasDefault[T]): Monoid[T] = ???
```

To implement a value of type `Monoid[T]`, we need to provide a function of type $T \times T \to T$ and a value of type T. Precisely that data is available in the typeclass instances of `Semigroup` and `HasDefault`, and so it is natural to use them:

```
def monoidOf[T](implicit ti1: Semigroup[T], ti2: HasDefault[T]): Monoid[T] =
  Monoid(ti1.combine, ti2.value)
```

Using the type constraint syntax, the equivalent code is:

```
def monoidOf[T: Semigroup : HasDefault]: Monoid[T] =
  Monoid(implicitly[Semigroup[T]].combine, implicitly[HasDefault[T]].value)
```

We can also define this function as an `implicit`, so that every type `T` with a `Semigroup` and `HasDefault` instances will automatically receive a `Monoid` typeclass instance as well.

Writing the types of the `Semigroup`, `HasDefault`, and `Monoid` instances in the type notation, we get:

$$\text{Semigroup}^T \stackrel{\text{def}}{=} T \times T \to T \quad , \qquad \text{HasDefault}^T \stackrel{\text{def}}{=} T \quad , \qquad \text{Monoid}^T \stackrel{\text{def}}{=} (T \times T \to T) \times T \quad .$$

It is now clear that the following equivalence holds:

$$\text{Monoid}^T \cong \text{Semigroup}^T \times \text{HasDefault}^T \quad .$$

Indeed, the code for `monoidOf` computes a pair of values from the `Semigroup` and `HasDefault` instances.

Is this implementation lawful with respect to the monoid laws? The associativity law will hold for the monoid if the `Semigroup` typeclass instance was already lawful. However, the value stored in the `HasDefault` instance is not guaranteed to satisfy the identity laws (8.2) with respect to the `combine` operation stored in the `Semigroup` instance. The programmer must verify that the identity laws hold. It can happen that, for some type T, typeclass instances of `Semigroup` and `HasDefault` already exist but are not "compatible", so that the monoid identity laws do not hold. In that case, a simple combination of `Semigroup` and `HasDefault` instances will not work, and a different `Monoid` instance must be defined.

Are there alternative implementations of the `Monoid` typeclass instance given `Semigroup` and `HasDefault` instances? The function `monoidOf` needs to produce a value of type $(T \times T \to T) \times T$ given values of type $T \times T \to T$ and a value of type T:

$$\text{monoidOf} : (T \times T \to T) \times T \to (T \times T \to T) \times T \quad .$$

When the type signature of `monoidOf` is written in this notation, it is clear that `monoidOf` should be the identity function; indeed, that is what our code translates to. Although there are many other implementations of the same type signature, only the code shown above will satisfy the monoid laws. An example of an unlawful implementation is:

```
def badMonoidOf[T](implicit ti1: Semigroup[T], ti2: HasDefault[T]): Monoid[T] =
  Monoid((x, y) => ti1.combine(x, ti1.combine(x, y)), ti2.value)
```

This implementation defines the monoid operation as $x \oplus x \oplus y$ instead of the correct definition $x \oplus y$. If we set $y = e_T$, we will get $x \oplus x$ instead of x, violating one of the identity laws.

8.2.6 Typeclasses for type constructors

As an example of a function parameterized by a type constructor, consider the following function `inject` with a type parameter `F` constrained to be a functor:

```
def inject[F[_]: Functor, A, B](a: A, f: F[B]): F[(A, B)] = ???
```

The `Functor` typeclass implementing this constraint will use the syntax `Functor[F[_]]` because the type parameter `F` is itself a type constructor.

What information needs to be wrapped by a typeclass instance? A functor `F` must have a `map` function with the standard type signature:

```
def map[A, B](fa: F[A])(f: A => B): F[B]
```

In the type notation, this type signature is written as:

$$\text{map} : \forall(A, B).\, F^A \to (A \to B) \to F^B \quad .$$

So, a typeclass instance of the `Functor` typeclass must contain this function as a value. But defining the typeclass as before via a `case class` does not work with Scala 2:

```
final case class Functor[F[_]](map: ∀(A,B).F[A] => (A => B) => F[B])   // Not possible in Scala 2.
```

Scala 3 will directly support an argument type that *itself* contains type quantifiers such as $\forall(A, B)$. In Scala 2, we need to represent such "nested" type quantifiers by writing a `trait` with a `def` method:

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

The type constructor `Functor` has the type parameter `F[_]`, which must be itself a type constructor. For any type constructor `F`, a value of type `Functor[F]` is a wrapper for a value of type $\forall(A, B).\, F^A \to (A \to B) \to F^B$.

Values of type `Functor` (i.e., typeclass instances) are implemented with the "`new { ... }`" syntax:

```
implicit val functorSeq = new Functor[Seq] {
  def map[A, B](fa: Seq[A])(f: A => B): Seq[B] = fa.map(f)
}
```

This is currently the most common way of defining typeclasses in Scala.

It is convenient to declare `map` as an extension method on the `Functor` type constructors:

```
implicit class FunctorSyntax[F[_]: Functor, A](fa: F[A]) {    // Syntax helper class.
  def map[B](f: A => B): F[B] = implicitly[Functor[F]].map(fa)(f)
}
```

If this class definition is in scope, the `map` method becomes available for values of functor types.

Using the `Functor` typeclass and the syntax helper, we can now implement the function `inject`:

```
def inject[F[_]: Functor, A, B](a: A, f: F[B]): F[(A, B)] = f.map { b => (a, b) }

scala> inject("abc", Seq(1, 2, 3))   // Assuming that implicit Functor[Seq] is in scope.
res0: Seq[(String, Int)] = List(("abc", 1), ("abc", 2), ("abc", 3))
```

Just like the `Monoid` typeclass, the code of the `Functor` typeclass does not enforce the functor laws on the implementation. It is the programmer responsibility to verify that the laws hold.

One way of checking the laws is to use the `scalacheck` library[1] that automatically runs random tests for the given assertions, trying to discover a set of values for which some assertion fails. Using the `Functor` typeclass constraint, we can implement a function (in our terminology, a PTVF) that checks the functor laws for *any* given type constructor `F[_]`:

```
import org.scalacheck.Arbitrary                                  // Necessary imports and definitions.
import org.scalatest.prop.GeneratorDrivenPropertyChecks
class FunctorTest extends Matchers with GeneratorDrivenPropertyChecks {
  def checkFunctorLaws[F[_], A, B, C]()(implicit ff: Functor[F],     // Use the ‘Arbitrary‘ typeclass
      fa: Arbitrary[F[A]], ab: Arbitrary[A => B], bc: Arbitrary[B => C]) = {    // from ‘scalacheck‘.
    forAll { (fa: F[A]) => fa.map(identity[A]) shouldEqual fa }  // Identity law. "For all ‘fa‘, ..."
    forAll { (f: A => B, g: B => C, fa: F[A]) =>          // Composition law. The assertion must hold
      fa.map(f).map(g) shouldEqual fa.map(f andThen g)    // for all ‘f‘, ‘g‘, ‘fa‘.
    }
  }
  // Check the laws for F = Seq using specific types instead of type parameters A, B, C.
  checkFunctorLaws[Seq, Int, String, Double]()
}
```

[1] https://www.scalacheck.org

The `scalacheck` library will substitute a large number of random values into the given assertions, Note that the laws are being tested only with a finite number of values and with type parameters set to specific types. While it is useful to test laws with `scalacheck` (we might find a bug), only a symbolic derivation provides a rigorous proof that the laws hold. One of the main themes of this book is to show how to perform symbolic derivations efficiently.

8.3 Deriving typeclass instances via structural analysis of types

In Chapter 6 we analyzed the structure of functors by checking which of the six standard type constructions can make new functors out of previous ones. We will now apply the same **structural analysis** to various typeclasses. Is a product of two monoids a monoid? Is a co-product of two semigroups a semigroup? Answers to such questions will enable us to:

- Quickly decide whether a given type can have a typeclass instance of `Monoid`, `Semigroup`, etc.
- If so, derive the code for the new typeclass instance without guessing.
- Have assurance that the required typeclass laws will hold for newly constructed instances.

In the following sections, we will show how to use this approach for some simple typeclasses.

8.3.1 Extractors

Our first typeclass is a generalization of Example 8.2.5.1. The functionality was to extract metadata of fixed types from a value of type `T`. The typeclass instance contained a pair of functions:

```
final case class HasMetadata[T](getName: T => String, getCount: T => Long)
```

In the type notation, this type constructor is written as

$$\text{HasMetadata}^T \stackrel{\text{def}}{=} (T \to \text{String}) \times (T \to \text{Long}) \quad .$$

A standard type equivalence (see Table 5.6) shows that this type is equivalent to $T \to \text{String} \times \text{Long}$. This motivates us to denote $\text{String} \times \text{Long}$ by Z and to consider a typeclass whose instances are values of type $T \to Z$. We may call this typeclass a "Z-extractor" since types T from its type domain permit us somehow to extract values of type Z. With a fixed type Z, we denote the typeclass by

$$\text{Extractor}^T \stackrel{\text{def}}{=} T \to Z \quad .$$

```
final case class Extractor[Z, T](extract: T => Z)
```

What kind of types can have an Extractor typeclass instance? To answer that question, we apply structural analysis and check whether any of the standard type constructions produce new typeclass instances. The results are summarized in Table 8.1. Let us show the required calculations.

Fixed types We check whether an `Extractor` typeclass instance can be computed for the `Unit` type or for another fixed type C. To compute $\text{Extractor}^{\mathbb{1}} = \mathbb{1} \to Z$ requires creating a value of type Z from scratch, which we cannot do in a fully parametric function. For a fixed type C, a value of type $\text{Extractor}^C = C \to Z$ can be computed only if we can compute a value of type Z given a value of type C. This is possible only if we choose C as $C = Z$. The typeclass instance for Extractor^Z is implemented as an identity function of type $Z \to Z$:

```
implicit def extractorZ[Z] = Extractor[Z, Z](identity)
```

Type parameters Creating a typeclass instance Extractor^A for an arbitrary type A means to compute $\forall A. A \to Z$; this is not possible since we cannot create values of type Z using only a value of an unknown type A. So, there is no `Extractor` for arbitrary types `A`.

Products If the types A and B are known to be within the type domain of `Extractor`, can we add the pair $A \times B$ to that domain? If we can extract a value of type Z from each of two values $a^{:A}$ and $b^{:B}$, we can certainly extract a value of type Z from the product $a \times b$ by choosing to extract only from a or only from b. So, it appears that we have two possibilities for implementing the typeclass for $A \times B$. Reasoning more rigorously, we see that computing a new typeclass instance from two previous ones requires implementing a conversion function with type signature:

$$\forall(A, B, Z).\, \text{Extractor}^A \times \text{Extractor}^B \to \text{Extractor}^{A\times B} = (A \to Z) \times (B \to Z) \to A \times B \to Z \quad .$$

We can derive only two fully parametric implementations of this type signature:

$$f^{:A\to Z} \times g^{:B\to Z} \to a \times b \to f(a) \qquad \text{and} \qquad f^{:A\to Z} \times g^{:B\to Z} \to a \times b \to g(b) \quad .$$

Both implementations give a valid `Extractor` instance (since there are no laws to check). However, every choice will use one of the two `Extractor` instances and ignore the other. So, we can simplify this construction by keeping the typeclass constraint only for A and allowing *any* type B:

$$\text{extractorPair} : \forall(A, B, Z).\, \text{Extractor}^A \to \text{Extractor}^{A\times B} \quad .$$

```
def extractorPair[Z, A, B](implicit ti: Extractor[Z, A]) =
  Extractor[Z, (A, B)] { case (a, b) => ti.extract(a) }
```

If A has an `Extractor` instance, the product of A with any type B also has an `Extractor` instance. Examples of this construction are $Z \times B$ and $P \times Q \times Z$ (since the type Z itself has an `Extractor` instance).

Co-products Given typeclass instances Extractor^A and Extractor^B, can we compute a value of type Extractor^{A+B}? Writing out the types, we get:

$$\text{Extractor}^A \times \text{Extractor}^B \to \text{Extractor}^{A+B} = (A \to Z) \times (B \to Z) \to A + B \to Z \quad .$$

Due to a known type equivalence (Table 5.6), we have a unique implementation of this function:

$$\text{extractorEither} \stackrel{\text{def}}{=} f^{:A\to Z} \times g^{:B\to Z} \to \begin{array}{|c||c|} & Z \\ \hline A & a \to f(a) \\ B & b \to g(b) \end{array} \quad .$$

```
def extractorEither[Z, A, B](implicit ti1: Extractor[Z, A], ti2: Extractor[Z, B]) =
  Extractor[Z, Either[A, B]] {
    case Left(a)    => ti1.extract(a)
    case Right(b)   => ti2.extract(b)
}
```

So, the co-product of A and B can be given a unique `Extractor` instance.

Since the product and the co-product constructions preserve `Extractor` instances, we conclude that any polynomial type expression has an `Extractor` instance as long as every product type contains at least one Z or another `Extractor` type. For example, the type expression

$$A \times Z + Z \times (P + Z \times Q) + B \times C \times Z$$

is of that form and therefore has an `Extractor` instance. Generally, a polynomial in Z will have an `Extractor` instance only if the polynomial is of the form $Z \times P^Z$ for some functor P.

Function types We need to investigate whether $C \to A$ or $A \to C$ can have an `Extractor` instance for some choice of C, assuming that we have an instance for A. The required conversion functions must have type signatures:

$$\text{Extractor}^A \to \text{Extractor}^{A\to C} \qquad \text{or} \qquad \text{Extractor}^A \to \text{Extractor}^{C\to A} \quad .$$

Writing out the types, we find:

$$(A \to Z) \to (A \to C) \to Z \qquad \text{or} \qquad (A \to Z) \to (C \to A) \to Z \quad .$$

None of these type signatures can be implemented. The first one is hopeless since we do not have values of type A; the second one is missing values of type C. However, since the type C is fixed, we may store a value of type C as part of the newly constructed type. So, we consider the pair type $C \times (C \to A)$ and find that its `Extractor` instance, i.e., a value of type $C \times (C \to A) \to Z$, can be derived from a value of type $A \to Z$ as:

$$f^{:A\to Z} \to c^{:C} \times g^{:C\to A} \to f(g(c)) \quad .$$

```
def extractorFunc[Z, A, C](implicit ti: Extractor[Z, A]) =
  Extractor[Z, (C, C => A)] { case (c, g) => ti.extract(g(c)) }
```

Examples of this construction are the type expressions $C \times (C \to Z)$ and $D \times (D \to Z \times P)$.

Another situation where an `Extractor` instance exists for the type $C \to A$ is when the type C has a known "default value" e_C (as in the `HasDefault` typeclass). In that case, we may omit the first C in $C \times (C \to A)$ and instead substitute the default value when necessary.

Recursive types We can add recursion to any of the constructions that create a new type with an `Extractor` instance out of a previous such type. For clarity, let us use type constructors F_1, F_2, etc., for describing the new types produced out of previous extractor types. For the product construction, we define $F_1^{B,A} \stackrel{\text{def}}{=} A \times B$. For the co-product construction, define $F_2^{B,A} \stackrel{\text{def}}{=} B + A$ (where B must be also an extractor type). For the function construction, define $F_3^{C,A} \stackrel{\text{def}}{=} C \times (C \to A)$.

Any recursive type equation that uses F_1, F_2, and/or F_3 will define a new recursive type with an `Extractor` instance. An example is the type equation involving the composition of F_2 and F_1:

$$T \stackrel{\text{def}}{=} F_2^{Z\times P, F_1^{T,Q}} = Z \times P + Q \times T \quad . \tag{8.3}$$

We can visualize this recursive type as an "infinite disjunction":

$$\begin{aligned} T &\cong Z \times P + Q \times T \\ &\cong Z \times P + Q \times (Z \times P + Q \times (Z \times P + ...)) \\ &\cong Z \times P \times (\mathbb{1} + Q + Q \times Q + ...) \cong Z \times P \times \text{List}^Q \quad . \end{aligned}$$

Since the resulting type is equivalent to $Z \times C$ where $C \stackrel{\text{def}}{=} P \times \text{List}^Q$, we find that the recursive equation (8.3) is equivalent to the product construction with a different type.

This will happen with any recursive equation containing F_1 and F_2 (but no F_3): since F_1 and F_2 are polynomial functors, the resulting type T will be a recursive polynomial in Z. To support an `Extractor` instance, such polynomials must be the form $Z \times P^Z$ for some functor P.

Recursive equations involving F_3 will produce new examples of `Extractor` types, such as:

$$T \stackrel{\text{def}}{=} F_3^{C, F_2^{Z, F_1^{T,P}}} = C \times (C \to Z + P \times T) \quad . \tag{8.4}$$

Heuristically, this type can be seen as an "infinite" exponential-polynomial type expression:

$$T = C \times (C \to Z + P \times C \times (C \to Z + P \times C \times (C \to Z + ...))) \quad .$$

Types of this form are useful in some applications involving lazy streams.

We will now show how to define an `Extractor` instance for any recursive type defined using F_1, F_2, and/or F_3. A recursive type equation defining a type T can be written generally as $T \stackrel{\text{def}}{=} S^T$, where S is a type constructor built up by composing F_1, F_2, and/or F_3 in some way. (The type constructor S may use Z or other fixed types.) For each of F_1, F_2, and/or F_3, we implemented a function of type:

$$\text{extractorF} : \text{Extractor}^A \to \text{Extractor}^{F^A} \quad .$$

Since S is a composition of F_1, F_2, and/or F_3, we are able to implement `extractorS`:

$$\text{extractorS} : \text{Extractor}^A \to \text{Extractor}^{S^A} \quad .$$

The `Extractor` instance for the recursive type T is then defined recursively as:

$$x^{:T\to Z} \stackrel{\text{def}}{=} \text{extractorS}\,(x) \quad .$$

The types match because the type T is equivalent to the type S^T. As long as the definition of the recursive type T is valid (i.e., the type recursion terminates), the extractor function will also terminate.

To illustrate this construction, let us derive an `Extractor` instance for the type T defined by Eq. (8.4). That equation has the form $T = S^T$ with the type constructor S defined by $S^A \stackrel{\text{def}}{=} C \times (C \to Z + P \times A)$.

```
type S[A] = (C, C => Either[Z, (P, A)])  // The fixed types ‘C‘ and ‘P‘ must be defined previously.
final case class TypeT(t: S[TypeT])      // Define the recursive type ‘TypeT‘.
```

To implement the function of type `Extractor[T] => Extractor[S[T]]`, which is $(T \to Z) \to S^T \to Z$, we begin with a typed hole:

$$f^{:T\to Z} \to s^{:C\times(C\to Z+P\times T)} \to ???^{:Z} \quad .$$

To fill $???^{:Z}$, we could apply $f^{:T\to Z}$ to some value of type T; but the only value of type T can be obtained if we apply the function of type $C \to Z + P \times T$ to the given value of type C. So we write:

$$f^{:T\to Z} \to c^{:C} \times g^{:C\to Z+P\times T} \to g(c) \triangleright ???^{:Z+P\times T\to Z} \quad .$$

The new typed hole has a function type. We can write the code in matrix notation as:

$$\text{extractorS} \stackrel{\text{def}}{=} f^{:T\to Z} \to c^{:C} \times g^{:C\to Z+P\times T} \to g(c) \triangleright \left|\begin{array}{c||c} & Z \\ \hline Z & \text{id} \\ P\times T & _^{:P} \times t^{:T} \to f(t) \end{array}\right| \quad .$$

```
def extractorS[A](f: Extractor[A]): Extractor[S[A]] = Extractor[S[A]] { case (c, g) =>
  g(c) match {
    case Left(z)          => z
    case Right((_, t))    => f.extract(t)
  }
}
```

The recursive construction defines an `Extractor` instance for T by a recursive equation:

$$\text{extractorT} \stackrel{\text{def}}{=} \text{extractorS}\,(\text{extractorT}) \quad . \tag{8.5}$$

```
def extractorT: Extractor[TypeT] = Extractor[TypeT] { case TypeT(t) =>
  extractorS(extractorT).extract(t)
}
```

To test this code, we define a value of type T while setting `C = Int`, `P = Boolean`, and `Z = String`:

```
val t = TypeT((10, x => Right((true, TypeT((x * 2, y => Left("abc")))))))

scala> extractorT.extract(t) // The recursive definition of ‘extractorT‘ terminates.
res0: String = abc
```

Why the recursion terminates The above code shows that the recursive definition (8.5) terminates. Why does it? A recursive definition of the form $x \stackrel{\text{def}}{=} f(x)$ could create an infinite loop, as in this code:

```
def f(x: Int): Int = x + 1
def x: Int = f(x)

scala> x                      // Infinite loop: f(f(f(f(...)))
java.lang.StackOverflowError
```

The code for `extractorT` works because `extractorT` is a value of a *function* type, and because the presence of the case class `Extractor` forces us to rewrite Eq. (8.5) in the form of an "expanded function":

$$\text{extractorT} \stackrel{\text{def}}{=} t \to \text{extractorS}\,(\text{extractorT})\,(t) \quad .$$

```
def extractorT: Extractor[TypeT] = Extractor { case TypeT(t) => extractorS(extractorT).extract(t) }
```

Although a function f is equivalent to its **expanded form** $t \to f(t)$, there is an important difference: using expanded forms in the code will make recursive definitions terminate.

To see an example of that, consider a recursive equation $f \stackrel{\text{def}}{=} k(f)$, where k is some function. The Scala code `val f = k(f)` creates an infinite loop when we compute anything that involves f:

```
def k(f: Int => Int): Int => Int = { x => if (x <= 0) 1 else 2 * f(x - 1) }
def f: Int => Int = k(f)   // This definition is invalid!

scala> f                   // Infinite loop: k(k(k(k(...)))
java.lang.StackOverflowError

scala> f(4)                // Infinite loop: k(k(k(k(...)))(4)
java.lang.StackOverflowError
```

This code is clearly invalid. But if we expand the right-hand side of the recursive equation to:

$$f \stackrel{\text{def}}{=} t \to k(f)(t)$$

instead of $f \stackrel{\text{def}}{=} k(f)$, the code will become valid, and the infinite loop disappears:

```
def f: Int => Int = { x => k(f)(x) }   // This defines f(n) = 2^n for n ≥ 0.

scala> f                               // We can compute f without an infinite loop.
res0: Int => Int = <function1>

scala> f(4)                            // We can compute f(4) without an infinite loop.
res1: Int = 16
```

The recursive use of f now occurs *within* a function body, and so $k(f)$ is evaluated only when f is applied to an argument. This allows the recursive definition of f to terminate.

Summary We derived the constructions that create new types with `Extractor` typeclass instances from previous ones. Any number of these constructions can be combined to create a new type expression that will always have an `Extractor` instance. An example is this type expression:

$$K^{Z,P,Q,R,S} \stackrel{\text{def}}{=} Z \times P + Q \times (Q \to Z + R \times (R \to Z \times S)) \quad .$$

Since the type K is built up step by step from fixed types via the product, co-product, and function constructions, an `Extractor` instance for K can be derived systematically with no guessing:

```
type K[Z,P,Q,R,S] = Either[(Z, P), (Q, Q => Either[Z, (R, R => (Z, S))])]
implicit def extractorK[Z,P,Q,R,S]: Extractor[Z, K[Z,P,Q,R,S]] = {   // Extractor values for types:
  implicit val e1 = extractorPair[Z, Z, S]                          // Z×S. Needs Extractor[Z, Z].
  implicit val e2 = extractorFunc[Z, (Z, S), R]                     // R×(R→Z×S).
  implicit val e3 = extractorEither[Z, Z, (R, R => (Z, S))]         // Z+R×(R→Z×S).
  implicit val e4 = extractorFunc[Z, Either[Z, (R, R => (Z, S))], Q] // Q×(Q→Z+R×(R→Z×S)).
  implicit val e5 = extractorPair[Z, Z, P]                          // Z×P.
  extractorEither[Z, (Z,P), (Q, Q => Either[Z, (R, R => (Z, S))])]   // Extractor for type K.
}
```

The code computes each implicit value `e1`, `e2`, ..., using constructions that require previously computed values to be present as implicits.

The `Extractor` typeclass is often used with `Z = String` as a way to "print" values of different types, and it is then called `Show`. When `Z = Array[Byte]`, the typeclass is often called a "serializer".

Construction	Type signature to implement
The `Unit` type, or other fixed type C	$\text{Extractor}^{\mathbb{1}}$ or Extractor^{C}
Product of extractor type A and any B	$\text{Extractor}^{A} \to \text{Extractor}^{A\times B}$
Co-product of extractor types A and B	$\text{Extractor}^{A} \times \text{Extractor}^{B} \to \text{Extractor}^{A+B}$
Function from or to another type C	$\text{Extractor}^{A} \to \text{Extractor}^{A\to C}$ or $\text{Extractor}^{C\to A}$
Recursive type	$\text{Extractor}^{A} \to \text{Extractor}^{S^A}$ where $T \stackrel{\text{def}}{=} S^T$

Table 8.1: Type constructions producing the `Extractor` typeclass.

8.3.2 Equality comparison: The `Eq` typeclass

In Scala, the built-in operation `==` is not type-safe because the code `x == y` will compile regardless of the types of `x` and `y`. We can replace `==` by a new operation `===` constrained to a typeclass called `Eq`, ensuring that types can be meaningfully compared for equality. The equality comparison for values of type A is a function of type $A \times A \to \mathbb{2}$ (where $\mathbb{2}$ denotes the `Boolean` type). Typeclass instances of `Eq` need to wrap a function of that type. We also define `===` as an extension method:

```
final case class Eq[A](equal: (A, A) => Boolean)
object Eq {
  implicit class EqOps[A: Eq](a: A) {
    def ===(b: A): Boolean = implicitly[Eq[A]].equal(a, b)
  } // Use type-specific comparisons to define some typeclass instances here:
  implicit val eqInt: Eq[Int] = Eq[Int](_ == _)
  implicit val eqString: Eq[String] = Eq[String](_ == _)
}
```

If we define an `Eq` typeclass instance for all types via the built-in `==` operation, we will find that this operation does not work as expected when comparing values of a function type, e.g., `Int => Int`:

```
import Eq._
implicit def eqTypeA[A] = Eq[A](_ == _)

scala> ((n: Int) => n) === ((n: Int) => n)
res1: Boolean = false
```

Our expectations about equality comparisons are formalized via the laws of identity, symmetry, reflexivity, and transitivity (see also a discussion on page 185 in Section 6.1.6). The **reflexivity** law states that $x = x$ for any x; so the comparison `x === x` should always return `true`. The example shown above violates that law when we choose $x \stackrel{\text{def}}{=} n^{:\text{Int}} \to n$.

Let us perform structural analysis for the `Eq` typeclass, defining $\text{Eq}^A \stackrel{\text{def}}{=} A \times A \to \mathbb{2}$. The results (see Table 8.2 below) will show which types can be usefully compared for equality.

Fixed types All primitive types have `Eq` instances that use type-specific equality comparisons.

Products If A and B have equality comparisons, we can compare pairs of type $A \times B$ by comparing each part of the pair separately:

```
def eqPair[A: Eq, B: Eq] = Eq[(A, B)]{ case ((a1, b1), (a2, b2)) => a1 === a2 && b1 === b2 }
```

It is easy to check that the identity, symmetry, reflexivity, and transitivity laws hold for the new comparison operation if they hold for comparisons of A and B separately.

It is important that the code for this construction assumes that both types A and B have lawful `Eq` instances and performs the comparisons `a1 === a2` and `b1 === b2`. If the above code performed only, say, the comparison `a1 === a2`, the resulting comparison operation would have violated the **identity law** of the equality operation (if $x = y$ then $f(x) = f(y)$ for any function f): we would have pairs such as $a \times b_1$ and $a \times b_2$ that would be "equal" according to this definition, and yet many functions $f^{:A\times B\to C}$ exist such that $f(a \times b_1) \neq f(a \times b_2)$.

Co-products If A and B have equality comparisons, we can compare values of type $A+B$ while ensuring that a value of type $A+\mathbb{0}$ is never equal to a value of type $\mathbb{0}+B$:

```
def eqEither[A: Eq, B: Eq] = Eq[Either[A, B]] {
  case (Left(a1), Left(a2))      => a1 === a2     // Compare a1+0 and a2+0.
  case (Right(b1), Right(b2))    => b1 === b2     // Compare 0+b1 and 0+b2.
  case _                         => false         // a+0 is never equal to 0+b.
}
```

The laws hold for the new operation because the code defines values of type `Either[A, B]` as equal only when the types are both `Left` or both `Right`. If the comparisons of types A and B satisfy the laws separately, the laws for $A+B$ will be satisfied separately for values of type `Left` and of type `Right`.

Defining the comparison operation in any other way (e.g., setting $a_1+\mathbb{0} \neq a_2+\mathbb{0}$ for all $a_1^{:A}$ and $a_2^{:A}$, or $\mathbb{0}+b_1 = \mathbb{0}+b_2$ for all $b_1^{:B}$ and $b_2^{:B}$) would have violated the reflexivity law or the identity law.

Since the `Eq` typeclass has both the product and the co-product constructions, any polynomial type expression made of primitive types will also have an `Eq` typeclass instance.

Functions If A has an `Eq` instance, can we create an `Eq` instance for $R \to A$ where R is some other type? This would be possible with a function of this type:

$$\forall A.\, (A \times A \to \mathbb{2}) \to (R \to A) \times (R \to A) \to \mathbb{2} \quad . \tag{8.6}$$

Here we assume that the type R is an arbitrary chosen type, so no values of type R can be computed from scratch. (This would not be the case when $R = \mathbb{1}$ or $R = \mathbb{2}$, say. But in those cases the type $R \to A$ can be simplified to a polynomial type, e.g., $\mathbb{1} \to A \cong A$ and $\mathbb{2} \to A \cong A \times A$, etc.) Without values of type R, we cannot compute any values of type A and cannot apply comparisons to them. Then the only possible implementations of the type signature (8.6) are constant functions returning `true` or `false`. However, the implementation that always returns `false` will violate the reflexivity law $x = x$. The implementation that always returns `true` is not useful.

We will be able to evaluate functions of type $R \to A$ if some chosen values of type R are available. Examples of that situation are types R of the form $R \cong \mathbb{1} + S$ for some type S; we then always have a chosen value $r_1 \stackrel{\text{def}}{=} 1 + \mathbb{0}^{:S}$ of type R. What if we compare functions only via their values at r_1?

```
def equals[R, A: Eq](f: R => A, g: R => A): Boolean = f(r1) === g(r1)  // Violates the identity law.
```

The above code defines a comparison operation that violates the identity law: there are many functions `f` and `g` that will give different results for arguments not equal to r_1.

Another way to see the problem is to write the type equivalence:

$$R \to A \cong \mathbb{1} + S \to A \cong A \times (S \to A) \quad ,$$

which reduces this case to the product construction we saw above. It follows that we need to have an `Eq` instance for $S \to A$ to define the equality operation for $R \to A$. If we have a chosen value of type S, e.g., if $S \cong \mathbb{1} + T$, we will again reduce the situation to the product construction with the function type $T \to A$. This process will end only if the type R has the form:

$$R \cong \mathbb{1} + \mathbb{1} + \ldots + \mathbb{1} \quad , \qquad R \to A \cong \mathbb{1} + \mathbb{1} + \ldots + \mathbb{1} \to A \cong A \times A \times \ldots \times A \quad ,$$

i.e., if R has a *known* finite number of distinct values. Then we can write code that applies functions of type $R \to A$ to every possible value of the argument of type R and compares all the resulting values of type A. However, we also see that the type $R \to A$ is equivalent to a polynomial type, which is a product of a fixed number of A's. The product construction already covers that type.

So, we can compare functions $f_1^{:R\to A}$ and $f_2^{:R\to A}$ only if we are able to check whether $f_1(r) = f_2(r)$ for *every* possible value of type R. We cannot implement such a comparison for a general type R.

We conclude that functions of type $R \to A$ cannot have an `Eq` instance for a general type R. A similar argument shows that functions of type $A \to R$ also cannot have a useful `Eq` instance. The only exceptions are types $R \to A$ or $A \to R$ that are equivalent to some polynomial types.

Recursive types Since all polynomial type expressions preserve `Eq` instances, the same logic can be applied to recursive polynomial types. For instance, lists and trees with `Eq`-comparable values are also `Eq`-comparable. Let us prove this rigorously.

Consider a recursive polynomial type T defined using a polynomial functor $S^\bullet$:

$$T \stackrel{\text{def}}{=} S^T \quad .$$

The functor $S^\bullet$ may use other fixed types that have `Eq` instances. To construct the typeclass instance for T, we first implement a function eqS of type:

$$\text{eqS} : \text{Eq}^A \to \text{Eq}^{S^A} \quad .$$

This function produces an `Eq` instance for S^A using `Eq` instances of A and of all other types that S^A depends on. The product and co-product constructions guarantee that it is always possible to implement this function for a polynomial functor $S^\bullet$. Then we define an `Eq` instance for T recursively:

$$\text{eqT} : \text{Eq}^T \quad , \qquad \text{eqT} \stackrel{\text{def}}{=} \text{eqS}\,(\text{eqT}) \quad .$$

The recursive equation for `eqT` needs to be implemented as an expanded function:

$$\text{eqT} \stackrel{\text{def}}{=} t^{:T} \times t^{:T} \to \text{eqS}\,(\text{eqT})\,(t \times t) \quad .$$

Then the recursion will terminate, as we have seen in the previous section.

As an example, let us make an `Eq` instance for the type T defined by $T \stackrel{\text{def}}{=} \text{Int} + T + \text{Int} \times T \times T$:

```
type S[A] = Either[Either[Int, A], (Int, (A, A))]
final case class T(s: S[T])                        // Recursive type equation T ≜ Int + T + Int × T × T.
def eqS[A](implicit ti: Eq[A]): Eq[S[A]] = {       // Function of type Eq[A] => Eq[S[A]].
  implicit val e1 = eqEither[Int, A]               // Instance for Int + A.
  implicit val e2 = eqPair[A, A]                   // Instance for A × A.
  implicit val e3 = eqPair[Int, (A, A)]            // Instance for Int × A × A.
  eqEither[Either[Int, A], (Int, (A, A))]          // Instance for Int + A + Int × A × A.
}
implicit def eqT: Eq[T] = Eq { case (T(s1), T(s2)) => eqS(eqT).equal(s1, s2) }
```

To test that the recursion terminates, define a value of type T and run a comparison:

```
val t = T(Left(Right(T(Left(Left(10))))))   // t : 0 + (10^:Int + 0 + 0)^:T + 0

scala> t === t
res0: Boolean = true
```

Checking laws for recursive instances We use induction to verify that the laws of identity, symmetry, reflexivity, and transitivity will hold for the `Eq` instance just defined. The `Eq` instance for T was defined as a recursive function `eqT = eqS(eqT)`. We know that `eqS(x)` satisfies all the required laws if `x` satisfies them ("`eqS` preserves the laws"). Let us visualize what happens when the function `eqT` is applied to some values of type T. Are the laws satisfied in that case (e.g., is `eqT(t, t) == true` for all `t:T`)? The function `eqT` will call itself recursively whenever the body of `eqS` calls `eqT`. *Assuming* that all the recursive calls to `eqT` satisfy the laws, we find that the outer call to `eqT` also satisfies the laws because `eqS` preserves them. This corresponds to the inductive step.

Eventually the recursion will terminate; so *some* calls of `eqT(t1, t2)` with certain values `t1` and `t2` do not cause any more recursive self-calls to `eqT`. For *those* values `t1` and `t2`, the function `eqT` will satisfy the required laws with no additional assumptions, because `eqS(eqT)(t1, t2)`, which satisfies the laws, does not call `eqT`. This corresponds to the base case of an inductive proof.

So, the recursively defined `Eq` instance is lawful. This argument is general and works in all cases when a typeclass instance t is defined by recursion, $t \stackrel{\text{def}}{=} s(t)$, via a function s that preserves the laws.

Construction	Type signature to implement
The `Unit` type, or other primitive type	$\text{Eq}^T \stackrel{\text{def}}{=} T \times T \to \mathbb{2}$
Product of `Eq` types A and B	$\text{Eq}^A \times \text{Eq}^B \to \text{Eq}^{A\times B}$
Co-product of `Eq` types A and B	$\text{Eq}^A \times \text{Eq}^B \to \text{Eq}^{A+B}$
Recursive types	$\text{Eq}^A \to \text{Eq}^{S^A}$ where $T \stackrel{\text{def}}{=} S^T$

Table 8.2: Type constructions producing the `Eq` typeclass.

Summary Instances of the `Eq` typeclass can be derived for any polynomial or recursive polynomial type expressions containing primitive types or type parameters constrained to be `Eq`-comparable. The derivation of the `Eq` instance is unambiguous and can be automated with libraries such as the `kittens`[2], `magnolia`[3] , or `scalaz-deriving`[4].

8.3.3 Semigroups

A type T has an instance of `Semigroup` when an associative binary operation of type $T \times T \to T$ is available. We will now apply structural analysis to this typeclass. The results are shown in Table 8.3.

Fixed types Each of the primitive types (`Boolean`, `Int`, `Double`, `String`, etc.) has at least one well-known associative binary operation that can be used to define a semigroup instance. Booleans have the conjunction and the disjunction operations; numbers can be added or multiplied, or the maximum or the minimum number chosen; strings can be concatenated or chosen in the alphabetical order. Examples 8.2.5.4 and 8.2.5.5 show several implementations of such binary operations. The `Unit` type has a trivially defined binary operation, which is also associative (since it always returns the same value). The same is true for any fixed type that has a chosen "default" value: the binary operation that always returns the default value is associative (although not likely to be useful).

Type parameters A semigroup instance parametric in type T means a value of type $\forall T. T \times T \to T$. There are two implementations of this type signature: $a^{:T} \times b^{:T} \to a$ and $a^{:T} \times b^{:T} \to b$. Both provide an associative binary operation, as Example 8.2.5.5(a) shows. So, any type T can be made into a "trivial" semigroup in one of these two ways. ("Trivial" semigroups are occasionally useful.)

Products If types A and B are semigroups, the product $A \times B$ can be also given a `Semigroup` instance.

To compute that instance means, in the general case, to implement a function with type:

$$\text{semigroupPair} : \forall (A, B). \text{Semigroup}^A \times \text{Semigroup}^B \to \text{Semigroup}^{A\times B} \quad .$$

Writing out the type expressions, we get the type signature:

$$\text{semigroupPair} : \forall (A, B).\ (A \times A \to A) \times (B \times B \to B) \to (A \times B \times A \times B \to A \times B) \quad .$$

While this type signature can be implemented in a number of ways, we look for code that preserves information, in hopes of satisfying the associativity law. The code should be a function of the form:

$$\text{semigroupPair} \stackrel{\text{def}}{=} f^{:A\times A\to A} \times g^{:B\times B\to B} \to a_1^{:A} \times b_1^{:B} \times a_2^{:A} \times b_2^{:B} \to ???^{:A} \times ???^{:B} \quad .$$

Since we are trying to define the new semigroup operation through the previously given operations f and g, it is natural to apply f and g to the given data a_1, a_2, b_1, b_2 and write:

$$f^{:A\times A\to A} \times g^{:B\times B\to B} \to a_1^{:A} \times b_1^{:B} \times a_2^{:A} \times b_2^{:B} \to f(a_1, a_2) \times g(b_1, b_2) \quad .$$

[2] https://github.com/typelevel/kittens
[3] https://github.com/propensive/magnolia
[4] https://github.com/scalaz/scalaz-deriving

This code defines a new binary operation $\oplus_{A\times B}$ via the previously given $\oplus_A$ and $\oplus_B$ as:

$$(a_1^{:A} \times b_1^{:B}) \oplus_{A\times B} (a_2^{:A} \times b_2^{:B}) = (a_1 \oplus_A a_2) \times (b_1 \oplus_B b_2) \quad . \tag{8.7}$$

```
def semigroupPair[A: Semigroup, B: Semigroup] =
  Semigroup[(A, B)]{ case ((a1, b1),(a2, b2)) => (a1 |+| a2, b1 |+| b2) }
```

This implementation satisfies the associativity law if the operations $\oplus_A$, $\oplus_B$ already do, i.e., if the results of computing $a_1 \oplus_A a_2 \oplus_A a_3$ and $b_1 \oplus_B b_2 \oplus_B b_3$ do not depend on the order of parentheses:

$$\begin{aligned}
&((a_1 \times b_1) \oplus_{A\times B} (a_2 \times b_2)) \oplus_{A\times B} (a_3 \times b_3) = ((a_1 \oplus_A a_2) \times (b_1 \oplus_B b_2)) \oplus_{A\times B} (a_3 \times b_3) \\
&\quad = (a_1 \oplus_A a_2 \oplus_A a_3) \times (b_1 \oplus_B b_2 \oplus_B b_3) \quad , \\
&(a_1 \times b_1) \oplus_{A\times B} ((a_2 \times b_2) \oplus_{A\times B} (a_3 \times b_3)) = (a_1 \times b_1) \oplus_{A\times B} ((a_2 \oplus_A a_3) \times (b_2 \oplus_B b_3)) \\
&\quad = (a_1 \oplus_A a_2 \oplus_A a_3) \times (b_1 \oplus_B b_2 \oplus_B b_3) \quad .
\end{aligned}$$

Co-products To compute a `Semigroup` instance for the co-product $A + B$ of two semigroups, we need:

$$\text{semigroupEither} : \forall(A, B).\, \text{Semigroup}^A \times \text{Semigroup}^B \to \text{Semigroup}^{A+B} \quad .$$

Writing out the type expressions, we get the type signature:

$$\text{semigroupEither} : \forall(A, B).\ (A \times A \to A) \times (B \times B \to B) \to (A + B) \times (A + B) \to A + B \quad .$$

Begin by writing a function with a typed hole:

$$\text{semigroupEither} \stackrel{\text{def}}{=} f^{:A\times A\to A} \times g^{:B\times B\to B} \to c^{:(A+B)\times(A+B)} \to ???^{:A+B} \quad .$$

Transforming the type expression $(A + B) \times (A + B)$ into an equivalent disjunctive type:

$$(A + B) \times (A + B) \cong A \times A + A \times B + B \times A + B \times B \quad ,$$

we can continue to write the function's code in matrix notation,

$$f^{:A\times A\to A} \times g^{:B\times B\to B} \to \left\| \begin{array}{c|cc} & A & B \\ \hline A \times A & ???^{:A\times A\to A} & ???^{:A\times A\to B} \\ A \times B & ???^{:A\times B\to A} & ???^{:A\times B\to B} \\ B \times A & ???^{:B\times A\to A} & ???^{:B\times A\to B} \\ B \times B & ???^{:B\times B\to A} & ???^{:B\times B\to B} \end{array} \right| \quad .$$

The matrix is 4×2 because the input type, $A \times A + A \times B + B \times A + B \times B$, is a disjunction with 4 parts, while the result type $A + B$ is a disjunction with 2 parts. In each row, we need to fill only one of the two typed holes because only one part of the disjunction $A + B$ can have a value.

To save space, we will omit the types in the matrices. The first and the last rows of the matrix must contain functions of types $A \times A \to A$ and $B \times B \to B$, and so it is natural to fill them with f and g:

$$f^{:A\times A\to A} \times g^{:B\times B\to B} \to \left\| \begin{array}{cc} f^{:A\times A\to A} & \mathbb{0} \\ ???^{:A\times B\to A} & ???^{:A\times B\to B} \\ ???^{:B\times A\to A} & ???^{:B\times A\to B} \\ \mathbb{0} & g^{:B\times B\to B} \end{array} \right| \quad .$$

The remaining two rows can be filled in four different ways:

$$f \times g \to \left\| \begin{array}{cc} f^{:A\times A\to A} & \mathbb{0} \\ a^{:A} \times b^{:B} \to a & \mathbb{0} \\ b^{:B} \times a^{:A} \to a & \mathbb{0} \\ \mathbb{0} & g^{:B\times B\to B} \end{array} \right| , \quad f \times g \to \left\| \begin{array}{cc} f & \mathbb{0} \\ \mathbb{0} & a^{:A} \times b^{:B} \to b \\ \mathbb{0} & b^{:B} \times a^{:A} \to b \\ \mathbb{0} & g \end{array} \right| ,$$

$$f \times g \to \left\| \begin{array}{cc} f^{:A\times A\to A} & \mathbb{0} \\ \mathbb{0} & a^{:A} \times b^{:B} \to b \\ b^{:B} \times a^{:A} \to a & \mathbb{0} \\ \mathbb{0} & g^{:B\times B\to B} \end{array} \right| , \quad f \times g \to \left\| \begin{array}{cc} f & \mathbb{0} \\ a^{:A} \times b^{:B} \to a & \mathbb{0} \\ \mathbb{0} & b^{:B} \times a^{:A} \to b \\ \mathbb{0} & g \end{array} \right| .$$

The Scala code corresponding to the four possible definitions of $\oplus$ is:

```
def semigroupEither1[A: Semigroup, B: Semigroup] = Semigroup[Either[A, B]] {
  case (Left(a1), Left(a2))     =>  a1 |+| a2 // Here a1 |+| a2 is a1 ⊕_A a2.
  case (Right(b1), Right(b2))   =>  b1 |+| b2 // Here b1 |+| b2 is b1 ⊕_B b2.
  case (Left(a), Right(b))      =>  a         // "Take A" - discard all data of type B.
  case (Right(b), Left(a))      =>  a
}
def semigroupEither2[A: Semigroup, B: Semigroup] = Semigroup[Either[A, B]] {
  case (Left(a1), Left(a2))     =>  a1 |+| a2
  case (Right(b1), Right(b2))   =>  b1 |+| b2
  case (Left(a), Right(b))      =>  b         // "Take B" - discard all data of type A.
  case (Right(b), Left(a))      =>  b
}
def semigroupEither3[A: Semigroup, B: Semigroup] = Semigroup[Either[A, B]] {
  case (Left(a1), Left(a2))     =>  a1 |+| a2
  case (Right(b1), Right(b2))   =>  b1 |+| b2
  case (Left(a), Right(b))      =>  a         // "Take first" - discard y in x ⊕ y.
  case (Right(b), Left(a))      =>  b
}
def semigroupEither4[A: Semigroup, B: Semigroup] = Semigroup[Either[A, B]] {
  case (Left(a1), Left(a2))     =>  a1 |+| a2
  case (Right(b1), Right(b2))   =>  b1 |+| b2
  case (Left(a), Right(b))      =>  b         // "Take last" - discard x in x ⊕ y.
  case (Right(b), Left(a))      =>  a
}
```

The four different choices of the binary operation can be described as:

1. ("Take A") Discard all data of type B: $(a^{:A} + \mathbb{0}) \oplus (\mathbb{0} + b^{:B}) = a$ and $(\mathbb{0} + b^{:B}) \oplus (a^{:A} + \mathbb{0}) = a$.
2. ("Take B") Discard all data of type A: $(a^{:A} + \mathbb{0}) \oplus (\mathbb{0} + b^{:B}) = b$ and $(\mathbb{0} + b^{:B}) \oplus (a^{:A} + \mathbb{0}) = b$.
3. ("Take first") $x \oplus y$ discards y: $(a^{:A} + \mathbb{0}) \oplus (\mathbb{0} + b^{:B}) = a$ and $(\mathbb{0} + b^{:B}) \oplus (a^{:A} + \mathbb{0}) = b$.
4. ("Take last") $x \oplus y$ discards x: $(a^{:A} + \mathbb{0}) \oplus (\mathbb{0} + b^{:B}) = b$ and $(\mathbb{0} + b^{:B}) \oplus (a^{:A} + \mathbb{0}) = a$.

Does the semigroup law (8.1) hold for the new typeclass instance with any of these implementations?

It turns out that *all four* implementations are lawful. To verify the associativity law, we need to show that values such as `Left(x) |+| Left(y) |+| Right(z)` or `Left(x) |+| Right(y) |+| Right(z)` do not depend on the order of inserted parentheses. Expressions of the form $x \oplus y \oplus z$ can have 8 possible combinations of `Left` and `Right` types. Each of them needs to be checked against each of the 4 implementations of $\oplus$. Instead of doing 32 separate derivations, we will reason case by case.

First consider the case when all three values are of type `Left(x)`. In all four implementations, the binary operation reduces to the binary operation of the semigroup A, which is associative since we assume that A is a lawful semigroup:

```
Left(x) |+| Left(y) |+| Left(z) == Left(x |+| y |+| z) == Left(x) |+| ( Left(y) |+| Left(z) )
```

The same argument applies to three values of type `Right`. It remains to consider the "mixed" cases.

In the first implementation ("take A"), we discard all data of type `Right` from the expression $x \oplus y \oplus z$, keeping only data of type `Left`. It is clear that discarding data of type `Right` will yield the same result regardless of the order of parentheses. If more than one item of type `Left` remains, the data is aggregated with the operation $\oplus_A$. So, the results do not depend on the order of parentheses:

```
Right(x) |+| Right(y) |+| Left(z) == Left(z) == Right(x) |+| ( Right(y) |+| Left(z) )
Left(x) |+| Right(y) |+| Left(z) == Left(x |+| z) == Left(x) |+| ( Right(y) |+| Left(z) )
```

A similar argument shows that the second implementation ("take B") is also associative.

The implementation "take first" will select the first value whenever the types are mixed. We get:

```
Left(x) |+| Left(y) |+| Right(z) == Left(x |+| y) == Left(x) |+| ( Left(y) |+| Right(z) )
Left(x) |+| Right(y) |+| p == Left(x) == Left(x) |+| ( Right(y) |+| p )    // Regardless of p.
Right(x) |+| Left(y) |+| p == Right(x) == Right(x) |+| ( Left(y) |+| p )   // Regardless of p.
Right(x) |+| Right(y) |+| Left(z) == Right(x |+| y) == Right(x) |+| ( Right(y) |+| Left(z) )
```

The results are independent of the parentheses. The same applies to the "take last" implementation.

Functions If A is a semigroup and E is any fixed type, are the types $A \to E$ and/or $E \to A$ semigroups? To create a `Semigroup` instance for $E \to A$ means to implement the type signature:

$$\text{Semigroup}^A \to \text{Semigroup}^{E\to A} = (A \times A \to A) \to (E \to A) \times (E \to A) \to E \to A \quad .$$

An implementation that preserves information is:

$$\text{semigroupFunc} \stackrel{\text{def}}{=} f^{:A\times A\to A} \to g_1^{:E\to A} \times g_2^{:E\to A} \to e^{:E} \to f(g_1(e), g_2(e)) \quad .$$

This defines the new $\oplus$ operation by $g_1 \oplus g_2 \stackrel{\text{def}}{=} e \to g_1(e) \oplus_A g_2(e)$.

```
def semigroupFunc[E, A: Semigroup] = Semigroup[E => A] { case (g1, g2) => e => g1(e) |+| g2(e) }
```

In the pipe notation, $e \triangleright (f \oplus g) = f(e) \oplus_A g(e)$. The associativity law holds for this operation:

$$\begin{aligned} &\text{left-hand side}: \quad e \triangleright ((f \oplus g) \oplus h) = (e \triangleright (f \oplus g)) \oplus_A h(e) = f(e) \oplus_A g(e) \oplus_A h(e) \quad . \\ &\text{right-hand side}: \quad e \triangleright (f \oplus (g \oplus h)) = f(e) \oplus_A (e \triangleright (g \oplus h)) = f(e) \oplus_A g(e) \oplus_A h(e) \quad . \end{aligned}$$

The type $A \to E$ only allows semigroup operations that discard the left or the right element: the type signature $f^{:A\times A\to A} \to h_1^{:A\to E} \times h_2^{:A\to E} \to ???^{:A\to E}$ can be implemented only by discarding f and one of h_1 or h_2. Either choice makes $A \to E$ into a trivial semigroup.

We have seen constructions that create new semigroups via products, co-products, and functions. Thus, any exponential-polynomial type expression built up from primitive types and/or existing semigroups is again a semigroup.

Recursive types A type T defined by a recursive type equation $T \stackrel{\text{def}}{=} S^T$ can have a semigroup instance when S^A is any exponential-polynomial type expression built up from primitive types, products, co-products, and the type parameter A. The known semigroup constructions guarantee that a typeclass instance `Semigroup[S[A]]` can be created out of `Semigroup[A]`. This gives us a function:

$$\text{semigroupS} : \text{Semigroup}^A \to \text{Semigroup}^{S^A} \quad .$$

Then a semigroup instance for T is defined recursively as:

$$\text{semigroupT} \stackrel{\text{def}}{=} \text{semigroupS}\,(\text{semigroupT}) \quad .$$

The recursive definition will terminate as long as we implement it in code as an expanded function. The associativity law holds for the semigroup T by induction, as shown at the end of Section 8.3.2.

Summary Any type whatsoever can have a `Semigroup` instance. Since the instances have several inequivalent implementations, automatic derivation of `Semigroup` instances is not often useful.

Construction	Type signature to implement
The `Unit` type, or other fixed type C	Semigroup^C
Product of semigroups A and B	$\text{Semigroup}^A \times \text{Semigroup}^B \to \text{Semigroup}^{A\times B}$
Co-product of semigroups A and B	$\text{Semigroup}^A \times \text{Semigroup}^B \to \text{Semigroup}^{A+B}$
Function from another type E	$\text{Semigroup}^A \to \text{Semigroup}^{E\to A}$
Recursive type	$\text{Semigroup}^A \to \text{Semigroup}^{S^A}$ where $T \stackrel{\text{def}}{=} S^T$

Table 8.3: Type constructions producing the `Semigroup` typeclass.

8.3.4 Monoids

Since a monoid is a semigroup with a default value, a `Monoid` instance is a value of type:

$$\text{Monoid}^A \stackrel{\text{def}}{=} (A \times A \to A) \times A \quad .$$

For the binary operation $A \times A \to A$, we can re-use the results of structural analysis for semigroups. Additionally, we will need to verify that the default value satisfies monoid's identity laws. The results are shown in Table 8.4.

Fixed types Each of the primitive types (`Boolean`, `Int`, `Double`, `String`, etc.) has a well-defined monoidal operation (addition or multiplication for numbers, concatenation for strings, and so on). The unit type, $\mathbb{1}$, also has a monoid instance where all methods return the unit value 1.

Type parameters This construction works for semigroups but *not* for monoids: the "trivial" semigroup operations $x \oplus y = x$ and $x \oplus y = y$ are not compatible with monoid's identity laws. (e.g., with the definition $x \oplus y = x$, no default value e could possibly satisfy the left identity law $e \oplus y = y$ because $e \oplus y = e$ for all y).

Products For two monoids A and B, a monoid instance for the product $A \times B$ is computed by:

$$\text{monoidPair} : \forall (A, B).\, \text{Monoid}^A \times \text{Monoid}^B \to \text{Monoid}^{A\times B} \quad .$$

The empty value for the monoid $A \times B$ is $e_{A\times B} \stackrel{\text{def}}{=} e_A \times e_B$, the pair of empty values from the monoids A and B. The new binary operation is defined by Eq. (8.7) as in the pair semigroup construction. We can now verify the new monoid's identity laws, assuming that they hold for the monoids A and B:

$$(a_1 \times b_1) \oplus (e_A \times e_B) = (a_1 \oplus_A e_A) \times (b_1 \oplus_B e_B) = a_1 \times b_1 \quad ,$$
$$(e_A \times e_B) \oplus (a_2 \times b_2) = (e_A \oplus_A a_2) \times (e_B \oplus_B b_2) = a_2 \times b_2 \quad .$$

An implementation in Scala is:

```
def monoidPair[A: Monoid, B: Monoid]: Monoid[(A, B)] = Monoid[(A, B)](
  { case ((a1, b1),(a2, b2)) => (a1 |+| a2, b1 |+| b2) },
  ( implicitly[Monoid[A]].empty, implicitly[Monoid[B]].empty )
)
```

Co-products For two monoids A and B, how can we implement a `Monoid` instance for $A + B$? We have seen four versions of the semigroup operation $\oplus$ for the type $A + B$. Independently of those, we need to define the empty element e_{A+B}, which must have type $A + B$. There are two possibilities:

$$e_{A+B} \stackrel{\text{def}}{=} e_A + \mathbb{0}^{:B} \qquad \text{or} \qquad e_{A+B} \stackrel{\text{def}}{=} \mathbb{0}^{:A} + e_B \quad .$$

It remains to see which of the eight combinations will satisfy the monoid identity laws:

$$(a + \mathbb{0}) \oplus e_{A+B} = a + \mathbb{0} \quad , \qquad e_{A+B} \oplus (a + \mathbb{0}) = a + \mathbb{0} \quad ,$$
$$(\mathbb{0} + b) \oplus e_{A+B} = \mathbb{0} + b \quad , \qquad e_{A+B} \oplus (\mathbb{0} + b) = \mathbb{0} + b \quad .$$

First choose $e_{A+B} = e_A + \mathbb{0}$; the reasoning for the other case will be quite similar. The first line above:

$$(a + \mathbb{0}) \oplus (e_A + \mathbb{0}) = a + \mathbb{0} \quad , \qquad (e_A + \mathbb{0}) \oplus (a + \mathbb{0}) = a + \mathbb{0} \quad ,$$

will hold because all four versions of the operation $\oplus$ will reduce to $\oplus_A$ on values of type $A + \mathbb{0}$. The second line, however, is compatible only with one version of the $\oplus$ operation, namely with "take B":

$$(a^{:A} + \mathbb{0}) \oplus (\mathbb{0} + b^{:B}) = b \qquad \text{and} \qquad (\mathbb{0} + b^{:B}) \oplus (a^{:A} + \mathbb{0}) = b \quad .$$

So, the co-product construction must choose one of the monoids, say B, as "preferred". The code is:

```
def monoidEitherPreferB[A: Monoid, B: Monoid] = Monoid[Either[A, B]]( {
  case (Left(a1), Left(a2))     => Left(a1 |+| a2)
  case (Left(a), Right(b))      => Right(b) // "Take B".
  case (Right(b), Left(a))      => Right(b)
  case (Right(b1), Right(b2))   => Right(b1 |+| b2)
}, Left(implicitly[Monoid[A]].empty) )
```

Similarly, the choice $e_{A+B} \stackrel{\text{def}}{=} \mathbb{0} + e_B$ forces us to choose the version "take A" of the $\oplus$ operation.

Functions The semigroup construction for function types works also for monoids. Exercise 8.4.2.3 will show that the function type $R \to A$ is a lawful monoid for any type R and any monoid A.

Additionally, the function type $R \to R$ is a monoid for any type R (even if R is not a monoid). The operation $\oplus$ and the empty value are defined as $f^{:R\to R} \oplus g^{:R\to R} \stackrel{\text{def}}{=} f \fatsemi g$ and $e_{R\to R} \stackrel{\text{def}}{=} \text{id}^R$. The code is:

```
def monoidFunc1[R]: Monoid[R => R] = Monoid( (f, g) => f andThen g, identity )
```

The monoidal operation $\oplus$ is the forward function composition $f \fatsemi g$, so the monoid laws for this operation are the identity and the associativity laws of function composition (see Section 4.2.2). We could also use the backward function composition $(f \circ g)$ to define $f \oplus g$.

Recursive types Can we define a `Monoid` instance for a type T defined by $T \stackrel{\text{def}}{=} S^T$, where $S^\bullet$ is some type constructor? As we have seen, products, co-products, and function type constructions preserve monoids. For any type built up via these constructions from monoids, a `Monoid` instance can be derived. These constructions cover all exponential-polynomial types. So, let us consider an exponential-polynomial type constructor S^A that contains a type parameter A, primitive types, and other known monoid types. For such type constructors $S^\bullet$, we will always be able to implement a function `monoidS` that derives a `Monoid` instance for S^A from a monoid instance for A:

$$\text{monoidS} : \text{Monoid}^A \to \text{Monoid}^{S^A} \quad .$$

A monoid instance for T is then defined recursively by:

$$\text{monoidT} \stackrel{\text{def}}{=} \text{monoidS}\,(\text{monoidT}) \quad .$$

As we saw before, the code for this definition will terminate only if we implement it as a recursive function. However, the type Monoid^A is not a function type: it is a pair $(A \times A \to A) \times A$. To obtain a working implementation of `monoidT`, we need to rewrite that type into an equivalent function type,

$$\text{Monoid}^A = (A \times A \to A) \times A \cong (A \times A \to A) \times (\mathbb{1} \to A) \cong (\mathbb{1} + A \times A \to A) \quad ,$$

where we used the known type equivalences $A \cong \mathbb{1} \to A$ and $(A \to C) \times (B \to C) \cong A + B \to C$.

```
final case class Monoid[A](methods: Option[(A, A)] => A)
```

With this new definition of the `Monoid` typeclass (and with the appropriate changes to the code of `monoidPair`, `monoidEitherPreferB`, and `monoidFunc`), we can now implement the recursive construction.

To illustrate how that works, consider the exponential-polynomial type constructor $S^\bullet$ defined as:

$$S^A \stackrel{\text{def}}{=} (\text{Int} + A) \times \text{Int} + \text{String} \times (A \to (A \to \text{Int}) \to A) \quad .$$

Construction	Type signature to implement
The `Unit` type, or primitive types	$\text{Monoid}^{\mathbb{1}}$, $\text{Monoid}^{\text{Int}}$, etc.
Product of monoids A and B	$\text{Monoid}^A \times \text{Monoid}^B \to \text{Monoid}^{A\times B}$
Co-product of monoid A and semigroup B	$\text{Monoid}^A \times \text{Semigroup}^B \to \text{Monoid}^{A+B}$
Function from another type E	$\text{Monoid}^A \to \text{Monoid}^{E\to A}$
Recursive type	$\text{Monoid}^A \to \text{Monoid}^{S^A}$ where $T \stackrel{\text{def}}{=} S^T$

Table 8.4: Type constructions producing the `Monoid` typeclass.

```
type S[A] = Either[(Either[Int, A], Int), (String, A => (A => Int) => A)]
```

It is clear that S^A is built up from type constructions that preserve monoids at each step. So, we expect that the recursive type $T \stackrel{\text{def}}{=} S^T$ is a monoid. We first implement the function `monoidS`:

```
def monoidS[A](implicit ti: Monoid[A]): Monoid[S[A]] = {
  implicit val m0 = monoidEitherPreferB[Int, A]
  implicit val m1 = monoidPair[Either[Int, A], Int]
  implicit val m2 = monoidFunc[A, A => Int]
  implicit val m3 = monoidFunc[(A => Int) => A, A]
  implicit val m4 = monoidPair[String, A => (A => Int) => A]
  monoidEitherPreferB[(Either[Int, A], Int), (String, A => (A => Int) => A)]
}
```

We can now use this function to define the recursive type T and a `Monoid` instance for it:

```
final case class T(s: S[T])
def monoidT: Monoid[T] = Monoid[T] {
  case None              => T(monoidS[T](monoidT).methods(None))
  case Some((t1, t2))    => T(monoidS[T](monoidT).methods(Some(t1.s, t2.s)))
}
```

To test this code, create a value of type T and perform a computation:

```
val t = T(Right(("a", t => f => T(Left((Left(f(t)), 10))))))

scala> t |+| t
res0: T = T(Right((aa,<function1>)))
```

Another way of implementing the recursive construction is to write the `Monoid` typeclass using a `trait`. Although the code is longer, it is easier to read. The recursive instance is implemented by:

```
def monoidT: Monoid[T] = new Monoid[T] {
  def empty: T = T(monoidS[T](monoidT).empty)   // This must be a ‘def empty‘, not a ‘val empty‘.
  def combine: (T, T) => T = (x, y) => T(monoidS[T](monoidT).combine(x.s, y.s))
}
```

The recursive definition of `monoidT` terminates because the methods of the `trait` are declared as `def` (not as `val`). The code of `monoidS` remains the same; we need to rewrite `monoidPair`, `monoidEitherPreferB`, and `monoidFunc` to accommodate the new definition of the `Monoid` typeclass. The full code is shown in Figure 8.1.

The monoid laws hold for the recursive instances by induction (see Section 8.3.2).

Summary A `Monoid` instance can be implemented, in at least one way, for *any* exponential-polynomial type expression (including recursive types) built from primitive types and other monoids.

```
trait Monoid[T] {
  def empty: T
  def combine: (T, T) => T
}
implicit val monoidInt: Monoid[Int] = new Monoid[Int] {
  def empty: Int = 0
  def combine: (Int, Int) => Int = _ + _
}
implicit val monoidString: Monoid[String] = new Monoid[String] {
  def empty: String = ""
  def combine: (String, String) => String = _ + _
}
implicit class MonoidOps[T: Monoid](t: T) {
  def |+|(a: T): T = implicitly[Monoid[T]].combine(t, a)
}
def monoidPair[A: Monoid, B: Monoid]: Monoid[(A, B)] = new Monoid[(A, B)] {
  def empty: (A, B) = (implicitly[Monoid[A]].empty, implicitly[Monoid[B]].empty)
  def combine: ((A, B), (A, B)) => (A, B) = {
    case ((a1, b1), (a2, b2)) => (a1 |+| a2, b1 |+| b2)
  }
}
def monoidEitherPreferB[A: Monoid, B: Monoid] = new Monoid[Either[A, B]] {
  def empty: Either[A, B] = Left(implicitly[Monoid[A]].empty)
  def combine: (Either[A, B], Either[A, B]) => Either[A, B] = {
    case (Left(a1), Left(a2))     => Left(a1 |+| a2)
    case (Left(a), Right(b))      => Right(b) // "Take B".
    case (Right(b), Left(a))      => Right(b)
    case (Right(b1), Right(b2))   => Right(b1 |+| b2)
  }
}
def monoidFunc[A: Monoid, E] = new Monoid[E => A] {
  def empty: E => A = _ => implicitly[Monoid[A]].empty
  def combine: (E => A, E => A) => E => A = {
    case (f, g) => e => f(e) |+| g(e)
  }
}
// This type constructor will be used below to define a recursive type T.
type S[A] = Either[(Either[Int, A], Int), (String, A => (A => Int) => A)]

// If we have a Monoid instance for A, we can get a Monoid instance for S[A].
def monoidS[A](implicit ti: Monoid[A]): Monoid[S[A]] = {
  implicit val m0 = monoidEitherPreferB[Int, A]
  implicit val m1 = monoidPair[Either[Int, A], Int]
  implicit val m2 = monoidFunc[A, A => Int]
  implicit val m3 = monoidFunc[(A => Int) => A, A]
  implicit val m4 = monoidPair[String, A => (A => Int) => A]
  monoidEitherPreferB[(Either[Int, A], Int), (String, A => (A => Int) => A)]
}
// Define a recursive type T and a Monoid instance for it.
final case class T(s: S[T])
implicit def monoidT: Monoid[T] = new Monoid[T] {
  def empty: T = T(monoidS[T](monoidT).empty) // Here, ‘val empty‘ will cause a StackOverflowError.
  def combine: (T, T) => T = (x, y) => T(monoidS[T](monoidT).combine(x.s, y.s))
}
val t = T(Right(("abc", t => f => T(Left((Left(f(t)), 10))))))
val e = implicitly[Monoid[T]].empty

scala> t |+| t |+| e   // Expect to see the string "abcabc".
res0: T = T(Right((abcabc,<function1>)))
```

Figure 8.1: Implementing a recursive instance of the `Monoid` typeclass via a `trait`.

8.3.5 Pointed functors: motivation and laws

Section 8.2.6 showed how to implement typeclasses for type *constructors*, e.g., the `Functor` typeclass. Typeclass instances in such cases often contain a nested type quantifier such as $\forall A.\,(...)$, so the implementation needs to use Scala's `trait` with `def` methods inside. We will now look at some examples of typeclasses that add further methods to `Functor`. Chapter 6 performed a structural analysis of functors, which we will extend to the new typeclasses.

The first typeclass is a "pointed" functor. A functor type F^T represents, in a generalized sense, "wrapped" values of type T. A frequently used operation is to create a "wrapped" value of type F^T out of a single given value of type T. This operation, usually called `pure` in Scala libraries, is implemented as a function with a type signature:

```
def pure[A]: A => F[A]
```

The code notation for this function is pu_F, and the type signature is written as $\text{pu}_F : \forall A.\, A \to F^A$.

Some examples of pointed functors in Scala are `Option`, `List`, `Try`, and `Future`. Each of these type constructors has a method that "wraps" a given single value:

```
val x: Option[Int] = Some(10)                    // A non-empty option that holds a value.
val y: List[String] = List("abc")                // A list that holds a single value.
val z: Try[Int] = Success(200)                   // A value computed without errors.
val f: Future[String] = Future.successful("OK")  // A ‘Future‘ value that is already available.
```

As we can see, "wrapping a single value" means a different thing for each of the type constructors. Although the relevant methods of these type constructors are not called "`pure`", we can create a PTVF `pure[F]` that would be defined only for `F` that can "wrap" a single value. Such type constructors `F` are called "**pointed**". We may define the typeclass `Pointed` via this code:

```
trait Pointed[F[_]] { def pure[A]: A => F[A] }
```

Now we can implement instances of the `Pointed` typeclass for some functors:

```
implicit val pointedOption = new Pointed[Option] { def pure[A]: A => Option[A] = x => Some(x)    }
implicit val pointedList   = new Pointed[List]   { def pure[A]: A => List[A]   = x => List(x)    }
implicit val pointedTry    = new Pointed[Try]    { def pure[A]: A => Try[A]    = x => Success(x) }
```

The PTVF `pure` can be defined and used like this:

```
def pure[F[_]: Pointed, A](x: A): F[A] =
  implicitly[Pointed[F]].pure(x)

scala> pure[Option, Int](123)
res0: Option[Int] = Some(123)
```

When a pointed type constructor F is a functor, we may use both the functor's `map` method and the `pure` method. Do these two methods need to be compatible in some way? If we "wrap" a value 123 in a `List` and then apply `.map(x => x + 1)`, we expect to obtain a list containing 124; any other result would break our intuition about "wrapping". We can generalize this situation to an arbitrary value $x^{:A}$ wrapped using `pure` and a function $f^{:A\to B}$ applied to the wrapped value via `map`:

```
pure(x).map(f)     //    pu_F(x) ▷ f^↑F
```

We expect that the result should be the same as a wrapped $f(x)$. This expectation can be formulated as a law, called the **naturality law** of `pure`; it must hold for any $f^{:A\to B}$:

```
pure(x).map(f) == pure(f(x))
```

$$\text{pu}_F(x) \triangleright f^{\uparrow F} = \text{pu}_F(f(x)) \quad .$$

In the $\triangleright$-notation, this law is $x \triangleright \text{pu}_F \triangleright f^{\uparrow F} = x \triangleright f \triangleright \text{pu}_F$ or equivalently $x \triangleright \text{pu}_F \,⨾\, f^{\uparrow F} = x \triangleright f \,⨾\, \text{pu}_F$. Since both sides of the law are functions applied to an arbitrary value $x^{:A}$, we can omit x and write:

$$\begin{tikzcd} A \arrow[r, "\text{pu}_F"] \arrow[d, "f"'] & F^A \arrow[d, "f^{\uparrow F}"] \\ B \arrow[r, "\text{pu}_F"] & F^B \end{tikzcd}$$

$$\text{pu}_F \,⨾\, f^{\uparrow F} = f \,⨾\, \text{pu}_F \quad . \tag{8.8}$$

This motivates the following definition: A functor $F^\bullet$ is **pointed** if there exists a fully parametric function $\text{pu}_F : \forall A.\, A \to F^A$ satisfying the naturality law (8.8) for any function $f^{:A\to B}$.

It turns out that we can avoid checking the naturality law of pointed functors if we use a trick: reduce `pure` to a simpler but equivalent form for which the law is satisfied automatically.

Both sides of the naturality law (8.8) are functions of type $A \to F^B$. The trick is to choose $A = \mathbb{1}$ (the `Unit` type) and $f^{:\mathbb{1}\to B} \stackrel{\text{def}}{=} (_ \to b)$, a constant function returning some fixed value $b^{:B}$. Both sides of the naturality law may then be applied to the unit value 1 and must evaluate to the same result:

$$1 \triangleright \text{pu}_F \triangleright (_ \to b)^{\uparrow F} = 1 \triangleright f \triangleright \text{pu}_F \quad .$$

Since $1 \triangleright f = f(1) = b$, we find:

$$\text{pu}_F(1) \triangleright (_ \to b)^{\uparrow F} = \text{pu}_F(b) \quad . \tag{8.9}$$

The naturality law (8.8) applies to all types A, B and to any function $f^{:A\to B}$. Thus, Eq. (8.9) must apply to an arbitrary value $b^{:B}$ for any type B. That formula expresses the function pu_F through one value $\text{pu}_F(1)$ of type $F^{\mathbb{1}}$. This value can be viewed as a "wrapped unit" value.

To perform the same derivation in Scala syntax, we may write

```
val one: Unit = ()
val f: Unit => B = { _ => b }
pure(one).map(f) == pure(f(one)) == pure(b)    // Because f(one) == b.
```

It follows that if pu_F satisfies the naturality law then a single "wrapped unit" value of that function, $\text{pu}_F(1)$, is sufficient to recover the entire function $\text{pu}_F(b)$ by using this code:

$$\text{pu}_F(b) \stackrel{\text{def}}{=} \text{pu}_F(1) \triangleright (_ \to b)^{\uparrow F}$$

So, given just a "wrapped unit" value (denoted wu_F) of type $F^{\mathbb{1}}$, we can define a new function pu_F:

```
def pure[A](x: A): F[A] = wu.map { _ => x }
```

$$\text{pu}_F^{:A\to F^A} \stackrel{\text{def}}{=} x^{:A} \to \text{wu}_F \triangleright (_ \to x)^{\uparrow F} \quad . \tag{8.10}$$

Does this function satisfy the naturality law with respect to an arbitrary $f^{:A\to B}$? It does:

$$\begin{aligned}
\text{expect to equal } x \triangleright f \fatsemi \text{pu}_F : \quad & x \triangleright \text{pu}_F \fatsemi f^{\uparrow F} \\
\text{definition of } \text{pu}_F : \quad & = \text{wu}_F \triangleright (_ \to x)^{\uparrow F} \fatsemi f^{\uparrow F} \\
\text{functor composition law of } F : \quad & = \text{wu}_F \triangleright ((_ \to x) \fatsemi f)^{\uparrow F} \\
\text{compute function composition} : \quad & = \text{wu}_F \triangleright (_ \to f(x))^{\uparrow F} \\
\text{definition of } \text{pu}_F : \quad & = \text{pu}_F(f(x)) \\
\triangleright\text{-notation} : \quad & = x \triangleright f \triangleright \text{pu}_F = x \triangleright f \fatsemi \text{pu}_F \quad .
\end{aligned}$$

Applied to the unit value, this new function gives $\text{pu}_F(1) = \text{wu}_F$ because:

$$\begin{aligned}
\triangleright\text{-notation} : \quad & \text{pu}_F(1) = 1 \triangleright \text{pu}_F \\
\text{definition of } \text{pu}_F \text{ via } \text{wu}_F : \quad & = \text{wu}_F \triangleright (_ \to 1)^{\uparrow F} \\
\text{the function } (_ \to 1) \text{ is the identity function } \text{id}^{:\mathbb{1}\to\mathbb{1}} : \quad & = \text{wu}_F \triangleright \text{id}^{\uparrow F} \\
\text{functor identity law of } F : \quad & = \text{wu}_F \triangleright \text{id} = \text{wu}_F \quad .
\end{aligned}$$

To summarize our results, we can say that for any functor F:

- If F is a lawful pointed functor, its `pure` method satisfies Eq. (8.10) where $\text{wu}_F \stackrel{\text{def}}{=} \text{pu}_F(1)$ is a fixed "wrapped unit" value of type $F^{\mathbb{1}}$.
- If any "wrapped unit" value $\text{wu}_F : F^{\mathbb{1}}$ is given, we may define a `pure` method by Eq. (8.10) and make the functor F into a lawful pointed functor; the naturality law will be satisfied automatically. The value $\text{pu}_F(1)$ will be equal to the originally given value wu_F.

So, the types of pu_F and wu_F are **equivalent**: each one can be converted into the other and back without loss of information. We may define a pointed functor equivalently as a functor with a chosen value wu_F of type $F^{\mathbb{1}}$. When reasoning about pointed functors, it is simpler to use the definition via the "wrapped unit" wu_F because it has a simpler type and no laws. When writing code, the `pure` method is more convenient.

8.3.6 Pointed functors: structural analysis

To perform structural analysis, we begin with the known functor-building constructions from Chapter 6 and impose an additional requirement that a "wrapped unit" value $\text{wu}_F : F^{\mathbb{1}}$ should exist. As we have seen in the previous section, no additional laws need to be checked. The results are shown in Table 8.5. The typeclass can be defined by a simpler code:

```
final case class Pointed[F[_]](wu: F[Unit])
def pure[F[_]: Pointed : Functor, A](a: A): F[A] = implicitly[Pointed[F]].wu.map(_ => a)
```

When the type $F^{\mathbb{1}}$ has several distinct values, the choice of wu_F is application-dependent. E.g., if $F \stackrel{\text{def}}{=} \text{List}$, the type $\text{List}^{\mathbb{1}}$ has values such as an empty list `List[Unit]()`, a list of length 1: `List(())`, a list of length 2: `List((), ())`, and so on. Each of these choices gives a valid `Pointed` instance for the `List` functor. It is up to the programmer to choose the `Pointed` instance that will be useful for the application at hand. In the case of `List`, the standard choice `wu == List(())` and correspondingly `pure(x) = List(x)` is motivated by the usage of the `List` functor to represent a choice of possibilities, e.g., in a search problem; then the "pure" list represents the situation with only one possibility.

Nameless type-to-type functions To implement the `Pointed` typeclass instances for the following constructions, we need to use some advanced syntax features of Scala. In the previous sections, we wrote PTVFs parameterized by a type built out of other types, for example:

```
def monoidPair[A: Monoid, B: Monoid]: Monoid[(A, B)] = ???
```

The function `monoidPair` creates a typeclass instance for the type $A \times B$, which is built out of the types A and B. In Scala, the typeclass instance is a value of type `Monoid[(A, B)]`. The analog for type constructors is a value of type `Pointed[L]` where the type constructor `L` is defined by the type alias:

```
type L[A] = (F[A], G[A])
```

However, the following declaration of the analogous function `pointedPair` is invalid in Scala:

```
def pointedPair[F[_]: Pointed, G[_]: Pointed]: Pointed[L]     // Does not work in Scala.
```

It is not possible to use the type alias `L` within this function declaration, because the type alias needs to use the type parameters `F` and `G` that are defined only within the type signature of the function. To achieve that, we would need somehow to insert a new type alias declaration within the type signature of `pointedPair`, but the syntax of Scala does not support that:

```
def pointedPair[F[_]: Pointed, G[_]: Pointed]: (            // Not a valid Scala syntax.
    type L[A] = (F[A], G[A]) // Temporarily define a type constructor L, and now use it:
    Pointed[L] )
```

The return type is required to be `Pointed[L]`, where `L` needs to be a type expression that defines a type constructor, i.e., a type-to-type function with a *single* type parameter. Writing a type alias with parameters `F` and `G`:

```
type L[F[_], G[_], A] = (F[A], G[A])
def pointedPair[F[_]: Pointed, G[_]: Pointed]: Pointed[L[F, G]]    // Still does not work.
```

will not help because the type expression `L[F, G]` in `Pointed[L[F, G]]` is not a valid type constructor with one type parameter. We cannot define a new type name (such as `L`) within the type signature; what we need is a *nameless* type-to-type function (i.e., a nameless type constructor).

The special Scala plugin called the "kind projector"[5] adds syntax for nameless type constructors. The syntax is similar to defining a nameless function: for instance, the pair functor $F^\bullet \times G^\bullet$ is defined as `Lambda[X => (F[X], G[X])]`. Such type expressions can be understood as nameless type-to-type functions. When using the "kind projector" plugin, the syntax for defining `pointedPair` is:

```
def pointedPair[F[_]: Pointed, G[_]: Pointed]: Pointed[Lambda[X => (F[X], G[X])]] = ???
```

[5] https://github.com/typelevel/kind-projector

Scala 3 will support a shorter syntax for nameless type-to-type functions, e.g., `[X] => (F[X], G[X])`. We will use the "kind projector" syntax in this book's code examples.

Fixed types A constant functor $\text{Const}^{Z,\bullet}$ is defined as $\text{Const}^{Z,A} \stackrel{\text{def}}{=} Z$, where Z is a fixed type. A "wrapped unit" value is thus also a value of type Z. Since we cannot produce values of an arbitrary type Z from scratch, the constant functor is not pointed in general. The constant functor *will* be pointed when there exists a known value of type Z. Examples are $Z = \mathbb{1}$ or $Z = \mathbb{1} + U$ (where U is an arbitrary type). If we know that Z is equivalent to $\mathbb{1} + U$, we will be able to produce a value of type Z as $1 + \mathbb{0}^{:U}$. In that case, we set $\text{wu}_{\text{Const}^{Z,\bullet}} = 1 + \mathbb{0}^{:U}$.

```
type Const[Z, A] = Z
def pointedOpt[U]: Pointed[Lambda[X => Const[Option[U], X]]] = Pointed(None: Const[Option[U], Unit])
```

Other cases are types such as $Z = \text{Int}$, $Z = \text{String}$, etc., that have well-known "default" values.

Type parameters The identity functor $\text{Id}^A \stackrel{\text{def}}{=} A$ is pointed since $\text{Id}^{\mathbb{1}} = \mathbb{1}$, and we can set $\text{wu}_{\text{Id}} = 1$.

```
type Id[A] = A
def pointedId: Pointed[Id] = Pointed[Id](())
```

The other functor constructions that work by setting type parameters are functor compositions. If F and G are two functors or two contrafunctors then $F \circ G$ is a functor. The functor $F \circ G$ is pointed when we can create a value of type $F^{G^{\mathbb{1}}}$. If both F and G are pointed, we can apply F's `pure` method to $\text{wu}_G : G^{\mathbb{1}}$ and obtain a value of type $F^{G^{\mathbb{1}}}$.

```
def pointedFoG[F[_]: Pointed : Functor, G[_]: Pointed]: Pointed[Lambda[X => F[G[X]]]]
  = Pointed[Lambda[X => F[G[X]]]](pure[F, G[Unit]](implicitly[Pointed[G]].wu))
```

The case when F and G are contrafunctors requires us to assume that F belongs to the "pointed contrafunctor" typeclass (see Section 8.3.8 below). A pointed contrafunctor has a "wrapped unit" value of type $F^{\mathbb{1}}$, which can be transformed into F^A for any type A by using `contramap` with a constant function $A \to \mathbb{1}$:

```
def cpure[F[_]: Pointed : Contrafunctor, A]: F[A] = implicitly[Pointed[F]].wu.cmap(_ => ())
```

In this way, we can create a value of type $F^{G^{\mathbb{1}}}$. The contrafunctor G does not need to be pointed.

```
def pointedCFoG[F[_]: Pointed : Contrafunctor, G[_]]: Pointed[Lambda[X => F[G[X]]]] =
  Pointed[Lambda[X => F[G[X]]]](cpure[F, G[Unit]])
```

Products If F and G are two pointed functors, is the functor product $L^A \stackrel{\text{def}}{=} F^A \times G^A$ a pointed functor? We need to produce a value $\text{wu}_L : F^{\mathbb{1}} \times G^{\mathbb{1}}$, and we have values $\text{wu}_F : F^{\mathbb{1}}$ and $\text{wu}_G : G^{\mathbb{1}}$. It is clear that we must set $\text{wu}_L = \text{wu}_F \times \text{wu}_G$.

```
def pointedFxG[F[_]: Pointed, G[_]: Pointed]: Pointed[Lambda[X => (F[X],G[X])]] =
  Pointed[Lambda[X => (F[X],G[X])]]((implicitly[Pointed[F]].wu, implicitly[Pointed[G]].wu))
```

Co-products If F and G are two pointed functors, is the functor co-product $L^A \stackrel{\text{def}}{=} F^A + G^A$ a pointed functor? We need to produce a value $\text{wu}_L : F^{\mathbb{1}} + G^{\mathbb{1}}$, and we have values $\text{wu}_F : F^{\mathbb{1}}$ and $\text{wu}_G : G^{\mathbb{1}}$. There are two choices, $\text{wu}_L = \text{wu}_F + \mathbb{0}^{:G^{\mathbb{1}}}$ and $\text{wu}_L = \mathbb{0}^{:F^{\mathbb{1}}} + \text{wu}_G$, both making $L^\bullet$ a pointed functor.

It is sufficient if just $F^\bullet$ is a pointed functor: $\text{wu}_L \stackrel{\text{def}}{=} \text{wu}_F + \mathbb{0}^{:G^{\mathbb{1}}}$ is a `Pointed` typeclass instance for $F^\bullet + G^\bullet$, even if $G^\bullet$ is not pointed.

```
def pointedEitherFG[F[_]: Pointed, G[_]]: Pointed[Lambda[X => Either[F[X],G[X]]]] =
  Pointed[Lambda[X => Either[F[X],G[X]]]](Left(implicitly[Pointed[F]].wu))
```

Functions If C is any contrafunctor and F is a pointed functor, the functor $L^A \stackrel{\text{def}}{=} C^A \to F^A$ will be pointed if we can produce a value $\text{wu}_L : C^{\mathbb{1}} \to F^{\mathbb{1}}$. We already have a value $\text{wu}_F : F^{\mathbb{1}}$, and we cannot use a value of type $C^{\mathbb{1}}$ with a general contrafunctor C. So, we have to set $\text{wu}_L \stackrel{\text{def}}{=} (_ \to \text{wu}_F)$. This makes L into a pointed functor.

```
def pointedFuncFG[F[_]: Pointed, C[_]]: Pointed[Lambda[X => C[X] => F[X]]] =
  Pointed[Lambda[X => C[X] => F[X]]](_ => implicitly[Pointed[F]].wu)
```

Recursive types The recursive construction for functors (see Statement 6.2.3.7) assumes a bifunctor $S^{\bullet,\bullet}$ and defines a recursive functor $F^\bullet$ via the type equation $F^A \stackrel{\text{def}}{=} S^{A,F^A}$. The functor $F^\bullet$ will be pointed if we can compute a value wu_F of type $F^{\mathbb{1}}$. The type $F^{\mathbb{1}}$ is a recursive type defined via the type equation $F^{\mathbb{1}} \stackrel{\text{def}}{=} S^{\mathbb{1},F^{\mathbb{1}}}$. If that type is not void, i.e., if there exists some value of that type, we will be able to define wu_F as that value.

How can we construct wu_F for a given bifunctor S? The procedure can be derived by structural analysis of S (see Section 8.5.1 below). For *polynomial* bifunctors S (which is the most often used kind of bifunctors), the necessary and sufficient condition is that the type $S^{\mathbb{1},\mathbb{0}}$ should be non-void. If we can create a value of type $S^{\mathbb{1},\mathbb{0}}$, arguments shown in Section 8.5.1 will guarantee that we can also create a value of type $F^{\mathbb{1}}$, and so the recursive functor F will be pointed.

As an example, consider the polynomial bifunctor $S^{A,R} \stackrel{\text{def}}{=} A + A \times R$. The corresponding recursive functor $F^A \stackrel{\text{def}}{=} S^{A,F^A} = A + A \times F^A$ is the non-empty list (see Example 3.3.2.1, Table 6.3, and Statement 6.2.3.7). The type F^A can be (non-rigorously) viewed as an "infinite disjunction":

$$F^A = A + A \times (A + A \times (A + ...)) = A + A \times A + A \times A \times A + ...$$

Since the type $S^{\mathbb{1},\mathbb{0}} = \mathbb{1} + \mathbb{1} \times \mathbb{0} \cong \mathbb{1}$ is non-void, the necessary and sufficient condition holds, so we expect that the recursive construction will work. The type $F^{\mathbb{1}}$ is defined by:

$$F^{\mathbb{1}} \stackrel{\text{def}}{=} S^{\mathbb{1},F^{\mathbb{1}}} = \mathbb{1} + \mathbb{1} \times F^{\mathbb{1}} \cong \mathbb{1} + F^{\mathbb{1}} \quad .$$

This type can be also viewed as an "infinite disjunction":

$$F^{\mathbb{1}} \cong \mathbb{1} + F^{\mathbb{1}} \cong \mathbb{1} + \mathbb{1} + F^{\mathbb{1}} = \mathbb{1} + \mathbb{1} + \mathbb{1} + ...$$

It is clear that a value of that type can be computed, for example, as:

$$\text{wu}_F = 1 + \mathbb{0} + \mathbb{0} + ... \cong 1 + \mathbb{0}^{:F^{\mathbb{1}}} \quad .$$

In Scala, this is `Left(())`. So, a `Pointed` typeclass instance for F is implemented by the following code:

```
type S[A, R] = Either[A, (A, R)]
final case class F[A](s: S[A, F[A]])
implicit val pointedF: Pointed[F] =
  Pointed( F(Left(())) )
```

The corresponding `pure` method will use F's `map` to transform $\text{wu}_F = 1 + \mathbb{0} + \mathbb{0} + ...$ into:

$$a + \mathbb{0} + \mathbb{0} + ... : A + A \times A + A \times A \times A + ... \quad .$$

The `pure` method of F creates a non-empty list with a single element $a^{:A}$.

Summary Can we recognize a pointed functor F by looking at its type expression, e.g.:

$$F^{A,B} \stackrel{\text{def}}{=} ((\mathbb{1} + A \to \text{Int}) \to A \times B) + \text{String} \times A \times A \quad ?$$

This type constructor is a functor in both A and B, and we ask whether F is pointed with respect to A and/or with respect to B.

To answer this question with respect to A, we set $A = \mathbb{1}$ in $F^{A,B}$ and obtain the type expression:

$$F^{\mathbb{1},B} = ((\mathbb{1} + \mathbb{1} \to \text{Int}) \to \mathbb{1} \times B) + \text{String} \times \mathbb{1} \times \mathbb{1} \cong ((\mathbb{2} \to \text{Int}) \to B) + \text{String} \quad .$$

The functor F will be pointed with respect to A if we can compute a value of this type from scratch. At the outer level, this type expression is a disjunctive type with two parts; it is sufficient to compute one of the parts. Can we compute a value of type $(\mathbb{2} \to \text{Int}) \to B$? Since the parameter B is an arbitrary, unknown type, we cannot construct values of type B using a given value of type $\mathbb{2} \to \text{Int}$. The remaining possibility is to compute the second part of the co-product, which is $\text{String} \times \mathbb{1} \times \mathbb{1}$. We are able to compute a value of this type because `String` is a fixed type with a known default value (an empty string). So, we conclude that $F^{A,B}$ is pointed with respect to A.

Considering now the type parameter B, we set $B = \mathbb{1}$ and obtain:

$$F^{A,\mathbb{1}} = ((\mathbb{1} + A \to \text{Int}) \to A \times \mathbb{1}) + \text{String} \times A \times A \quad .$$

Construction	Type signature to implement
Constant functor returning a fixed type Z	value of type Z
Identity functor	$\mathbb{1}$
Composition of pointed functors/contrafunctors	$\text{Pointed}^{F^\bullet} \times \text{Pointed}^{G^\bullet} \to \text{Pointed}^{F^{G^\bullet}}$
Product of pointed functors F and G	$\text{Pointed}^{F^\bullet} \times \text{Pointed}^{G^\bullet} \to \text{Pointed}^{F^\bullet \times G^\bullet}$
Co-product of a pointed functor F and any G	$\text{Pointed}^{F^\bullet} \times \text{Functor}^{G^\bullet} \to \text{Pointed}^{F^\bullet + G^\bullet}$
Function from any C to a pointed F	$\text{Pointed}^{F^\bullet} \times \text{Contrafunctor}^{C^\bullet} \to \text{Pointed}^{C^\bullet \to F^\bullet}$
Recursive type	$\text{Pointed}^{F^\bullet} \to \text{Pointed}^{S^{\bullet, F^\bullet}}$ where $F^A \stackrel{\text{def}}{=} S^{A, F^A}$

Table 8.5: Type constructions producing the `Pointed` functor typeclass.

The type A is now an arbitrary and unknown type, so we cannot compute any values of A or $\text{String} \times A \times A$ from scratch. The function type $(\mathbb{1} + A \to \text{Int}) \to A \times \mathbb{1}$ cannot be implemented because a value of type A cannot be computed from a function $\mathbb{1} + A \to \text{Int}$ that *consumes* values of type A. So, $F^{A,B}$ is not pointed with respect to B.

8.3.7 Co-pointed functors

Pointed functors provide the functionality of wrapping a given value in a "pure wrapper". Another useful operation is *extracting* a value from a given "wrapper":

```
def extract[A]: F[A] => A
```

$$\text{ex} : \forall A.\, F^A \to A \quad .$$

Functors having this operation are called **co-pointed**. We may define the `Copointed` typeclass as

```
trait Copointed[F[_]] { def ex[A]: F[A] => A }
def extract[F[_]: Copointed, A](f: F[A]): A = implicitly[Copointed[F]].ex(f)
```

The `extract` function must be fully parametric and obey the **naturality law** (compare to Eq. (8.8)):

$$\begin{tikzcd} F^A \arrow[r, "\text{ex}_F"] \arrow[d, "f^{\uparrow F}"'] & A \arrow[d, "f"] \\ F^B \arrow[r, "\text{ex}_F"] & B \end{tikzcd}$$

$$\text{ex}_F \,⨾\, f = f^{\uparrow F} \,⨾\, \text{ex}_F \quad . \tag{8.11}$$

The naturality law formulates our expectation that the extractor function somehow "selects" a value of type A among all the values wrapped by F^A, and the "selection" works independently of the values. If all wrapped values are transformed by a function f into wrapped values of type B, the extractor function will still select a value of type B in the same way as it did for values of type A. So, the result will be the same as if we first extracted a value of type A and then transformed that value with f.

Both sides of the law (8.11) are functions of type $F^A \to B$. We saw in the previous section that, when F is a functor, the type of the `pure` method of the `Pointed` typeclass is equivalent to the type $F^{\mathbb{1}}$. That provides a simpler form of the `Pointed` typeclass. For co-pointed functors, there is no simpler form of the `extract` method. If we set $A = \mathbb{1}$ and $f^{:\mathbb{1} \to B} \stackrel{\text{def}}{=} (1 \to b)$ in the naturality law, both sides will become functions of type $F^{\mathbb{1}} \to B$. But the type $F^{\mathbb{1}}$ might be void, or a value of type $F^{\mathbb{1}}$ may not be computable via fully parametric code. So, we cannot deduce any further information from the naturality law of co-pointed functors.

However, if F is also a pointed functor, we *will* have a chosen value $\text{wu}_F : F^{\mathbb{1}}$ to which we may then apply both sides of the naturality law (8.11) and obtain:[6]

$$\text{wu}_F \triangleright \text{ex}_F \,⨾\, f \stackrel{!}{=} \text{wu}_F \triangleright f^{\uparrow F} \,⨾\, \text{ex}_F \quad .$$

[6]The symbol $\stackrel{!}{=}$ means "must be equal, according to what we know".

Calculating both sides separately, we find:

$$\text{wu}_F \triangleright \text{ex}_F \fatsemi f = \underline{\text{wu}_F \triangleright \text{ex}_F} \triangleright f = 1 \triangleright f = b \quad ,$$
$$\underline{\text{wu}_F \triangleright f^{\uparrow F}} \fatsemi \text{ex}_F = \text{pu}_F(b) \triangleright \text{ex}_F = b \triangleright \text{pu}_F \fatsemi \text{ex}_F \quad .$$

So, we know that $b \triangleright \text{pu}_F \fatsemi \text{ex}_F \overset{!}{=} b$. This can hold for all $b^{:B}$ only if:

$$\text{pu}_F \fatsemi \text{ex}_F = \text{id} \quad .$$

This additional **nondegeneracy law** is a consequence of naturality laws if the functor F is pointed and co-pointed at the same time.

In the derivation above, we have used the equation:

$$\text{wu}_F \triangleright \text{ex}_F = 1 \quad . \tag{8.12}$$

This equation holds automatically, simply because its left-hand side is of type $\mathbb{1}$, which has only one value (which we denote by 1). □

Let us perform structural analysis for co-pointed functors; the results are shown in Table 8.6.

Fixed types A constant functor $\text{Const}^{Z,A} \overset{\text{def}}{=} Z$ is *not* co-pointed because we cannot implement $\forall A. Z \to A$ (a value of an arbitrary type A cannot be computed from a value of a fixed type Z).

Type parameters The identity functor $\text{Id}^A \overset{\text{def}}{=} A$ is co-pointed with $\text{ex} \overset{\text{def}}{=} \text{id}^{:A\to A}$. An identity function will always satisfy any naturality law.

```
type Id[A] = A
def copointedId: Copointed[Id] = new Copointed[Id] { def ex[A]: Id[A] => A = identity }
```

Composition of two co-pointed functors F, G is co-pointed:

$$\text{ex}_{F\circ G} \overset{\text{def}}{=} h^{:F^{G^A}} \to \text{ex}_G(\text{ex}_F(h)) \quad \text{or equivalently} \quad \text{ex}_{F\circ G} = \text{ex}_F \fatsemi \text{ex}_G \quad .$$

```
def copointedFoG[F[_]: Copointed, G[_]: Copointed]: Copointed[Lambda[X => F[G[X]]]] =
  new Copointed[Lambda[X => F[G[X]]]] {
    def ex[A]: F[G[A]] => A = extract[F,G[A]] _ andThen extract[G, A]
  }
```

The naturality law holds for $\text{ex}_{F\circ G}$ because:

$$\begin{aligned}
\text{expect to equal } \text{ex}_{F\circ G} \fatsemi f: \quad & \underline{f^{\uparrow F\circ G}} \fatsemi \underline{\text{ex}_{F\circ G}} \\
\text{definition of } f^{\uparrow F\circ G}\text{, see Eq. (6.18)}: \quad & = \underline{(f^{\uparrow G})^{\uparrow F} \fatsemi \text{ex}_F} \fatsemi \text{ex}_G \\
\text{naturality law of } \text{ex}_F: \quad & = \text{ex}_F \fatsemi \underline{f^{\uparrow G} \fatsemi \text{ex}_G} \\
\text{naturality law of } \text{ex}_G: \quad & = \underline{\text{ex}_F \fatsemi \text{ex}_G} \fatsemi f = \text{ex}_{F\circ G} \fatsemi f \quad .
\end{aligned}$$

Products If functors F or G are co-pointed, we can implement a function of type $F^A \times G^A \to A$ by discarding F^A (if G is co-pointed) or by discarding G^A (if F is co-pointed). The functor product $F^\bullet \times G^\bullet$ is then made into a co-pointed functor. For instance, if F is co-pointed, we discard G^A, so the code for the `extract` method will be:

$$\text{ex}_{F\times G} \overset{\text{def}}{=} f^{:F^A} \times g^{:G^A} \to \text{ex}_F(f) = \pi_1 \fatsemi \text{ex}_F \quad ,$$

where we used the pair projection function $\pi_1 \overset{\text{def}}{=} (a \times b \to a)$.

```
def copointedFxG[F[_]: Copointed, G[_]]: Copointed[Lambda[X => (F[X], G[X])]] =
  new Copointed[Lambda[X => (F[X], G[X])]] {
    def ex[A]: ((F[A], G[A])) => A = { case (f, g) => extract(f) }
  }
```

The following calculation verifies the naturality law (8.11) for this definition of $\text{ex}_{F\times G}$:

$$\begin{aligned}
\text{expect to equal } \text{ex}_{F\times G} \fatsemi f : \quad & \underline{f^{\uparrow F\times G}} \fatsemi \text{ex}_{F\times G} \\
\text{definition of } f^{\uparrow F\times G}\text{, see Eq. (6.13)} : \quad & = (f^{\uparrow F} \boxtimes f^{\uparrow G}) \fatsemi \underline{\text{ex}_{F\times G}} \\
\text{definition of } \text{ex}_{F\times G} : \quad & = \underline{(f^{\uparrow F} \boxtimes f^{\uparrow G}) \fatsemi \pi_1 \fatsemi \text{ex}_F} \\
\text{use the property } (p \boxtimes q) \fatsemi \pi_1 = \pi_1 \fatsemi p : \quad & = \pi_1 \fatsemi \underline{f^{\uparrow F} \fatsemi \text{ex}_F} \\
\text{naturality law of } \text{ex}_F : \quad & = \underline{\pi_1 \fatsemi \text{ex}_F} \fatsemi f \\
\text{definition of } \text{ex}_{F\times G} : \quad & = \text{ex}_{F\times G} \fatsemi f \quad .
\end{aligned}$$

To demonstrate the property $(p \boxtimes q) \fatsemi \pi_1 = \pi_1 \fatsemi p$ used in this proof, apply both sides to $a \times b$:

$$\begin{aligned}
&(a \times b) \triangleright (p \boxtimes q) \underline{\fatsemi} \pi_1 = \underline{(a \times b) \triangleright (p \boxtimes q)} \triangleright \pi_1 = (p(a) \times q(b)) \triangleright \pi_1 = p(a) \quad , \\
&(a \times b) \triangleright \pi_1 \underline{\fatsemi} p = \underline{(a \times b) \triangleright \pi_1} \triangleright p = a \triangleright p = p(a) \quad .
\end{aligned}$$

Co-products For co-pointed functors F and G, there is only one possible implementation of the type signature $\text{ex}_{F+G} : F^A + G^A \to A$, given that we have functions ex_F and ex_G:

```
def copointedEitherFG[F[_]: Copointed, G[_]: Copointed]:
                 Copointed[Lambda[X => Either[F[X],G[X]]]] =
  new Copointed[Lambda[X => Either[F[X],G[X]]]] {
    def ex[A]: Either[F[A], G[A]] => A = {
      case Left(f)    => extract(f)
      case Right(g)   => extract(g)
    }
  }
```

$$\text{ex}_{F+G} \stackrel{\text{def}}{=} \left\| \begin{array}{c|c} & A \\ \hline F^A & \text{ex}_F \\ G^A & \text{ex}_G \end{array} \right| \quad .$$

To verify that ex_{F+G} satisfies the naturality law, we compute:

$$\begin{aligned}
\text{expect to equal } \text{ex}_{F+G} \fatsemi f : \quad & \underline{f^{\uparrow F+G}} \fatsemi \underline{\text{ex}_{F+G}} \\
\text{definition of } f^{\uparrow F+G}\text{, see Eq. (6.14)} : \quad & = \left\| \begin{array}{cc} f^{\uparrow F} & \mathbb{0} \\ \mathbb{0} & f^{\uparrow G} \end{array} \right| \fatsemi \left\| \begin{array}{c} \text{ex}_F \\ \text{ex}_G \end{array} \right| \\
\text{matrix function composition} : \quad & = \left\| \begin{array}{c} \underline{f^{\uparrow F} \fatsemi \text{ex}_F} \\ \underline{f^{\uparrow G} \fatsemi \text{ex}_G} \end{array} \right| \\
\text{naturality laws of } \text{ex}_F \text{ and } \text{ex}_G : \quad & = \left\| \begin{array}{c} \text{ex}_F \underline{\fatsemi f} \\ \text{ex}_G \underline{\fatsemi f} \end{array} \right| = \underline{\left\| \begin{array}{c} \text{ex}_F \\ \text{ex}_G \end{array} \right|} \fatsemi f = \text{ex}_{F+G} \fatsemi f \quad .
\end{aligned}$$

Functions A functor of the form $L^A \stackrel{\text{def}}{=} C^A \to P^A$ (where C is a contrafunctor and P is a functor) will be co-pointed if we can implement a function of type $\forall A. (C^A \to P^A) \to A$:

$$\text{ex}_L : \forall A. (C^A \to P^A) \to A \quad , \qquad \text{ex}_L \stackrel{\text{def}}{=} h^{:C^A \to P^A} \to ???^{:A} \quad .$$

Since the type A is arbitrary, the only way of computing a value of type A is somehow to use the function h. The only way of using h is to apply it to a value of type C^A, which will yield a value of type P^A. So, we need to assume that we can somehow create a value of type C^A, for any type A. We may call a contrafunctor C with a method `cpure` of type $\forall A. C^A$ a **pointed contrafunctor**. Assuming that C is pointed and denoting its `cpure` method by cpu_C, we can thus compute a value of type P^A as $h(\text{cpu}_C)$. To extract A from P^A, we need to assume additionally that P is co-pointed and use its method $\text{ex}_P : P^A \to A$. Finally, we have:

$$\text{ex}_L \stackrel{\text{def}}{=} h^{:C^A \to P^A} \to \text{ex}_P(h(\text{cpu}_C)) \qquad \text{or equivalently :} \qquad h^{:C^A \to P^A} \triangleright \text{ex}_L = \text{cpu}_C \triangleright h \triangleright \text{ex}_P \quad . \tag{8.13}$$

To verify the naturality law, we apply both sides to an arbitrary $h^{:C^A \to P^A}$ and compute

$$\begin{aligned}
\text{expect to equal } h \triangleright \text{ex}_L \,;\, f : \quad & h \triangleright f^{\uparrow L} \,;\, \text{ex}_L = (h \triangleright f^{\uparrow L}) \triangleright \underline{\text{ex}_L} \\
\text{use Eq. (8.13)} : \quad & = \text{cpu}_C \triangleright (h \triangleright f^{\uparrow L}) \triangleright \text{ex}_P = \text{cpu}_C \triangleright \underline{(h \triangleright f^{\uparrow L})} \,;\, \text{ex}_P \\
\text{definition of } f^{\uparrow L}\text{, see Eq. (6.16)} : \quad & = \text{cpu}_C \triangleright f^{\downarrow C} \,;\, h \,;\, \underline{f^{\uparrow P} \,;\, \text{ex}_P} \\
\text{naturality law of } \text{ex}_P : \quad & = \text{cpu}_C \triangleright f^{\downarrow C} \,;\, h \,;\, \text{ex}_P \,;\, f \quad .
\end{aligned}$$

We expect the last expression to equal this one:

$$h \triangleright \text{ex}_L \,;\, f = \text{cpu}_C \triangleright h \triangleright \text{ex}_P \triangleright f = \text{cpu}_C \triangleright h \,;\, \text{ex}_P \,;\, f \quad .$$

This is possible only if $\text{cpu}_C \triangleright f^{\downarrow C} = \text{pu}_C$ for all f. This motivates us to assume that law as the **naturality law** of cpu_C for pointed contrafunctors. With that last assumption, we have finished proving the naturality law of ex_L. The code for ex_L is

```
def copointedFunc[C[_]: Pointed, P[_]: Copointed]: Copointed[Lambda[X => C[X] => P[X]]] =
  new Copointed[Lambda[X => C[X] => P[X]]] {
    def ex[A]: (C[A] => P[A]) => A = h => extract[P, A](h(cpure[C, A]))
  }      // In Scala 2.13: h => cpure[C, A] pipe h pipe extract[P, A] as in h → pu_C ▷ h ▷ ex_P
```

We will analyze pointed contrafunctors and their naturality law in Section 8.3.8.

Recursive types Consider a functor F defined by a recursive equation $F^A \stackrel{\text{def}}{=} S^{A,F^A}$ where S is a bifunctor (see Section 6.2.2). The functor F is co-pointed if a method $\text{ex}_F : (F^A \to A) \cong (S^{A,F^A} \to A)$ can be defined. Since the recursive definition of F uses F^A as a type argument in S^{A,F^A}, we may assume (by induction) that an extractor function $F^A \to A$ is already available when applied to the recursively used F^A. Then we can use the `bimap` method of S to map $S^{A,F^A} \to S^{A,A}$. It remains to extract a value of type A out of a bifunctor value $S^{A,A}$. We call a bifunctor S **co-pointed** if a fully parametric function $\text{ex}_S : S^{A,A} \to A$ exists and satisfies the corresponding **naturality law**:

$$\text{ex}_S \,;\, f = \text{bimap}_S(f)(f) \,;\, \text{ex}_S \quad . \tag{8.14}$$

Assuming that S is co-pointed, we can finally define ex_F by recursion:

$$\text{ex}_F \stackrel{\text{def}}{=} s^{:S^{A,F^A}} \to s \triangleright \left(\text{bimap}_S(\text{id})(\text{ex}_F)\right) \triangleright \text{ex}_S \quad \text{or equivalently} \quad \text{ex}_F \stackrel{\text{def}}{=} \text{bimap}_S(\text{id})(\text{ex}_F) \,;\, \text{ex}_S \quad .$$

To verify the naturality law of ex_F, we denote recursive uses by an overline and compute:

$$\begin{aligned}
\text{expect to equal } \text{ex}_F \,;\, f : \quad & \underline{f^{\uparrow F}} \,;\, \text{ex}_F \\
\text{definition of } f^{\uparrow F}\text{, see Eq. (6.20)} : \quad & = \text{bimap}_S(f)(\overline{f^{\uparrow F}}) \,;\, \underline{\text{ex}_F} \\
\text{definition of } \text{ex}_F : \quad & = \underline{\text{bimap}_S(f)(\overline{f^{\uparrow F}}) \,;\, \text{bimap}_S(\text{id})(\overline{\text{ex}_F})} \,;\, \text{ex}_S \\
\text{bifunctor composition law (6.10)} : \quad & = \text{bimap}_S(f \,;\, \text{id})(\underline{\overline{f^{\uparrow F} \,;\, \text{ex}_F}}) \,;\, \text{ex}_S \\
\text{naturality law of } \text{ex}_F : \quad & = \underline{\text{bimap}_S(f)(\overline{\text{ex}_F \,;\, f})} \,;\, \text{ex}_S \\
\text{bifunctor composition law in reverse} : \quad & = \text{bimap}_S(\text{id})(\overline{\text{ex}_F}) \,;\, \text{bimap}_S(f)(\overline{f}) \,;\, \text{ex}_S \\
\text{naturality law (8.14) of } \text{ex}_S : \quad & = \underline{\text{bimap}_S(\text{id})(\overline{\text{ex}_F}) \,;\, \text{ex}_S} \,;\, f \\
\text{definition of } \text{ex}_F : \quad & = \text{ex}_F \,;\, f \quad .
\end{aligned}$$

An example illustrating the recursive construction is the bifunctor $S^{A,R} \stackrel{\text{def}}{=} A + R \times R$ that defines F^A to be the binary tree functor:

$$F^A \stackrel{\text{def}}{=} S^{A,F^A} = A + F^A \times F^A \quad .$$

The bifunctor S is co-pointed because there exists a suitable function, $\text{ex}_S : S^{A,A} \to A$.

Construction	Type signature to implement
Identity functor	$\text{id} : A \to A$
Composition of co-pointed functors	$\text{Copointed}^{F^\bullet} \times \text{Copointed}^{G^\bullet} \to \text{Copointed}^{F^{G^\bullet}}$
Product of co-pointed functor F and any G	$\text{Copointed}^{F^\bullet} \times \text{Functor}^{G^\bullet} \to \text{Copointed}^{F^\bullet \times G^\bullet}$
Co-product of co-pointed functors F and G	$\text{Copointed}^{F^\bullet} \times \text{Copointed}^{G^\bullet} \to \text{Copointed}^{F^\bullet + G^\bullet}$
Function from pointed C to co-pointed F	$\text{Pointed}^{C^\bullet} \times \text{Copointed}^{F^\bullet} \to \text{Copointed}^{C^\bullet \to F^\bullet}$
Recursive type	$\text{Copointed}^{F^\bullet} \to \text{Copointed}^{S^{\bullet,F^\bullet}}$ where $F^A \stackrel{\text{def}}{=} S^{A,F^A}$

Table 8.6: Type constructions producing the `Copointed` functor typeclass.

```
type S[A, R] = Either[A, (R, R)]
def exS[A]: S[A, A] => A = {
  case Left(a)          => a
  case Right((a1, a2))  => a1 // Could be a2.
}
```

In the code notation, the function ex_S is written as:

$$\text{ex}_S \stackrel{\text{def}}{=} \left| \begin{array}{c||c} & A \\ \hline A & \text{id} \\ A \times A & \pi_1 \end{array} \right| .$$

This function extracts the left-most leaf of a binary tree because it is using the projection π_1 (not π_2).

To implement the co-pointed instance ex_F for the binary tree functor, we need to use the implementation of bimap_S:

```
def bimap_S[A,B,P,Q](f: A => B)(g: P => Q): S[A, P] => S[B, Q] = {
  case Left(a)          => Left(f(a))
  case Right((x, y))    => Right((g(x), g(y)))
}
```

Now we can define the recursive type constructor $F^\bullet$ and a co-pointed instance for it:

```
final case class F[A](s: S[A, F[A]])
val copointedF: Copointed[F] = new Copointed[F1] {
  def ex[A]: F1[A] => A = { case F1(s) => exS1(bimap_S(identity[A], ex[A])(s)) }
}
```

The naturality law holds for the recursive instance by induction (see Section 8.3.2).

Summary Can we recognize a co-pointed functor F by looking at its type expression, e.g.:

$$F^{A,B} \stackrel{\text{def}}{=} ((\mathbb{1} + A \to \text{Int}) \to A \times B) + \text{String} \times A \times A \quad ?$$

The constructions shown in this section tell us that a co-product of two co-pointed functors is again co-pointed. Let us first check whether $\text{String} \times A \times A$ is co-pointed. We can certainly extract a value of type A out of $\text{String} \times A \times A$, but not a value of type B. So, there is no hope for $F^{A,B}$ to be co-pointed with respect to the parameter B.

It remains to consider A as the type parameter. The type constructor $\text{String} \times A \times A$ is co-pointed, but we still need to check $(\mathbb{1} + A \to \text{Int}) \to A \times B$, which is a function type. The function construction requires $\mathbb{1} + A \to \text{Int}$ to be a pointed contrafunctor and $A \times B$ to be a co-pointed functor (with respect to A). It is clear that $A \times B$ is co-pointed with respect to A since we have $\pi_1 : A \times B \to A$. It remains to check that the contrafunctor $C^A \stackrel{\text{def}}{=} \mathbb{1} + A \to \text{Int}$ is pointed. A contrafunctor $C^\bullet$ is pointed if values of type $C^\mathbb{1}$ can be computed (see Section 8.3.8); this requires us to compute a value of type $\mathbb{1} + \mathbb{1} \to \text{Int}$. One such value is `{ _ => 0 }`, a constant function that always returns the integer `0`.

We conclude that $F^{A,B}$ is co-pointed with respect to A but not co-pointed with respect to B.

8.3.8 Pointed contrafunctors

In the previous section, the function-type construction required a contrafunctor $C^\bullet$ to have a method cpu_C of type $\forall A. C^A$; we called such contrafunctors **pointed**. We also needed to assume that the

naturality law holds for all functions $f^{:A\to B}$:

$$\text{cpu}_C \triangleright f^{\downarrow C} = \text{cpu}_C \quad , \quad \text{or equivalently}: \quad \text{cmap}_C(f^{:A\to B})(\text{cpu}_C^{:C^B}) = \text{cpu}_C^{:C^A} \quad . \tag{8.15}$$

$$\begin{tikzcd} \mathbb{1} \arrow[r] \arrow[dr] & \text{cpu}_C : C^B \arrow[d, "\text{cmap}_C(f)"] \\ & \text{cpu}_C : C^A \end{tikzcd}$$

We may simplify the formulation of the typeclass by setting $B = \mathbb{1}$ in the naturality law (8.15) and denoting $\text{wu}_C \stackrel{\text{def}}{=} \text{cpu}_C^{\mathbb{1}}$. The law (8.15) then gives:

$$\text{cpu}_C^A = \text{wu}_C \triangleright (_^{:A} \to 1)^{\downarrow C} \quad . \tag{8.16}$$

In this way, we express the `cpure` method through a chosen value $\text{wu}_C : C^{\mathbb{1}}$. For the same reasons as in the case of pointed functors, the types of cpu_C and wu_C are equivalent. The law (8.15) for cpu_C will be satisfied automatically if cpu_C is defined via Eq. (8.16). To verify that, compute:

$$\begin{aligned} \text{expect to equal } \text{cpu}_C^A: \quad & \text{cpu}_C^B \triangleright f^{\downarrow C} \\ \text{use definition (8.16)}: \quad & = \text{wu}_C \triangleright \underline{(_^{:B} \to 1)^{\downarrow C} \triangleright f^{\downarrow C}} \\ \text{composition law of contrafunctor } C: \quad & = \text{wu}_C \triangleright \underline{(f \fatsemi (_^{:B} \to 1))^{\downarrow C}} \\ \text{compute function composition}: \quad & = \text{wu}_C \triangleright (_^{:A} \to 1)^{\downarrow C} = \text{cpu}_C^A \quad . \end{aligned}$$

So, a pointed contrafunctor instance for $C^\bullet$ is equivalent to a chosen value of type $C^{\mathbb{1}}$.

```
final case class Pointed[F[_]](wu: F[Unit])
def cpure[F[_]: Pointed : Contrafunctor, A]: F[A] = implicitly[Pointed[F]].wu.cmap(_ => ())
```

Here we used a `Contrafunctor` typeclass (see Example 8.4.1.7 below). We can now apply structural analysis to pointed contrafunctors, similarly to Section 6.2.4. The results are shown in Table 8.7.

Fixed types This construction is the same as for pointed functors (Section 8.3.6). A fixed type Z gives a constant contrafunctor $C^A \stackrel{\text{def}}{=} Z$. Since $C^{\mathbb{1}} = Z$, the constant contrafunctor is pointed if we have a chosen value of type Z; this will be the case, for instance, if $Z = \mathbb{1} + U$ for some type U.

Type parameters Since the identity functor $\text{Id}^A \stackrel{\text{def}}{=} A$ is not a contrafunctor, it remains to consider the functor compositions C^{F^A} and F^{C^A} where $C^\bullet$ is a contrafunctor and $F^\bullet$ is a functor.

If C is pointed, we can always obtain a value cpu_C of type C^A for any type A, in particular for $A = F^{\mathbb{1}}$ (whether or not a value of type $F^{\mathbb{1}}$ can be computed). So, $C^{F^\bullet}$ is a pointed contrafunctor whenever $C^\bullet$ is one, for any (not necessarily pointed) functor F.

```
def pointedCoF[C[_]: Pointed: Contrafunctor, F[_]]: Pointed[Lambda[X => C[F[X]]]] =
  Pointed[Lambda[X => C[F[X]]]](cpure[C, F[Unit]])
```

Creating a value of type $F^{C^{\mathbb{1}}}$ requires F to have a `pure` method that could be applied to a value of type $C^{\mathbb{1}}$ to compute a value of type $F^{C^{\mathbb{1}}}$. So, $F^{C^\bullet}$ is pointed whenever both $C^\bullet$ and $F^\bullet$ are pointed.

```
def pointedFoC[C[_]: Pointed, F[_]: Pointed : Functor]: Pointed[Lambda[X => F[C[X]]]] =
  Pointed[Lambda[X => F[C[X]]]](pure[F, C[Unit]](implicitly[Pointed[C]].wu))
```

Products The construction is the same as for pointed functors: If we have values of type $C^{\mathbb{1}}$ and $D^{\mathbb{1}}$, we can compute the pair $C^{\mathbb{1}} \times D^{\mathbb{1}}$. This makes the product contrafunctor $L^A \stackrel{\text{def}}{=} C^A \times D^A$ pointed if both $C^\bullet$ and $D^\bullet$ are pointed contrafunctors.

Co-products The construction is the same as for pointed functors: If at least one of the contrafunctors $C^\bullet$ and $D^\bullet$ is pointed, we can create a `Pointed` instance for the co-product contrafunctor $L^A \stackrel{\text{def}}{=} C^A + D^A$ as either $\text{wu}_L = \text{wu}_C + \mathbb{0}^{:D^{\mathbb{1}}}$ or $\text{wu}_L = \mathbb{0}^{:C^{\mathbb{1}}} + \text{wu}_D$.

Construction	Type signature to implement
Constant functor returning a fixed type Z	value of type Z
Composition of pointed functors/contrafunctors	$\text{Pointed}^{F^\bullet} \times \text{Pointed}^{G^\bullet} \to \text{Pointed}^{F^{G^\bullet}}$
Product of pointed contrafunctors F and G	$\text{Pointed}^{F^\bullet} \times \text{Pointed}^{G^\bullet} \to \text{Pointed}^{F^\bullet \times G^\bullet}$
Co-product of a pointed F and any G	$\text{Pointed}^{F^\bullet} \times \text{Contrafunctor}^{G^\bullet} \to \text{Pointed}^{F^\bullet + G^\bullet}$
Function from a functor F to a pointed C	$\text{Pointed}^{C^\bullet} \times \text{Functor}^{F^\bullet} \to \text{Pointed}^{F^\bullet \to C^\bullet}$
Recursive type	$\text{Pointed}^{C^\bullet} \to \text{Pointed}^{S^{\bullet,C^\bullet}}$ where $C^A \stackrel{\text{def}}{=} S^{A,C^A}$

Table 8.7: Type constructions producing the `Pointed` contrafunctor typeclass.

Functions The construction is $L^A \stackrel{\text{def}}{=} F^A \to C^A$, where $C^\bullet$ is a contrafunctor and $F^\bullet$ is a functor. To create a value $\text{wu}_L : L^{\mathbb{1}}$ means to create a function of type $F^{\mathbb{1}} \to C^{\mathbb{1}}$. That function cannot use its argument of type $F^{\mathbb{1}}$ for computing a value $C^{\mathbb{1}}$ since F is an arbitrary functor. So, wu_L must be a constant function $(_^{:F^{\mathbb{1}}} \to \text{wu}_C)$, where we assumed that a value $\text{wu}_C : C^{\mathbb{1}}$ is available. Thus, $F^A \to C^A$ is pointed when C is a pointed contrafunctor and F is any functor.

```
def pointedFuncFC[C[_]: Pointed, F[_]]: Pointed[Lambda[X => F[X] => C[X]]] =
      Pointed[Lambda[X => F[X] => C[X]]](_ => implicitly[Pointed[C]].wu)
```

Recursive types The recursive construction for contrafunctors (see Statement 6.2.4.3) is $C^A \stackrel{\text{def}}{=} S^{A,C^A}$ where $S^{A,R}$ is a contrafunctor in A and a functor in R. Values of type $C^{\mathbb{1}}$ will exist when the recursive type equation $T \stackrel{\text{def}}{=} S^{\mathbb{1},T}$ defines a non-void type T. This condition is similar to that for pointed functors, and the resulting construction is the same.

Summary Can we recognize a pointed contrafunctor C by looking at its type expression, e.g.

$$C^{A,B} \stackrel{\text{def}}{=} (\mathbb{1} + A \to B) + (\text{String} \times A \times B \to \text{String}) \text{ with respect to type parameter } A?$$

We need to set $A = \mathbb{1}$ and try to create a value $\text{wu} : C^{\mathbb{1},B}$. In this example, $C^{\mathbb{1},B} = (\mathbb{1} + \mathbb{1} \to B) + (\text{String} \times \mathbb{1} \times B \to \text{String})$. A value of this type is $\text{wu}_C \stackrel{\text{def}}{=} \mathbb{0} + (s^{:\text{String}} \times 1 \times b^{:B} \to s)$. So, the contrafunctor $C^{A,B}$ is pointed with respect to A.

8.4 Summary

What problems can we solve now?

- Define arbitrary PTVFs using typeclass constraints on type parameters.
- Define typeclasses and typeclass instances for types and for type constructors.
- Implement `Monoid`, `Functor`, and other standard typeclasses, and prove their laws.
- Use known constructions to derive typeclass instances from previous ones.

What problems cannot be solved with these tools?

- *Automatically* derive type class instances for given data types or type constructors.
- Combine typeclasses and express dependencies, e.g., typeclass `TC1` requires `TC2` and `TC3`.

We may want to write code such as:

```
type F[A] = (A => Int) => A                    // Define a type constructor.
implicit val functorF: Functor[F] = implement  // Automatically implement typeclass instance for F.
implicit val pointedF: Pointed[F] = implement  // Automatically use the function-type construction.
```

However, no currently available library provides such functionality. Also, typeclass instances are not always derived uniquely, as we have seen in several cases (e.g., the co-product construction of monoids or pointed functors).

We will discuss how to combine typeclasses in Section 8.5.6 below.

8.4.1 Solved examples

Example 8.4.1.1 Define a PTVF with type signature `def bitsize[T]: Int` such that `bitsize[Short]` returns 16, `bitsize[Int]` returns 32, and `bitsize[Long]` returns 64. For all other types `T`, the expression `bitsize[T]` should remain undefined.

Solution The function `bitsize[T]` needs to take an additional implicit argument that will be available only for `T = Short`, `T = Int`, and `T = Long`. To implement that, we need to define a new type constructor, say `HasBitsize[T]`, and create implicit values of the corresponding types. The new type constructor defines a typeclass whose instances need to carry the information about the bit size:

```
final case class HasBitsize[T](size: Int)
object HasBitsize {
  implicit val bitsizeShort = HasBitsize[Short](16)
  implicit val bitsizeInt   = HasBitsize[Int]  (32)
  implicit val bitsizeLong  = HasBitsize[Long] (64)
}
```

Now we can define the function `bitsize` as a PTVF,

```
def bitsize[T](implicit ti: HasBitsize[T]): Int = ti.size
```

The instance argument such as `ti: HasBitsize[T]` is sometimes called the "evidence" argument because its presence provides "evidence" that the type `T` belongs to the type domain of the typeclass.

We can check that the function `bitsize` is defined only for supported types:

```
scala> bitsize[Long]
res0: Int = 64

scala> bitsize[String]
<console>:15: error: could not find implicit value for evidence parameter of type HasBitsize[String]
       bitsize[String]
               ^
```

The current implementation of `HasBitsize` allows the programmer to add new types to its type domain whenever necessary. For example, the following code will add support for the `Boolean` type so that `bitsize[Boolean]` will evaluate to `1`:

```
implicit val bitsizeBoolean = HasBitsize[Boolean](1)
```

In some applications, it is important that the type domain of a PTVF should remain fixed (e.g., as defined in a library). To prevent the programmer from creating any further values of type `HasBitsize`, we could make it a non-case class whose constructor is declared as a `private` function like this:

```
final class HasBitsize[T] private (val size: Int) // Not a case class; the constructor is private.
object HasBitsize {            // The companion object is allowed to call the private constructor.
  implicit val bitsizeShort = new HasBitsize[Short](16)
  implicit val bitsizeInt   = new HasBitsize[Int]  (32)
  implicit val bitsizeLong  = new HasBitsize[Long] (64)
}
```

The code of `bitsize[T]` remains unchanged. With these definitions, no further typeclass instances can be created by any code outside of the companion object `HasBitsize`:

```
scala> implicit val bitsizeBoolean = new HasBitsize[Boolean](1)
<console>:16: error: constructor HasBitsize in class HasBitsize cannot be accessed in object $iw
       implicit val bitsizeBoolean = new HasBitsize[Boolean](1)
                                     ^
```

An implementation via a `trait` requires longer code but brings no significant advantages:

```
trait HasBitsize[T] { def size: Int } // Declare the trait as ‘sealed‘ to prohibit further instances.
object HasBitsize {
  implicit val bitsizeShort = new HasBitsize[Short]{ def size: Int = 16 }
  implicit val bitsizeInt   = new HasBitsize[Int]  { def size: Int = 32 }
  implicit val bitsizeLong  = new HasBitsize[Long] { def size: Int = 64 }
}
```

Example 8.4.1.2 Define a `Monoid` instance for the type $\mathbb{1} + (\text{String} \to \text{String})$.

Solution We look for suitable monoid constructions (Section 8.3.4) that build up the given type expression from simpler parts. Since the type expression $\mathbb{1} + (\text{String} \to \text{String})$ is a co-product at the outer level, we must start with the co-product construction, which requires us to choose one of the parts of the disjunction, say $\text{String} \to \text{String}$, as the "preferred" monoid. Next, we need to produce `Monoid` instances for $\mathbb{1}$ and for $\text{String} \to \text{String}$. While the `Unit` type has a unique `Monoid` instance, there are several for $\text{String} \to \text{String}$. The function-type construction gives two possible monoid instances: the monoid $R \to A$ with $R = A = \text{String}$, and the function composition monoid. Let us choose the latter. The code using the monoid constructions from Section 8.3.4 can then be written as:

```
val monoidX: Monoid[Either[Unit, String => String]] = {
  implicit val m1 = Monoid[Unit]( (x, y) => (), () )
  implicit val m2: Monoid[String => String] = monoidFunc1[String]
  monoidEitherPreferB[Unit, String => String]
)
```

We can translate the constructions into code for a `Monoid` instance for the type `Option[String => String]`:

```
val monoidX: Monoid[Option[String => String]] = Monoid( {
  case (None, None)          => None
  case (None, Some(f))       => Some(f)
  case (Some(f), None)       => Some(f)
  case (Some(f), Some(g))    => Some(f andThen g)
}, None )
```

Example 8.4.1.3 Show that if A is a monoid and B is a semigroup then $A + B$ is a monoid.

Solution The co-product construction $A + B$ (where both A, B are monoids) has two implementations: one of the empty elements e_A or e_B must be chosen as the empty element for the monoid $A + B$. If B is not a monoid, the only choice is to set $e_{A+B} \stackrel{\text{def}}{=} e_A + \mathbb{0}$. This is the implementation in the function `monoidEitherPreferB` (Section 8.3.4). We just need to replace the `Monoid` typeclass constraint for B by `Semigroup`:

```
def monoidEitherSemigroup[A: Monoid, B: Semigroup] = Monoid[Either[A, B]]( {
  case (Left(a1), Left(a2))      => Left(a1 |+| a2)
  case (Left(a), Right(b))       => Right(b) // "Take B".
  case (Right(b), Left(a))       => Right(b)
  case (Right(b1), Right(b2))    => Right(b1 |+| b2)
}, Left(implicitly[Monoid[A]].empty) ) // The type B does not need an empty element.
```

The monoid laws hold here because the proofs of the laws do not depend on the existence of e_B.

Example 8.4.1.4 (a routing monoid) A (much simplified) web server is implemented as a number of "routes". Each route may respond to one or more URL paths by evaluating a custom function. The task is to implement a `combine` operation for routes. The combined route should respond to all paths that at least one of the previous routes responds to:

```
type Path = String              //    Here, the types ‘Path‘ and ‘Response‘ are defined only as an
type Response = (Int, String)   // illustration. The code will use these types as type parameters.

type Route = Path => Option[Response]

val r1: Route = { case "/get_users" => (200, "user1, user2, user3") }
val r2: Route = { case "/get_names" => (200, "name1, name2, name3") }
      // The task is to implement an extension method |+| such that this works correctly:
```

```
val route: Route = r1 |+| r2     // Should respond to both '/get_users' and '/get_names'.
```

Use the `cats` library for implementing a `Monoid` instance for routes; verify that the monoid laws hold.

Solution We will first figure out how to implement the required functionality, and then adapt the code to the `cats` library's definition of the `Monoid` typeclass.

A "route" is a function of type `Path => Option[Response]` that returns a non-empty option if the route responds to a given path value. A combination of two routes `r1` and `r2` needs to be a new route, i.e., a function `Path => Option[Response]`. The new function will first check whether `r1` responds to a given path. If so, it will evaluate the result of applying `r1`. Otherwise, it will try applying `r2` to the given path. We implement the required business logic in a function called `combineRoutes`:

```
def combineRoutes(r1: Route, r2: Route): Route = { path =>
  r1(path) match {
    case Some(response) => Some(response)
    case None => r2(path)
  }
}
```

A monoid also needs to have an empty element. An "empty route" can be combined with any other route and will not change the behavior of that route. If the empty route responds to any path, it would prevent another route from also responding to the same path. So, the only solution is to define the "empty route" as a function that *never* responds to any path:

```
val emptyRoute: Route = { _ => None }
```

The `cats` library defines the `Monoid` typeclass via a `trait` with methods `empty` and `combine`. We can define that typeclass using our existing code:

```
import $ivy.`org.typelevel::cats-core:1.5.0`, cats.Monoid      // Using `ammonite` for convenience.
implicit val catsMonoidRoute: Monoid[Route] = new Monoid[Route] {
  def empty: Route = emptyRoute
  def combine(x: Route, y: Route): Route = combineRoutes(x, y)
}
```

We can now check that the routes can be combined as we intended:

```
import cats.syntax.monoid._
val route: Route = r1 |+| r2

ammonite@ route("/get_users")
res0: Response = (200, "user1, user2, user3")

ammonite@ route("/get_names")
res1: Response = (200, "name1, name2, name3")
```

To verify that the monoid laws hold, we could look for direct proofs using the code of `emptyRoute` and `combineRoutes`. However, it is easier to figure out how to reduce the definition of the `Route` monoid to a number of constructions listed in Section 8.3.4.

For more convenient reasoning, we replace the types `Path` and `Response` by type parameters P and R. We note that these types are used fully parametrically by the code of our functions. So, we will write the type `Route` as $\text{Route} \stackrel{\text{def}}{=} P \to \mathbb{1} + R$. Since `Route` is a function type at the outer level, we start with the function-type construction, which requires $\mathbb{1} + R$ to be a monoid. This suggests using the co-product construction; however, R is not necessarily a monoid. There are several possible monoid instances for $\mathbb{1} + R$, so let us look at how the code of `combineRoutes` handles values of that type. The value $1 + \mathbb{0}^{:R}$ corresponds to a route that is not responding to a given path; in that case, `combineRoutes` will switch to the other route. So, the empty value of the monoid $\mathbb{1} + R$ must be $1 + \mathbb{0}^{:R}$. This is indeed the value returned by `emptyRoute` when applied to any path.

A non-empty value $\mathbb{0} + r$ corresponds to a route that responds to a given path; given two such values, `combineRoutes` will take the first one. This corresponds to the binary operation $\oplus_R$ defined by $r_1 \oplus_R r_2 = r_1$. This operation makes R into a trivial semigroup (see Section 8.3.3). As Example 8.4.1.3 showed, $\mathbb{1} + R$ is a monoid if R is a semigroup (since the unit type $\mathbb{1}$ is a monoid).

So, we have reduced the monoid instance defined by `emptyRoute` and `combineRoutes` to a trivial semigroup, a co-product construction, and a function-type construction. Since all those constructions are guaranteed to produce lawful monoids, we do not need to prove the monoid laws by hand.

Example 8.4.1.5 Using the `cats` library, define a `Functor` instance for the type `Seq[Try[T]]`.

Solution The `cats` library defines the `Functor` typeclass as a `trait` with a `map` method:

```
trait Functor[F[_]] { def map[A, B](fa: F[A])(f: A => B): F[B] }
```

The functor `Seq[Try[T]]` is a composition of functors `Seq` and `Try`, so we can define the `map` method for it using the functor composition (see Statement 6.2.3.6). We wrap `Seq[Try[T]]` into a case class:

```
import cats.Functor
final case class F[T](s: Seq[Try[T]]) // ‘type F[T] = Seq[Try[T]]‘ does not work with ‘map‘ method.
implicit val functorF: Functor[F] = new Functor[F] {
  def map[A, B](fa: F[A])(f: A => B): F[B] = F(fa.s.map(_.map(f)))
}
```

One more `cats`-specific import is necessary to enable the `map` extension method for functors:

```
import cats.syntax.functor._      // Enable the ‘map‘ method.
val s = F(Seq(Try(1), Try(2), Try(3)))

ammonite@ s.map(_ * 10)
res0: F[Int] = F(List(Success(10), Success(20), Success(30)))
```

For the `map` method to work, the type constructor `F` must be defined as a `class` or a `trait`. Defining `F` as a type alias would make the Scala compiler confused: if the type `F[T]` were *the same* as `Seq[Try[T]]`, the expression `Seq(Try(1)).map(_ * 10)` would mean to apply `Seq`'s built-in `map` method rather than the extension method `map` defined for the type constructor `F` via the `Functor` typeclass.

Example 8.4.1.6 Using the `cats` library, implement a `Bifunctor` instance for $Q^{X,Y} \stackrel{\text{def}}{=} X + X \times Y$.

Solution The `cats` library defines the `Bifunctor` typeclass as a `trait` with the `bimap` method. Implementing that method as an information-preserving, fully parametric function is straightforward:

```
final case class Q[X, Y](q: Either[X, (X, Y)])
implicit val bifunctorQ = new Bifunctor[Q] {
  def bimap[A, B, C, D](fab: Q[A, B])(f: A => C, g: B => D): Q[C, D] = fab.q match {
    case Left(a)          => Q(Left(f(a)))
    case Right((a, b))    => Q(Right((f(a), g(b))))
  }
}
```

Example 8.4.1.7 Define a `Contrafunctor` typeclass having the method `contramap` (see Section 6.1.7):

```
def contramap[A, B](c: C[A])(f: B => A): C[B]
```

Implement a `Contrafunctor` instance for the type constructor $C^A \stackrel{\text{def}}{=} A \to \text{Int}$.

Solution An instance value must contain the `contramap` method of type $\forall(A, B).\, C^A \to (B \to A) \to C^B$. This signature has type parameters, so the typeclass needs to be implemented as a `trait`:

```
trait Contrafunctor[C[_]] { def contramap[A, B](c: C[A])(f: B => A): C[B] }
```

A typeclass instance for the type constructor $C^A \stackrel{\text{def}}{=} A \to \text{Int}$ is created by

```
type C[A] = A => Int
implicit val contrafunctorC = new Contrafunctor[C] {
  def contramap[A, B](c: A => Int)(f: B => A): B => Int = f andThen c
}
```

The `cats` library defines an equivalent typeclass named `Contravariant` with the method `contramap`.

Example 8.4.1.8 Define a `Functor` instance for recursive type constructor $Q^A \stackrel{\text{def}}{=} (\text{Int} \to A) + \text{Int} + Q^A$.

Solution Begin by defining Q^A as a recursive disjunctive type:

```
sealed trait Q[A]
final case class C1[A](i: Int => A) extends Q[A]
```

```
final case class C2[A](x: Int)      extends Q[A]
final case class C3[A](q: Q[A])     extends Q[A]
```

The methods of Section 6.2.3 show how to implement the `map` function for `Q[A]`. Without repeating those steps, we will write code for a `Functor` instance directly:

```
implicit val functorQ: Functor[Q] = new Functor[Q] { // The function ‘map‘ is recursive.
  def map[A, B](qa: Q[A])(f: A => B): Q[B] = qa match {
    case C1(i)   => C1(i andThen f)
    case C2(x)   => C2(x)
    case C3(q)   => C3(map(q)(f)) // Recursive case.
  }
}
```

Example 8.4.1.9 Using a function parameterized by the type constructors F and G (required to be functors), implement a `Functor` instance for $F^A + G^A$.

Solution The co-product construction (Statement 6.2.3.4) shows how to implement the `map` function for the functor $F^A + G^A$. We begin by writing code for a `Functor` instance assuming that the type constructors `F` and `G` are given:

```
type L[A] = Either[F[A], G[A]]
implicit val functorEither = new Functor[L] {
  def map[A, B](e: L[A])(f: A => B): L[B] = e match {
    case Left(fa)    => Left(fa.map(f))
    case Right(ga)   => Right(ga.map(f))
  }
}
```

We will now to rewrite this code by making `F` and `G` into type parameters. To achieve that, we need to use the "kind projector" plugin and replace the type constructor `L` by a nameless type function `Lambda[X => Either[F[X], G[X]]]`. The code becomes:

```
implicit def functorEither[F[_], G[_]] = new Functor[Lambda[X => Either[F[X], G[X]]]] {
  type L[A] = Either[F[A], G[A]]         // We may use F and G to define a type alias in this scope.
  def map[A, B](e: L[A])(f: A => B): L[B] = e match {
    case Left(fa)    => Left(fa.map(f))
    case Right(ga)   => Right(ga.map(f))
  }
}
```

Example 8.4.1.10* **(a)** Implement a function with type signature $C^A + C^B \to C^{A\times B}$ parameterized by a type constructor C (required to be a contrafunctor) and by arbitrary types A, B. Show that the inverse type signature $C^{A\times B} \to C^A + C^B$ is not implementable for some contrafunctors C.

(b) Implement a function with type signature $F^{A\times B} \to F^A \times F^B$ parameterized by a type constructor F (required to be a functor) and by arbitrary types A, B. Show that the inverse type signature $F^A \times F^B \to F^{A\times B}$ is not implementable for some functors F.

Solution **(a)** We need to implement a function with type signature

$$\forall(A, B).\, C^A + C^B \to C^{A\times B} \quad .$$

Begin by looking at the types involved. We need to relate values $C^{A\times B}$, C^A, and C^B; can we relate $A \times B$, A, and B? There exist unique fully parametric functions π_1 and π_2 of types $A \times B \to A$ and $A \times B \to B$. If we lift these functions to the contrafunctor C, we will get $\pi_1^{\downarrow C} : C^A \to C^{A\times B}$ and $\pi_2^{\downarrow C} : C^B \to C^{A\times B}$. The required type signature is then implemented via a `match` expression like this:

```
def f[C[_]: Contrafunctor, A, B]
                : Either[C[A], C[B]] => C[(A, B)] = {
  case Left(ca)    => ca.contramap { case (a, b) => a }
  case Right(cb)   => cb.contramap { case (a, b) => b }
}
```

The code notation for this function is:

$$f^{:C^A+C^B\to C^{A\times B}} \stackrel{\text{def}}{=} \begin{array}{|c||c|} & C^{A\times B} \\ \hline C^A & \pi_1^{\downarrow C} \\ C^B & \pi_2^{\downarrow C} \end{array} \quad .$$

To show that it is not possible to implement a function g with the inverse type signature:

$$g : \forall(A, B).\, C^{A\times B} \to C^A + C^B \quad ,$$

we choose the contrafunctor $C^A \stackrel{\text{def}}{=} A \to R$, where R is a fixed type. The type signature becomes:

$$g : \forall(A, B).\, (A \times B \to R) \to (A \to R) + (B \to R) \quad .$$

To implement this function, we need to decide whether to return values of type $A \to R$ or $B \to R$. Can we compute a value of type $A \to R$ given a value of type $A \times B \to R$?

$$g^{:(A\times B\to R)\to A\to R} \stackrel{\text{def}}{=} q^{:A\times B\to R} \to a^{:A} \to ???^{:R}$$

We cannot compute a value of type R because that requires us to apply the function q to a pair $A \times B$, while we only have a value of type A. So, the typed hole $???^{:R}$ cannot be filled.

Similarly, we are not able to compute a value of type $B \to R$ from a value of type $A \times B \to R$. Whatever choice we make, $A \to R$ or $B \to R$, we cannot implement the required type signature.

(b) We need to implement a function with type signature:

$$\forall(A, B).\, F^{A\times B} \to F^A \times F^B \quad .$$

The types $A \times B$, A, and B are related by the functions $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$. Lifting these functions to the functor F, we obtain $\pi_1^{\uparrow F} : F^{A\times B} \to F^A$ and $\pi_2^{\uparrow F} : F^{A\times B} \to F^B$. It remains to take the product of the resulting values:

$$f^{:F^{A\times B}\to F^A\times F^B} \stackrel{\text{def}}{=} p^{:F^{A\times B}} \to (p \triangleright \pi_1^{\uparrow F}) \times (p \triangleright \pi_2^{\uparrow F}) \quad .$$

```
def f[F[_]: Functor, A, B](p: F[(A, B)]): (F[A], F[B]) =
  (p.map { case (a, b) => a }, p.map { case (a, b) => b })  // Or (p.map(_._1), p.map(_._2))
```

A shorter code for f via the "diagonal" function $\Delta \stackrel{\text{def}}{=} (q^{:Q} \to q \times q)$ and the pair product $\boxtimes$ is:

$$f^{:F^{A\times B}\to F^A\times F^B} \stackrel{\text{def}}{=} \Delta \,\fatsemi\, (\pi_1^{\uparrow F} \boxtimes \pi_2^{\uparrow F}) \quad .$$

This notation is sometimes easier to reason about when deriving properties of functions.

If we try implementing a function g with the inverse type signature:

$$g : \forall(A, B).\, F^A \times F^B \to F^{A\times B} \quad , \tag{8.17}$$

we will find that g *can* be implemented for functors such as $F^A \stackrel{\text{def}}{=} A \times A$, $F^A \stackrel{\text{def}}{=} \mathbb{1} + A$, and $F^A \stackrel{\text{def}}{=} P \to A$. It is not obvious how to find a functor F for which the function g has no implementation. By looking through the known functor constructions (Table 6.2) and trying various combinations, we eventually find a suitable functor: $F^A \stackrel{\text{def}}{=} (P \to A) + (Q \to A)$. The type signature of g becomes:

$$g : \forall(A, B).\, ((P \to A) + (Q \to A)) \times ((P \to B) + (Q \to B)) \to (P \to A \times B) + (Q \to A \times B) \quad .$$

The argument of the function g has the following type:

$$((P \to A) + (Q \to A)) \times ((P \to B) + (Q \to B)) \quad ,$$

which can be transformed equivalently into a disjunction of four cases:

$$(P \to A) \times (P \to B) + (P \to A) \times (Q \to B) + (Q \to A) \times (P \to B) + (Q \to A) \times (Q \to B) \quad .$$

Implementing the function g requires, in particular, to handle the case when we are given values of types $P \to A$ and $Q \to B$, and we are required to produce a value of type $(P \to A \times B) + (Q \to A \times B)$. The resulting type signature is:

$$(P \to A) \times (Q \to B) \to (P \to A \times B) + (Q \to A \times B)$$

and cannot be implemented by fully parametric code. Indeed, to return a value of type $P \to A \times B$, we would need to produce a pair of type $A \times B$ from a value of type P. But the given arguments have types $P \to A$ and $Q \to B$, so producing a pair $A \times B$ requires us to have also a value of type Q, which we do not have. Similarly, we cannot return a value of type $Q \to A \times B$.

We find that the function g cannot be implemented for the functor $F^A \stackrel{\text{def}}{=} (P \to A) + (Q \to A)$. Functors F for which the type signature (8.17) *can* be implemented are called "applicative" (see Chapter 11 for precise conditions). The functor F is an example of a functor that is not applicative.

Example 8.4.1.11* (R. O'Connor [7]) Assume that a functor F admits a function p with type signature:

```
def p[A, B, F[_]: Functor]: Either[A, F[B]] => F[Either[A, B]]
```

$$p^{A,B} : A + F^B \to F^{A+B} \quad ,$$

additionally satisfying the special laws of identity and associativity:

$$p^{\mathbb{0},B} = (b^{:B} \to \mathbb{0} + b)^{\uparrow F} \quad , \qquad p^{A+B,C} = \left|\begin{array}{c||cc} & A & F^{B+C} \\ \hline A & \text{id} & \mathbb{0} \\ B + F^C & \mathbb{0} & p^{B,C} \end{array}\right| ⨾ p^{A,B+C} \quad .$$

Show that the functor F is *pointed* if such a function p exists. Then show that *any* pointed functor F admits a function p with these properties. So, the given conditions for the function p are just a more complicated but equivalent definition of the "pointed functor" typeclass.

Solution To show that F is pointed, it is sufficient to find a value wu_F of type $F^{\mathbb{1}}$. We note that the given function p can create values of type F^{A+B} from input values of type $A + \mathbb{0}$. So, we set the type parameters $A = \mathbb{1}$ and $B = \mathbb{0}$ and apply p to the value $1 + \mathbb{0}^{:F^0}$:

$$\text{wu}_F \stackrel{\text{def}}{=} (1 + \mathbb{0}^{:F^0}) ▷ p^{\mathbb{1},\mathbb{0}} ▷ (1 + \mathbb{0} \to 1)^{\uparrow F} \quad .$$

```
val wu[F[_]: Functor]: F[Unit] = p[Unit, Nothing](Left(())).map { case Left(_) => () }
```

Conversely, assuming that F is pointed, we use its pu_F function to define p as:

```
def p[F[_]: Functor : Pointed, A, B]
              : Either[A, F[B]] => F[Either[A, B]] = {
  case Left(a)     => pure[F, Either[A, B]](Left(a))
  case Right(fb)   => fb.map { b => Right[A, B](b) }
}
```

$$p^{A,B} \stackrel{\text{def}}{=} \left|\begin{array}{c||c} & F^{A+B} \\ \hline A & (a^{:A} \to a + \mathbb{0}^{:B}) ⨾ \text{pu}_F \\ F^B & (b^{:B} \to \mathbb{0}^{:A} + b)^{\uparrow F} \end{array}\right| \quad .$$

It remains to show that p satisfies the required laws. The identity law holds because:

$$\text{expect to equal } (b^{:B} \to \mathbb{0} + b)^{\uparrow F} : \quad p^{\mathbb{0},B} = \left|\begin{array}{c||c} & F^{\mathbb{0}+B} \\ \hline \mathbb{0} & \text{(we may delete this line)} \\ F^B & (b^{:B} \to \mathbb{0} + b)^{\uparrow F} \end{array}\right| = (b^{:B} \to \mathbb{0} + b)^{\uparrow F} \quad .$$

To verify the associativity law, we begin with its right-hand side since it is more complicated:

$$\left|\begin{array}{c||cc} & A & F^{B+C} \\ \hline A & \text{id} & \mathbb{0} \\ B + F^C & \mathbb{0} & p^{B,C} \end{array}\right| ⨾ p^{A,B+C} = \left|\begin{array}{c||cc} & A & F^{B+C} \\ \hline A & \text{id} & \mathbb{0} \\ B + F^C & \mathbb{0} & p^{B,C} \end{array}\right| ⨾ \left|\begin{array}{c||c} & F^{A+B+C} \\ \hline A & (a^{:A} \to a + \mathbb{0}^{:B+C}) ⨾ \text{pu}_F \\ F^{B+C} & (x^{:B+C} \to \mathbb{0}^{:A} + x)^{\uparrow F} \end{array}\right|$$

$$= \left|\begin{array}{c||c} & F^{A+B+C} \\ \hline A & (a^{:A} \to a + \mathbb{0}^{:B+C}) ⨾ \text{pu}_F \\ B + F^C & p^{B,C} ⨾ (x^{:B+C} \to \mathbb{0}^{:A} + x)^{\uparrow F} \end{array}\right| = \left|\begin{array}{c||c} & F^{A+B+C} \\ \hline A & (a^{:A} \to a + \mathbb{0}^{:B+C}) ⨾ \text{pu}_F \\ B & (b^{:B} \to b + \mathbb{0}^{:C}) ⨾ \text{pu}_F ⨾ (x^{:B+C} \to \mathbb{0}^{:A} + x)^{\uparrow F} \\ F^C & (c^{:C} \to \mathbb{0}^{:A+B} + c)^{\uparrow F} \end{array}\right| \quad .$$

[7] This example is based on the post https://mail.haskell.org/pipermail/haskell-cafe/2015-November/122357.html

In the last line, we have expanded the type matrix to three rows corresponding to the disjunctive type $A + B + F^C$. We need to show that the last matrix equals $p^{A+B,C}$; so let us rewrite $p^{A+B,C}$ as a similarly expanded type matrix, using the type isomorphisms such as $\mathbb{0}^{:A} + \mathbb{0}^{:B} \cong \mathbb{0}^{:A+B}$:

$$p^{A+B,C} = \left| \begin{array}{c} \\ A \\ B \\ F^C \end{array} \begin{array}{c} F^{A+B+C} \\ \hline (a^{:A} \to a + \underline{\mathbb{0}^{:B} + \mathbb{0}^{:C}}) \,{}^{;}\!\mathrm{pu}_F \\ (b^{:B} \to \mathbb{0}^{:A} + b + \mathbb{0}^{:C}) \,{}^{;}\!\mathrm{pu}_F \\ (c^{:C} \to \underline{\mathbb{0}^{:A} + \mathbb{0}^{:B}} + c)^{\uparrow F} \end{array} \right| = \left| \begin{array}{c} \\ A \\ B \\ F^C \end{array} \begin{array}{c} F^{A+B+C} \\ \hline (a^{:A} \to a + \mathbb{0}^{:B+C}) \,{}^{;}\!\mathrm{pu}_F \\ (b^{:B} \to \mathbb{0}^{:A} + b + \mathbb{0}^{:C}) \,{}^{;}\!\mathrm{pu}_F \\ (c^{:C} \to \mathbb{0}^{:A+B} + c)^{\uparrow F} \end{array} \right| .$$

The only remaining difference is in the second lines of the matrices. We write those lines separately:

$$\begin{aligned} \text{expect to equal } (b^{:B} \to \mathbb{0}^{:A} + b + \mathbb{0}^{:C}) \,{}^{;}\!\mathrm{pu}_F : \quad & (b^{:B} \to b + \mathbb{0}^{:C}) \,{}^{;}\!\underline{\mathrm{pu}_F \,{}^{;}\!(x^{:B+C} \to \mathbb{0}^{:A} + x)^{\uparrow F}} \\ \text{naturality law of } \mathrm{pu}_F : \quad & = \underline{(b^{:B} \to b + \mathbb{0}^{:C}) \,{}^{;}\!(x^{:B+C} \to \mathbb{0}^{:A} + x)} \,{}^{;}\!\mathrm{pu}_F \\ \text{compute function composition} : \quad & = (b^{:B} \to \mathbb{0}^{:A} + b + \mathbb{0}^{:C}) \,{}^{;}\!\mathrm{pu}_F \quad . \end{aligned}$$

This completes the proof of the required laws.

8.4.2 Exercises

Exercise 8.4.2.1 Define a PTVF `def isLong[T]: Boolean` that returns `true` for `T = Long` or `Double` and returns `false` for `T = Int`, `Short`, or `Float`. The function should remain undefined for other types `T`.

Exercise 8.4.2.2 Implement a `Monoid` instance for the type $\text{String} \times (\mathbb{1} + \text{Int})$.

Exercise 8.4.2.3 **(a)** If A is a monoid and R any type, implement a `Monoid` instance for $R \to A$:

```
def monoidFunc[A: Monoid, R]: Monoid[R => A] = ???
```

Prove that the monoid laws hold for that instance.

(b) With the choice `R = Boolean`, use the type equivalence $(R \to A) = (\mathbb{2} \to A) \cong A \times A$ and verify that the monoid instance `monoidFunc[A, Boolean]` is the same as the monoid instance for $A \times A$ computed by `monoidPair[A, A]` in Section 8.3.4.

Exercise 8.4.2.4 Show that if `S` is a semigroup then `Option[S]` and `Option[(S, S)]` are monoids.

Exercise 8.4.2.5 A framework implements a "route" type R as $R \stackrel{\text{def}}{=} Q \to (E + S)$, where Q is a query, E is an error response, and S is a success response. A server is defined as a combination of several routes. For a given query Q, the response is the first route (if it exists) that yields a success response. Implement the route combination operation and show that it makes R into a semigroup. What would be necessary to make R into a monoid?

Exercise 8.4.2.6 Using the `cats` library, implement a `Functor` instance for `type F[T] = Try[Seq[T]]`.

Exercise 8.4.2.7 Using the `cats` library, implement a `Bifunctor` instance for $B^{X,Y} \stackrel{\text{def}}{=} (\text{Int} \to X) + Y \times Y$.

Exercise 8.4.2.8 Define a `Profunctor` typeclass having the method `xmap`:

```
def xmap[A, B](f: A => B, g: B => A): F[A] => F[B]
```

Implement a `Profunctor` instance for $P^A \stackrel{\text{def}}{=} A \to (\text{Int} \times A)$.

Exercise 8.4.2.9 Implement a `Functor` instance for the recursive type $Q^A \stackrel{\text{def}}{=} \text{String} + A \times A \times Q^A$.

Exercise 8.4.2.10 Show explicitly that a value $\mathrm{wu}_C : C^{\mathbb{1}}$ is computationally equivalent to a value $\mathrm{pu}_C : \forall A. C^A$ that satisfies the naturality law (8.15).

Exercise 8.4.2.11 Using a function parameterized by the type constructors F and G (required to be functors), implement a `Functor` instance for $F^A \times G^A$.

Exercise 8.4.2.12 Implement a `Functor` instance for $F^A \to G^A$ as a function parameterized by type constructors F and G, where F^A is required to be a contrafunctor and G^A is required to be a functor. For the contrafunctor F, use either the `Contrafunctor` typeclass from Example 8.4.1.7 or the `cats` library's typeclass `Contravariant`.

Exercise 8.4.2.13* **(a)** Implement a function with type signature $F^A + F^B \to F^{A+B}$ parameterized by a type constructor F (required to be a functor) and by arbitrary types A, B. Show that the inverse type signature $F^{A+B} \to F^A + F^B$ is not implementable for some functors F.

(b) Implement a function with type signature $C^{A+B} \to C^A \times C^B$ parameterized by a type constructor C (required to be a contrafunctor) and by arbitrary types A, B. Show that the inverse type signature $C^A \times C^B \to C^{A+B}$ is not implementable for some contrafunctors C.

Exercise 8.4.2.14* Implement a function with type signature $F^{A\to B} \to A \to F^B$ parameterized by a functor F and arbitrary types A, B. Show that the inverse type signature, $(A \to F^B) \to F^{A\to B}$, cannot be implemented for some functors F. (Functors admitting a function with that type signature are called "rigid"; see Section 14.9.3.)

Exercise 8.4.2.15* (dual O'Connor) Assume that a functor F admits a function q with type signature $q^{A,B} : F^{A\times B} \to A \times F^B$, additionally satisfying the special laws of identity and associativity:

```
def q[A, B, F[_]: Functor]: F[(A, B)] => (A, F[B])
```

$$q^{\mathbb{1},B} = f^{:F^{\mathbb{1}\times B}} \to 1 \times (f \triangleright (1 \times b^{:B} \to b)^{\uparrow F}) \quad , \qquad q^{A,B\times C} \fatsemi (\mathrm{id}^A \boxtimes q^{B,C}) = q^{A\times B,C} \quad .$$

This Scala code illustrates the required laws in the programming language syntax:

```
      // For any value f: F[(Unit, B)], we first compute
val (a1, b1) = q[Unit, B, F](f)._1
      // Then we must have:    a1 == ()   and   b1 == f.map { case (_, b) => b }

      // For any value p: F[(A, B, C)], we first compute
val ((a1: A, b1: B), f1: F[C]) = q[(A, B), C, F](p.map { case (a, b, c) => ((a, b), c) })
val (a2: A, f2: F[(B, C)]) = q[A, (B, C), F](p.map { case (a, b, c) => (a, (b, c)) }
      // Then we must have:    a1 == a2   and   (b1, f1) == p[B, C, F](f2)
```

The following naturality law should also hold for q:

$$(f^{:A\to C} \boxtimes g^{:B\to D})^{\uparrow F} \fatsemi q^{C,D} = q^{A,B} \fatsemi f \boxtimes (g^{\uparrow F}) \quad .$$

```
      // For any value k: F[(A, B)] and any functions f: A => C, g: B => D, first compute
val (a: A, fb: F[B]) = q[A, B, F](k)  // Then we must have:
(f(a), fb.map(g)) == q[C, D, F](k.map { case (a, b) => (f(a), g(b)) })
```

Show that the functor F is *co-pointed* if a function q with these properties exists. Show that *any* lawful co-pointed functor F has a function q with these properties. So, the given conditions for the function q are just a more complicated but equivalent definition of the "co-pointed functor" typeclass.

8.5 Further developments

8.5.1 The existence of values for recursive types

A recursive type T is defined by a type equation such as $T \stackrel{\text{def}}{=} \mathbb{1} + \text{Int} + T \times T$. Can we decide whether such a type equation has a "solution", i.e., a well-defined type T with values that we can create and manipulate in a program?

In all examples seen so far, the recursive type equations had the form $T \stackrel{\text{def}}{=} S^T$ where the type constructor S is a *functor*. Type equations with non-functor S (e.g., the equation $T \stackrel{\text{def}}{=} T \to \text{Int}$) do not seem to be useful in practice, and we will not consider them in this book.

In a rigorous approach, showing that T is a "solution" (called a **fixpoint**) of the type equation $T \stackrel{\text{def}}{=} S^T$ means proving that the types T and S^T are equivalent (isomorphic). We must implement this type isomorphism as two functions, named e.g., `fix` and `unfix`, satisfying the conditions:

$$\text{fix} : S^T \to T \quad , \qquad \text{unfix} : T \to S^T \quad , \qquad \text{fix} \fatsemi \text{unfix} = \text{id} \quad , \qquad \text{unfix} \fatsemi \text{fix} = \text{id} \quad .$$

Given a type constructor S, we can define the recursive type T with this Scala code:

```
final case class T(s: S[T])                // Type constructor S[_] must be already defined.
def fix:    S[T] => T   =   { s => T(s) }
def unfix:  T => S[T]   =   { t => t.s }
```

We can generalize this code to a "fixpoint" constructor Fix^S that is *parameterized* by S:

```
final case class Fix[S[_]](s: S[Fix[S]])
def fix[S[_]]:    S[Fix[S]] => Fix[S]   =   { s => Fix(s) }
def unfix[S[_]]:  Fix[S] => S[Fix[S]]   =   { t => t.s }
```

In both implementations, the functions `fix` and `unfix` are inverses of each other because they merely wrap and unwrap values within a case class. So, we are always able to write code that *defines* the recursive type T. The remaining question is whether we will be able to create values of type T.

Applying structural analysis to the functor S, we begin with the constant functor $S^A \stackrel{\text{def}}{=} Z$, where Z is a fixed type. The type equation $T \stackrel{\text{def}}{=} S^T$ is just $T \stackrel{\text{def}}{=} Z$; so, T is not a recursive type.

The next case is the identity functor $S^A \stackrel{\text{def}}{=} A$. The type equation $T \stackrel{\text{def}}{=} S^T$ has the form $T \stackrel{\text{def}}{=} T$; since all types T satisfy that equation trivially, we find that the identity functor does not define any specific type T. If we translate this type equation into Scala code, we will run into a problem with recursion:

```
type Id[A] = A
final case class T(s: Id[T])
// Equivalent to 'final case class T(s: T)'.
val x = T(x)                // Infinite loop!
```

A value of `case class T` can be created only if we supply a value of type `T`. Writing `def x = T(x)` instead of `val` does not help: the evaluation of the code for `x` will not terminate. We conclude that this recursive definition is invalid: one cannot create values of the resulting type `T`. The type `T` is void.

Next, consider a product functor such as $S^A \stackrel{\text{def}}{=} A \times A \times \text{Int}$. Can we create values of type $T \stackrel{\text{def}}{=} S^T$?

```
final case class T(s: (T, T, Int))
val x = T(s = (x, x, 123))  // Infinite loop!
```

We again have an infinite loop when creating values of type `T`. As in the example with the identity functor, the case class `T` requires us to compute some values of type `T` before we can create a value of type `T`. That requirement is impossible to satisfy.

For some disjunctive type constructors $S^\bullet$, values of type Fix^S are created with no difficulty. One example is $S^A \stackrel{\text{def}}{=} \text{Int} + A \times A$:

```
final case class T(s: Either[Int, (T, T)])
val x: T = T(Left(123))     // OK
val y: T = T(Right(x, x))   // OK
```

We are able to create `x` of type $S^{\mathbb{0}} \stackrel{\text{def}}{=} \text{Int} + \mathbb{0}^{:T\times T}$ without need for any previous values of T. We can then use `x` to create a value `y` of type T. This resembles defining a value by induction: the base case is the type $\text{Int} + \mathbb{0}^{:T\times T}$, which is a disjunctive part of S^A that does *not* contain any values of type A. The inductive step is the type $\mathbb{0}^{:\text{Int}} + T \times T$, which creates a new value of type T from two previous values. Type recursion terminates when the base case exists.

The examples we saw previously, $S^A \stackrel{\text{def}}{=} A$ and $S^A \stackrel{\text{def}}{=} \text{Int} \times A \times A$, do not have a base case where a value of type S^A could be computed without need for any previous values of type A.

Given a functor S^A, how can we determine whether the type recursion $T \stackrel{\text{def}}{=} S^T$ terminates? If S^A is a *polynomial* functor, we can view S^A as a polynomial function of A and reduce it to the form:

$$S^A \cong C_0 + C_1 \times A + C_2 \times A \times A + \ldots \tag{8.18}$$

It is clear from this formula that the type C_0 is found by setting $A = \mathbb{0}$ in S^A, i.e., $C_0 \cong S^{\mathbb{0}}$. If we can compute a value c_0 of type C_0 (i.e., if C_0 is not void, $C_0 \ncong \mathbb{0}$), we get a base-case value t_0 of type T as:

$$t_0^{:T} \stackrel{\text{def}}{=} c_0^{:C_0} + \mathbb{0}^{:C_1\times A} + \mathbb{0}^{:C_2\times A\times A} + \ldots$$

Further values of type T could be computed as $c_1 \times t_0$ or $c_2 \times t_0 \times t_0$, etc.

We conclude that the recursive type $T \stackrel{\text{def}}{=} S^T$ defined via a *polynomial* functor S will be non-void (i.e., type recursion will terminate) when the type $S^{\mathbb{0}}$ is non-void.

In the examples $S^A \stackrel{\text{def}}{=} A$ and $S^A \stackrel{\text{def}}{=} \text{Int} \times A \times A$, we find $S^{\mathbb{0}} \cong \mathbb{0}$, so the type recursion is invalid.

It remains to consider functors S that cannot be reduced to the polynomial form (8.18). As an example, take $S^A \stackrel{\text{def}}{=} \text{String} \times (\text{Int} \to A)$. For this functor S, the condition $S^{\mathbb{0}} \not\cong \mathbb{0}$ does not hold:

$$\begin{aligned} & S^{\mathbb{0}} \cong \text{String} \times (\underline{\text{Int} \to \mathbb{0}}) \\ \text{use the type equivalence } (\text{Int} \to \mathbb{0}) \cong \mathbb{0}: \quad & \cong \text{String} \times \mathbb{0} \cong \mathbb{0} \quad . \end{aligned}$$

Nevertheless, we can write Scala code implementing values of the type T defined by $T \stackrel{\text{def}}{=} S^T$:

```
final case class T(message: String, next: Int => T)
val t: T = T("start", n => T(s"have $n", _ => t))   // A recursive ‘val‘.
```

The value `t` refers to itself recursively within a nested function `_ => t`, and there is no "base case" in the type S^T. However, the recursion does not lead to an infinite loop; we can use `t` safely:

```
scala> t.next(10)
res0: T = T(have 10,<function1>)

scala> t.next(10).next(10)
res1: T = T(start,<function1>)
```

A value of type T contains a function `next` that, when applied, returns a new value of the same type T. The new value of type T does not need to be computed in advance: its evaluation is *delayed* until some code decides to call `next`. For this reason, an infinite loop is avoided even though the structure functor S has no "base case". Values of type T can be viewed as an infinite stream of `String` values computed on demand. An `Int` value is required in order to produce the next element of the stream. A symbolic (and non-rigorous) representation of that type is:

$$T = \text{String} \times (\text{Int} \to \text{String} \times (\text{Int} \to \text{String} \times (\text{Int} \to \text{String} \times ...))) \quad .$$

As another example, consider $S^A \stackrel{\text{def}}{=} \mathbb{1} \to \text{String} + \text{Int} \times A$. Using the type equivalence $P \cong (\mathbb{1} \to P)$, we could transform S^A into an equivalent functor $\tilde{S}^A$ in the polynomial form (8.18):

$$\tilde{S}^A \stackrel{\text{def}}{=} \text{String} + \text{Int} \times A \quad .$$

Although the types S^A and $\tilde{S}^A$ are equivalent, the recursive types Fix^S and $\text{Fix}^{\tilde{S}}$ are different. While

$$\begin{aligned} \text{Fix}^{\tilde{S}} &= \text{String} + \text{Int} \times (\text{String} + \text{Int} \times (\text{String} + \text{Int} \times ...)) \\ &\cong (\mathbb{1} + \text{Int} + \text{Int} \times \text{Int} + \text{Int} \times \text{Int} \times \text{Int} + ...) \times \text{String} \cong \text{String} \times \text{List}^{\text{Int}} \quad , \end{aligned}$$

the type Fix^S admits recursively defined values representing *unbounded* streams of integers, e.g.:

```
final case class T(e: () => Either[String, (Int, T)]) // Define the type T ≝ String + Int × T.
val t1: T = T(() => Right((1, t1)))                   // Stream [1, 1, 1, ...].
def t2(n: Int): T = T(() => Right((n, t2(n+1))))      // Stream [n, n+1, n+2, ...].
```

The type $T \stackrel{\text{def}}{=} \text{Fix}^S$ also admits finite streams that may be defined as *non-recursive* values of type T:

```
val t0 = T(() => Right((10, T(() => Left("stop")))))  // Finite stream [10, "stop"].
```

We can recognize that Fix^S has non-recursive values by checking that the type $S^{\mathbb{0}}$ is not void:

$$S^{\mathbb{0}} = \mathbb{1} \to \text{Int} \times \mathbb{0} + \text{String} \cong \mathbb{1} \to \text{String} \cong \text{String} \not\cong \mathbb{0} \quad .$$

How can we recognize functors S that admit valid recursive values of type Fix^S? The exponential functor construction (Statement 6.2.3.5) shows that $C^A \to P^A$ is a functor when C is a contrafunctor and P is a functor. If the type expression for S^A contains a sub-expression of the form $C^A \to P^A$, how can we implement a value of type $C^T \to P^T$? We can use recursion to implement a value of T, as long as we can then somehow produce a value of type P^T out of a value of type T. This is precisely the condition for $P^\bullet$ to be a pointed functor. The contrafunctor C^A is an argument of the function of

type $C^A \to P^A$, so we are not required to produce values of type C^A— we *consume* those values. It follows that we can implement a value of type $C^T \to P^T$ (with $T = \text{Fix}^S$) as long as $P^\bullet$ is a pointed functor. As we saw in Section 8.3.5, a functor $P^\bullet$ is pointed if we can compute a value of type $P^{\mathbb{1}}$.

In the example $S^A \stackrel{\text{def}}{=} \text{String} \times (\text{Int} \to A)$, the functor S^A is pointed since:

$$S^{\mathbb{1}} \cong \text{String} \times (\text{Int} \to \mathbb{1}) \cong \text{String} \not\cong \mathbb{0} \quad .$$

This consideration applies to any sub-expression of the form $C^A \to P^A$ within the type constructor S^A. The condition for values of Fix^S to exist is that every functor P^A involved in such sub-expressions should be pointed. To check that, we can set the type parameter $A = \mathbb{1}$ in all return types of *functions* within S^A. If the resulting type is not void, we will be able to implement a recursively defined value of type Fix^S.

If the functor S^A has the property $S^{\mathbb{0}} \not\cong \mathbb{0}$ (i.e., we have a base case for the inductive definition of Fix^S), we will also be able to implement non-recursive values of type Fix^S.

We conclude that $S^{\mathbb{0}} \not\cong \mathbb{0}$ is a *sufficient* condition for the type Fix^S to be non-void. If S is a polynomial functor, this condition is also a necessary condition. For exponential-polynomial functors S, recursive values of type Fix^S can be implemented if every function-type sub-expression $C^A \to P^A$ in S^A involves a pointed functor P (i.e., if $P^{\mathbb{1}} \not\cong \mathbb{0}$).

8.5.2 Proofs of associativity of `concat` for lists and arrays

The `concat` function is defined for both lists and arrays, and works similarly for these data types:

```
scala> Array.concat(Array(1, 2), Array(3, 4), Array(5, 6))
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6)

scala> List.concat(List(1, 2), List(3, 4), List(5, 6))
res1: List[Int] = List(1, 2, 3, 4, 5, 6)
```

In this section, we will show rigorously that concatenation is an associative operation.

In Scala, `Array[A]` is a sequence whose elements can be accessed by index. The array access function (the `apply` method defined on the `Array` class) is a partial function of type `Int => A` whose integer argument must be between 0 and $n-1$, where n is the array length:

```
val x = Array("a", "b", "c")

scala> x(2)  // The syntax ‘x(2)‘ is the same as ‘x.apply(2)‘.
res2: String = c

scala> x(3)  // Applying the partial function ‘x.apply‘ to the value 3 will fail:
java.lang.ArrayIndexOutOfBoundsException: 3
```

We can denote the type of this function by $\text{Int}_{[0,n-1]} \to A$ to indicate the bounds of the index.

The `List` type constructor is a recursive disjunctive type defined by the type equation:

$$\text{List}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times \text{List}^A = \mathbb{1} + A + A \times A + A \times A \times A + ...$$

Although the definitions of types are different, lists and arrays are conceptually similar — they are both sequences of values of type A. However, proving properties is easier for non-recursive types than for recursive types. We begin by proving the associativity property of the array concatenation.

Statement 8.5.2.1 For arrays, $\text{Array}^A_n \stackrel{\text{def}}{=} \text{Int}_{[0,n-1]} \to A$, the `concat` function (denoted $+\!\!+$) defined by:

$$a_1^{:\text{Array}^A_{n_1}} +\!\!+ a_2^{:\text{Array}^A_{n_2}} \stackrel{\text{def}}{=} i^{:\text{Int}_{[0,n_1+n_2-1]}} \to \begin{cases} 0 \le i < n_1 : & a_1(i) \\ n_1 \le i < n_1 + n_2 : & a_2(i - n_1) \end{cases}$$

satisfies the associativity law:

$$(a_1 +\!\!+ a_2) +\!\!+ a_3 = a_1 +\!\!+ (a_2 +\!\!+ a_3) \quad .$$

Proof Both sides of the law evaluate to the same partial function of type Int $\to A$:

$$(a_1 +\!\!+ a_2) +\!\!+ a_3 = a_1 +\!\!+ (a_2 +\!\!+ a_3) = i^{:\mathrm{Int}_{[0,n_1+n_2+n_3-1]}} \to \begin{cases} 0 \le i < n_1 : & a_1(i) \\ n_1 \le i < n_1 + n_2 : & a_2(i - n_1) \\ n_1 + n_2 \le i < n_1 + n_2 + n_3 : & a_3(i - n_1 - n_2) \end{cases} .$$

Now we establish a rigorous equivalence between the `List` and `Array` types.

Statement 8.5.2.2 The type $\mathrm{List}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times \mathrm{List}^A$ is equivalent to $\mathrm{Array}^A_n \stackrel{\text{def}}{=} \mathrm{Int}_{[0,n-1]} \to A$, where $n \ge 0$ and $\mathrm{Int}_{[0,n-1]}$ is the (possibly empty) subset of integers i within the range $0 \le i \le n-1$.

Proof We need to implement two isomorphism maps f_1, f_2 and show that:

$$f_1 : \mathrm{Array}^A_n \to \mathrm{List}^A \quad , \qquad f_2 : \mathrm{List}^A \to \mathrm{Array}^A_n \quad , \qquad f_1 \,;\, f_2 = \mathrm{id} \quad , \qquad f_2 \,;\, f_1 = \mathrm{id} \quad .$$

To implement f_1, we proceed by induction in n. The base case is $n = 0$, and we map Array_0 into an empty list. The inductive step assumes that f_1 is already defined on arrays of length n, and we now need to define f_1 for arrays of length $n+1$. An array of length $n+1$ is a partial function $g^{:\mathrm{Int}_{[0,n]} \to A}$ defined on the integer interval $[0, n]$. We now split that array into its first element, $g(0)$, and the rest of the array, which needs to be represented by another partial function, say $g' \stackrel{\text{def}}{=} i \to g(i+1)$, defined on the integer interval $[0, n-1]$. The function g represents an array of length n. By the inductive assumption, f_1 is already defined for arrays of length n. So, we can compute $f_1(g') : \mathrm{List}^A$ and thus create a value of type $\mathbb{0} + A \times \mathrm{List}^A$, which is equivalent to a value of type List^A.

```
def f1[A](arr: Array[A]): List[A] =
  if (arr.length == 0) List()
  else arr(0) :: f1(arr.tail)
```

$$f_1(\mathrm{Array}^A_0) \stackrel{\text{def}}{=} \mathbb{1} + \mathbb{0}^{:A \times \mathrm{List}^A} \quad ,$$
$$f_1(g^{:\mathrm{Int}_{[0,n]} \to A}) \stackrel{\text{def}}{=} \mathbb{0} + g(0)^{:A} \times \overline{f_1}(i \to g(i+1)) \quad .$$

To implement f_2, we use induction in the structure of the list. The length n of the array is not known in advance and needs to be computed as we perform pattern matching on the given `List[A]` value. The base case is an empty list, which yields an empty array (i.e., an array of length $n = 0$). In the inductive step, we assume that we already defined f_2 on lists of length n, and we now need to define f_2 for lists of length $n+1$. Such a list must have the form $\mathbb{0} + x^{:A} \times s^{:\mathrm{List}^A}$, where s is a list of length n. By the inductive assumption, we are allowed to apply f_2 to s and obtain an array of length n, i.e., a partial function $g^{:\mathrm{Int}_{[0,n-1]} \to A}$. So we define $f_2(\mathbb{0} + x \times s)$ as a new array whose 0th element is x and the ith element is computed by applying the function $g \stackrel{\text{def}}{=} f_2(s)$ to $i - 1$:

```
def f2[A: ClassTag]: List[A] => Array[A] =
{  case List()    => Array()
   case x :: s    => Array(x) ++ f2.apply(s)
}
```

$$f_2(\mathbb{1} + \mathbb{0}^{:A \times \mathrm{List}^A}) \stackrel{\text{def}}{=} \mathrm{Array}^A_0 \quad ,$$
$$f_2(\mathbb{0} + x^{:A} \times s^{:\mathrm{List}^A}) \stackrel{\text{def}}{=} i^{:\mathrm{Int}_{[0,n]}} \to \begin{cases} i = 0 : & x \quad , \\ i \ge 1 : & \overline{f_2}(s)(i-1) \quad . \end{cases}$$

To show that f_1 and f_2 are inverse functions for each other, we again need to use induction. The base case is the empty list and the empty array, which are indeed mapped isomorphically to each other. The inductive step for $f_1 \,;\, f_2$ is an array of length $n+1$ with the inductive assumption that $\overline{f_1 \,;\, f_2} = \mathrm{id}$ for arrays of length n. Writing out the code of $f_1 \,;\, f_2$, we find that $g \triangleright f_1 \,;\, f_2 = g$:

$$g^{:\mathrm{Int}_{[0,n]} \to A} \triangleright f_1 \,;\, f_2 = g \triangleright f_1 \triangleright f_2 = (\mathbb{0} + g(0) \times \overline{f_1}(i \to g(i+1))) \triangleright f_2$$
$$= i \to \begin{cases} i = 0 : & g(0) \\ i \ge 1 : & \underline{\overline{f_2}(\overline{f_1}}(i \to g(i+1)))(i-1) \end{cases} = i \to \begin{cases} i = 0 : & g(0) \\ i \ge 1 : & \underline{\mathrm{id}}(i \to g(i+1)))(i-1) \end{cases}$$
$$= i \to \begin{cases} i = 0 : & g(0) \\ i \ge 1 : & g(\underline{(i-1)+1}) \end{cases} = (i \to g(i)) = g .$$

Similarly, we find that $(\mathbb{0}+x^{:A}\times s^{:\text{List}^A})\triangleright f_2\,{}^{\circ}_{\circ}\,f_1=(\mathbb{0}+x^{:A}\times s^{:\text{List}^A})$ via this calculation:

$$(\mathbb{0}+x^{:A}\times s^{:\text{List}^A})\triangleright f_2\,{}^{\circ}_{\circ}\,f_1=f_1\left(i^{:\text{Int}_{[0,n]}}\to\begin{cases}i=0: & x\\ i\geq 1: & \overline{f_2}(s)(i-1)\end{cases}\right)$$
$$=\mathbb{0}+x\times\overline{f_1}(\underline{i\to\overline{f_2}(s)(i+1-1)})=\mathbb{0}+x\times\underline{\overline{f_1}(\overline{f_2}}(s))=\mathbb{0}+x\times s\quad.$$

This concludes the proof of the isomorphism between `Array` and `List`.

Since arrays and lists are isomorphic as types, the concatenation for lists is associative as long as we show that the concatenation operation for lists is isomorphic to that we defined for arrays.

Statement 8.5.2.3 The `concat` function for lists is defined recursively as:

```
def concat[A](p: List[A], q: List[A])
    : List[A] = p match {
  case List() => q
  case a :: t => a :: concat(t, q)
}
```

$$p^{:\text{List}^A}{+\!\!+}q^{:\text{List}^A}\stackrel{\text{def}}{=}p\triangleright\begin{array}{c||c}\hline & \text{List}^A\\ \hline \mathbb{1} & q\\ A\times\text{List}^A & a\times t\to\mathbb{0}+a\times(t\overline{{+\!\!+}}q)\\ \hline\end{array}\quad,$$

and is equivalent to the `concat` function on arrays defined in Statement 8.5.2.1:

$$\forall a^{:\text{Array}^A_{n_1}},b^{:\text{Array}^A_{n_2}}.\quad f_1(a){+\!\!+}f_1(b)=f_1(a{+\!\!+}b)\quad,$$
$$\forall p^{:\text{List}^A},q^{:\text{List}^A}.\quad f_2(p){+\!\!+}f_2(q)=f_2(p{+\!\!+}q)\quad,$$

where f_1, f_2 are the isomorphism maps defined in Statement 8.5.2.2.

Proof If we show the property for f_2, we can apply f_1 to both sides and obtain the other property.

The base case for lists is $[]{+\!\!+}q=q$. This is clearly isomorphic to concatenating an empty array with another array since $f_2([])$ is an empty array.

The inductive step is $p{+\!\!+}q$ where p is a non-empty list, $p=\mathbb{0}+a^{:A}\times t^{:\text{List}^A}$. We need to show that:

$$f_2(\mathbb{0}+a\times t){+\!\!+}f_2(q)=f_2((\mathbb{0}+a\times t){+\!\!+}q)=f_2(\mathbb{0}+a\times(t{+\!\!+}q))\quad.$$

By definition of f_2, we have:

$$f_2(\mathbb{0}+a\times t)=i\to\begin{cases}i=0: & a\\ i\geq 1: & \overline{f_2}(t)(i-1)\end{cases}\quad,$$
$$f_2(\mathbb{0}+a\times(t{+\!\!+}q))=i\to\begin{cases}i=0: & a\\ i\geq 1: & \overline{f_2}(t{+\!\!+}q)(i-1)\end{cases}\quad.$$

The inductive assumption guarantees that $\overline{f_2}(t{+\!\!+}q)=f_2(t){+\!\!+}f_2(q)$. Using the definition of array concatenation and assuming that the length of t is n_1 and the length q is n_2, we get:

$$f_2(\mathbb{0}+a\times(t{+\!\!+}q))=i\to\begin{cases}i=0: & a\\ 1\leq i<n_1+1: & f_2(t)(i-1)\\ n_1+1\leq i<n_1+1+n_2: & f_2(q)(i-n_1-1)\end{cases}\quad.$$

This is the same array as the concatenation $f_2(\mathbb{0}+a\times t){+\!\!+}f_2(q)$. This concludes the proof.

8.5.3 Higher-order type functions. Kinds

Type constructors are types parameterized by other types. We have also seen types parameterized by type constructors, e.g.:

```
trait Functor[F[_]] { ... }
```

It is important to distinguish the ways in which types can be parameterized. A type constructor such as `List` can be seen as a type-to-type function (TTF): given a type, e.g., `Int`, it produces another type, `List[Int]`. It is a type error to apply `List` to a type parameter that is not a simple type:

```
scala> val x: List[List] = ???
<console>:11: error: type List takes type parameters
       val x: List[List] = ???
                  ^
```

To describe the restriction on possible type parameters of `List`, we say that `List` has **kind signature** $* \to *$. The symbol $*$ means an ordinary type (not a type function), i.e., a type that can have values. In the type notation, we can write "kind signatures" as:

$$\text{Int} : * \quad , \qquad \text{List} : * \to * \quad , \qquad \text{List}^{\text{Int}} : * \quad .$$

So, the concept of "**kind**" can be understood as the "type signature of a type". Types of kind $*$ (such as `Int` or `String`) can have values, but types of kind $* \to *$ cannot:

```
scala> val x: Seq = ???
<console>:11: error: type Seq takes type parameters
       val x: Seq = ???
              ^
```

Although Scala will check that all kinds match, there is no syntax in Scala to declare kinds.

The `Functor` typeclass is another example of a non-trivial "kind": a type whose type parameter is itself a type constructor.

$$\text{Functor} : (* \to *) \to * \quad .$$

The `Functor` type can be seen as a higher-order type-to-type function (TTF) since it takes a type parameter that is itself a TTF. Such types are often called "higher-kinded types".

For higher-order TTFs, Scala requires syntax such as `Functor[F[_]]` or `F[_[_]]`, for example:

```
// Ap1 and Ap2 will simply substitute type arguments into type constructors of various kinds.
type Ap1[F[_], A] = F[A]    // Ap1 : ((* → *) × *) → *
type Ap2[P[_[_], _], Q[_], R] = P[Q, R] // Ap2: (((* → *) × * → *) × (* → *) × *) → *
type G[A] = Either[(A, A, String), A]   // G: * → *
type X = Ap2[Ap, G, Int]   // OK; X is now Either[(Int, Int, String), Int]
type Y = Ap2[Ap, Ap, Int]  // Type error: the second argument of Ap2 has wrong kind.
type Z = Ap2[G, G, Int]    // Type error: the first argument of Ap2 has wrong kind.
```

The "kind projector" plugin is often needed when writing code with higher-order TTFs:

```
// ‘Twice‘ will apply the type constructor Q twice to its type argument, and substitute into P.
type Twice[P[_[_], _], Q[_], R] = P[Lambda[X => Q[Q[X]]], R] //
     Twice: (((* → *) × * → *) × (* → *) × *) → *
type O2[A] = Option[(A, A)]                                  // O2: * → *
type X2 = Twice[Ap1, O2, Int]          // X2 is now Option[(Option[(Int, Int)], Option[(Int, Int)])].
val x2: X2 = Some((Some((1, 2)), Some((3, 4))))   // Types match for ‘x2‘.
```

8.5.4 Inductive typeclasses and their properties

We have seen many examples of typeclasses that have similar structural properties. For instance, a product of semigroups is a semigroup, a product of monoids is a monoid, a product of functors is a functor, a product of pointed functors is a pointed functor, and so on. It turns out that all these typeclasses have a common structure that simplifies reasoning. The common structure is that all typeclass methods can be expressed as a *single* uncurried function of type $P^A \to A$, as Table 8.8 shows.

Functions of type $P^A \to A$ compute new values of type A from previous values (of type A or other types) wrapped by the functor P. This superficially resembles defining values of type A by induction and so motivates the following definition: A typeclass is **inductive** if a type A's typeclass instance is a value of type $P^A \to A$ with some functor P called the **structure functor** of the typeclass.

Typeclass	Instance type as a function	Inductive form	Structure functor
default value	$\mathbb{1} \to A$	$P^A \to A$	$P^A \stackrel{\text{def}}{=} \mathbb{1}$
semigroup	$A \times A \to A$	$P^A \to A$	$P^A \stackrel{\text{def}}{=} A \times A$
monoid	$\mathbb{1} + A \times A \to A$	$P^A \to A$	$P^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$
functor	$F^A \times (A \to B) \to F^B$	$S^{\bullet,F} \to F^\bullet$	$S^{B,F^\bullet} \stackrel{\text{def}}{=} \forall A.\, F^A \times (A \to B)$
pointed functor	$B + F^A \times (A \to B) \to F^B$	$S^{\bullet,F} \to F^\bullet$	$S^{B,F^\bullet} \stackrel{\text{def}}{=} \forall A.\, B + F^A \times (A \to B)$
contrafunctor	$F^A \times (B \to A) \to F^B$	$S^{\bullet,F} \to F^\bullet$	$S^{B,F^\bullet} \stackrel{\text{def}}{=} \forall A.\, F^A \times (B \to A)$
pointed contrafunctor	$\mathbb{1} + F^A \times (B \to A) \to F^B$	$S^{\bullet,F} \to F^\bullet$	$S^{B,F^\bullet} \stackrel{\text{def}}{=} \forall A.\, \mathbb{1} + F^A \times (B \to A)$

Table 8.8: Structure of typeclass instance values for various inductive typeclasses.

A value of type $P^A \to A$ contains all the methods of the inductive typeclass in a single function. We can implement this definition by the Scala code:

```
final case class InductiveTypeclass[P[_]: Functor, A](methods: P[A] => A)
```

For example, the `Monoid` typeclass is inductive because its instances have type:

$$\text{Monoid}^A = (A \times A \to A) \times A \cong (\mathbb{1} + A \times A \to A) = P^A \to A \quad \text{where } P^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A \quad .$$

So, the `Monoid` typeclass can be declared as an inductive typeclass by code like this:

```
final case class MonoidStructure[A](s: Option[(A, A)])
val monoidStructureFunctor: Functor[MonoidStructure] = ... /* implementation */
type InductiveMonoid[A] = InductiveTypeclass[MonoidStructure, A]
```

Implementing the `Monoid` typeclass via a structure functor P and a function $P^A \to A$ is inconvenient for practical programming. The inductive form $P^A \to A$ is mainly useful for reasoning about general properties of typeclasses. To illustrate that kind of reasoning, we will show that the product-type, the function-type, and the recursive-type constructions work for all inductive typeclasses.

Consider an arbitrary inductive typeclass `TC` defined via a structure functor `P`:

```
type P[A] = ...
type TC[A] = P[A] => A
```

If two types A, B have an instance of the typeclass `TC`, an instance for the product $A \times B$ can be derived automatically using the function `productTC`:

```
def productTC[A, B](f: P[A] => A, g: P[B] => B): P[(A, B)] => (A, B) =
  p => ( f(p.map(_._1)), g(p.map(_._2)) )
```

$$\text{productTC} : (P^A \to A) \times (P^B \to B) \to P^{A\times B} \to A \times B \quad ,$$
$$\text{productTC} \stackrel{\text{def}}{=} (f^{:P^A\to A} \times g^{:P^B\to B}) \to p^{:P^{A\times B}} \to (p \triangleright \pi_1^{\uparrow P} \triangleright f) \times (p \triangleright \pi_2^{\uparrow P} \triangleright g) \quad .$$

This explains why product types have instances for all typeclasses listed in Table 8.8: those typeclasses are inductive. (However, this argument does not prove that the *laws* of a given typeclass will also hold for the product types.)

It is interesting to note that the co-product construction cannot be derived in general for arbitrary inductive typeclasses: given $P^A \to A$ and $P^B \to B$, it is not guaranteed that we can compute $P^{A+B} \to A + B$. Not all inductive typeclasses support the co-product construction (although many do).

The function-type construction promises a typeclass instance for $E \to A$ if the type A has a typeclass instance. This construction works for any inductive typeclass because a value of type

$P^{E\to A} \to E \to A$ can be computed from a value of type $P^A \to A$ when P is a functor:

$$q : (P^A \to A) \to P^{E\to A} \to E \to A \quad , \qquad q \stackrel{\text{def}}{=} h^{:P^A\to A} \to p^{:P^{E\to A}} \to e^{:E} \to p \triangleright (x^{:E\to A} \to e \triangleright x)^{\uparrow P} \triangleright h \quad .$$

As in the case of the product construction, the laws still need to be checked for the new instances.

Finally, let us show that the recursive type construction works for inductive typeclasses. Consider a recursive type T defined by a type equation $T \stackrel{\text{def}}{=} S^T$, where the functor $S^\bullet$ "preserves" typeclass instances: if A has an instance then S^A also does, as we saw in all our examples in this chapter. In other words, we have a function $\text{tcS} : (P^A \to A) \to P^{S^A} \to S^A$ that creates typeclass instances of type $P^{S^A} \to S^A$ out of instances of type $P^A \to A$. Then we define an instance `tcT` for T as:

$$\text{tcT}^{:P^T\to T} \stackrel{\text{def}}{=} p^{:P^T} \to \text{tcS}\,(\overline{\text{tcT}})(p) \quad .$$

This recursive definition terminates because it is implemented as an expanded function. The types match since we can convert between the equivalent types T and S^T whenever necessary, so $p^{:P^T}$ can be converted to a value of type P^{S^T}, while the value $\text{tcS}\,(\text{tcT})(p)$ can be converted from type S^T back to type T. In Scala code, the conversions between T and S^T are implemented by the constructor `T(...)` and accessor `(_.s)` methods of the case class that wraps the type T.

```
type S[A] = ...                // Define a functor S as required.
final case class T(s: S[T])    // Define the recursive type T as T≜S^T.
def tcS: TC[A] => TC[S[A]] = ... // Compute instances for S[A] from instances of A.
def tcT: P[T] => T = p => T(tcS(tcT)(p.map(_.s))) // Define the recursive instance.
```

In this way, we can implement the recursive-type construction with any inductive typeclass. (The typeclass laws still need to be checked for the recursively defined instances.)

The `Functor` and `Pointed` typeclasses are also inductive. Instead of a structure functor, we need to use a higher-order type function denoted by $S^{A,F^\bullet}$ and parameterized by a type constructor $F^\bullet$ as well as by a type parameter A. (The type S has kind $* \times (* \to *) \to *$.) The methods of these typeclasses are expressed as $S^{\bullet,F} \to F^\bullet$, which is analogous to $P^A \to A$ except for additional type parameters. Similar arguments can be made for these typeclasses, although it is more difficult to reason about type constructors (and the laws will not hold without additional assumptions). As we have seen, the product-type, the function-type, and the recursive-type constructions work for functors, contrafunctors, pointed functors, and pointed contrafunctors.

Important inductive typeclasses are functors and contrafunctors (Chapter 6), filterables (Chapter 9), monads (Chapter 10), and applicatives (Chapter 11). We will see in Chapter 13 that another general construction also works for all inductive typeclasses, — the "free" type construction.

We have also seen examples of typeclasses that are *not* inductive. The `Extractor` typeclass has instances of type $A \to Z$, which is not of the form $P^A \to A$ for any functor P. Another such typeclass is `Copointed`, whose method $F^A \to A$ is not of the form $S^{\bullet,F} \to F^\bullet$. However, the methods of these typeclasses can be written in the inverted form $A \to P^A$ with some functor P. We call them **co-inductive typeclasses**. A motivation for this name is that the *co-product* construction (rather than the product construction) works with co-inductive typeclasses: given values of types $A \to P^A$ and $B \to P^B$, we can produce a value of type $A + B \to P^{A+B}$, i.e., a co-product instance, but not necessarily a value of type $A \times B \to P^{A\times B}$, which would be a product-type instance. The function-type construction is not guaranteed to work with co-inductive typeclasses, but the recursive-type construction and the "free" type construction can be implemented after appropriate modifications.

The `Eq` typeclass is "uninductive" (neither inductive nor co-inductive) because its instance type, $A \times A \to \mathbb{2}$, is neither of the form $P^A \to A$ nor $A \to P^A$. The traversable functor (Chapter 12) is another example of an uninductive typeclass. Uninductive typeclasses usually support fewer type constructions. For example, only *polynomial* types can have instances of `Eq` or `Traversable` typeclasses.

8.5.5 Typeclasses with several type parameters (type relations)

A typeclass constraint in a function, such as `func[A: Monoid]`, restricts a type parameter to a certain type domain. Sometimes it is necessary to restrict *several* type parameters to satisfy some conditions together. A condition that constrains several type parameters at once is called a **type relation**. Let us look at two simple examples of using type relations in practice.

The first example is converting integer numbers to floating point. The ranges of the available types allow us to convert a `Short` to a `Float` and an `Int` to a `Double`. Can we implement the type signature

```
def convertNumber[M, N](x: M): N
```

where the type parameters `M`, `N` are constrained to be either `M = Short` and `N = Float`, or `M = Int` and `N = Double`? (Also, later we may need to add support for more types.)

The second example is converting mutable data structures into the corresponding immutable ones. The Scala library contains data structures such as sequences, sets, and dictionaries; each data type has a mutable and an immutable version. Can we implement a function with type signature

```
def convertData[Mut[_], Immut[_], A](data: Mut[A]): Immut[A]
```

where the type parameters `Mut` and `Immut` are constrained to represent the mutable / immutable versions of a supported data structure (for example, `Mut = mutable.Set` and `Immut = immutable.Set`, etc.)?

A typeclass constraint for a single type parameter, such as `func[A: Monoid]`, is implemented by requiring an "evidence value" of type `Monoid[A]` as an additional argument of the function `func`. Implementing a type relation for *two* type parameters `A`, `B` is similar: We define a type constructor, say `Rel[_, _]`, and create some evidence values of type `Rel[A, B]` with chosen type parameters `A`, `B`. Any function, say `f[A, B]`, that needs a relation constraint on its type parameters will take an extra evidence argument of type `Rel[A, B]`. This will prevent our code from using `f[A, B]` with types `A`, `B` that are not in the required relation.

Using this technique, we can define a relation `MayConvert` and write the code for `convertNumber`:

```
final case class MayConvert[A, B](convert: A => B)  // Evidence value contains a conversion function.
implicit val ev1 = MayConvert[Short, Float](_.toFloat)          // Evidence value for [Short, Float].
implicit val ev2 = MayConvert[Int, Double](_.toDouble)          // Evidence value for [Int, Double].
def convertNumber[M, N](x: M)(implicit ev: MayConvert[M, N]): N = ev.convert(x)
```

With these definitions, it will be a compile-time error to use `convertNumber` with unsupported types:

```
scala> convertNumber(123)
res0: Double = 123.0

scala> convertNumber(123:Short)
res1: Float = 123.0

scala> convertNumber("abc")
<console>:17: error: could not find implicit value for parameter ev: MayConvert[String,N]
       convertNumber("abc")
                    ^
```

As we have just seen, a type relation is defined by creating some evidence values. The code above defines `MayConvert` as a one-to-one type relation because the evidence values (or "relation instances") `ev1` and `ev2` do not have any types in common. So, `MayConvert` is equivalent to a type-to-type *function* that maps `Short` to `Float` and `Int` to `Double`. However, type relations are not limited to one-to-one relations. By creating suitable implicit evidence values, we can implement one-to-many or many-to-many relations when needed.

A practical example of a many-to-many type relation is the compatibility between physical units. Miles can be converted into kilometers; pounds can be converted into ounces or kilograms; but kilograms cannot be converted into miles. Type relations allow us to implement type-safe operations for quantities with units. Adding kilograms to pounds will automatically convert the quantity to a common unit, while adding kilograms to miles will raise a compile-time error.

Begin by declaring a type for "quantity with units" and some type names for the supported units:

```
trait KG; trait LB; trait OZ; trait KM; trait MI; trait FT // Declare names for some supported units.
final case class Quantity[U](value: Double)    // Constant functor: No values of type ‘U‘ are stored.
def add[U1, U2](x: Quantity[U1], y: Quantity[U2]): Quantity[U2] = ???
```

The parameter `U` in `Quantity[U]` is called a **phantom** type parameter because `Quantity[U]` contains no values that use `U`. The parameter `U` would not be phantom if `Quantity[U]` were not a constant functor but instead contained values e.g., of type `U` or of type `U => Double` or of some other type that uses `U`.

The function `add[U1, U2](x, y)` must impose a type relation constraint on `U1` and `U2`, so that `x` and `y` may be added only when they have compatible units. The type relation is implemented as a type constructor `Convertible` with two type parameters. A "relation instance" (i.e., a value of type `Convertible[U1, U2]`) will contain a multiplier for converting any quantity from units `U1` to units `U2`:

```
final case class Convertible[U1, U2](multiplier: Double)
implicit val c1 = Convertible[KG, KG](1.0)
implicit val c2 = Convertible[LB, KG](0.453592) // Pounds in kilograms.
implicit val c3 = Convertible[KM, KM](1.0)
implicit val c4 = Convertible[MI, KM](1.60934)  // Miles in kilometers. Add more definitions here.
```

Now we can implement the `add` function and verify that the type relation works as we intended:

```
def add[U1, U2](x: Quantity[U1], y: Quantity[U2])(implicit ev: Convertible[U1, U2]): Quantity[U2] =
  Quantity(x.value * ev.multiplier + y.value)

scala> add(Quantity[LB](1), Quantity[KG](1))   // 1 pound + 1 kg = 1.453592 kg.
res0: Quantity[KG] = Quantity(1.453592)

scala> add(Quantity[MI](1), Quantity[KG](1))   // Compile-time error: cannot add miles to kilograms.
<console>:25: error: could not find implicit value for parameter ev: Convertible[MI,KG]
```

To make this code more convenient for practical use, we can add extension methods that shorten the syntax from `Quantity[KG](1)` to `1.kg` and from `add(2.lb, 2.kg)` to a more readable `2.lb + 2.kg`:

```
implicit class QuantitySyntax(x: Double) {
  def kg = Quantity[KG](x)
  def lb = Quantity[LB](x)        // Add more definitions as needed.
}
implicit class QuantityAdd[U1](x: Quantity[U1]) {
  def +[U2](y: Quantity[U2])(implicit ev: Convertible[U1, U2]): Quantity[U2] = add(x, y)
}

scala> 2.lb + 2.kg        // Compute 2 pounds + 2 kilograms; the result is in kilograms.
res1: Quantity[KG] = Quantity(2.907184)
```

Another necessary improvement is reducing the number of implicit values. The current code uses n^2 implicit values for every group of n compatible units. Adding support for a new unit (say, inches) requires adding implicit values for converting between inches and all previously defined units of length. To avoid this problem, the code must be reorganized to convert all compatible quantities to chosen units (e.g., all length to kilometers and all mass to kilograms). This makes the `Convertible` relation many-to-one instead of many-to-many. The resulting code is shown in Figure 8.2.

8.5.6 Inheritance and automatic conversions of typeclasses

It often happens that one typeclass is a subset of another; for example, a `Monoid` instance for a type T means that T already has the properties of both `Semigroup` and `HasDefault` typeclasses. One says that the `Monoid` typeclass **inherits** from the `Semigroup` and `HasDefault` typeclasses. We may want to express inheritance relations between typeclasses, so that, e.g., a `Monoid` instance should automatically imply the presence of a `Semigroup` instance, without extra code.

One way of inheriting a typeclass is to add a constraint to the new typeclass constructor:

```
final case class Semigroup[T](combine: (T, T) => T)
final case class Monoid[T: Semigroup](empty: T)  // Inherit ‘Semigroup‘.
```

```
trait KG; trait LB; trait OZ; trait KM; trait MI; trait FT            // Some supported units.

// This type relation is many-to-1: it relates all mass units to KG and all length units to KM.
final case class Convertible[U1, U2](multiplier: Double) extends AnyVal
// Units of mass.
implicit val cKG = Convertible[KG, KG](1.0)
implicit val cLB = Convertible[LB, KG](0.453592)
implicit val cOZ = Convertible[OZ, KG](0.0283495)
// Units of distance.
implicit val cKM = Convertible[KM, KM](1.0)
implicit val cMI = Convertible[MI, KM](1.60934)
implicit val cFT = Convertible[FT, KM](0.0003048)
// We can add support for new units whenever necessary.

final case class Quantity[U](value: Double) extends AnyVal { // Use `AnyVal` to reduce run-time cost.
  def +[U2, C](q: Quantity[U2])(implicit ev1: Convertible[U, C], ev2: Convertible[U2, C]) =
      Quantity[U2](value * ev1.multiplier / ev2.multiplier + q.value)

  def ==[U2, C](q: Quantity[U2])(implicit ev1: Convertible[U, C], ev2: Convertible[U2, C]) =
      value * ev1.multiplier == q.value * ev2.multiplier
}

implicit class QuantitySyntax(x: Double) {    // Extension methods defined for convenience.
  def kg = Quantity[KG](x)
  def lb = Quantity[LB](x)
  def oz = Quantity[OZ](x)
  def km = Quantity[KM](x)
  def mi = Quantity[MI](x)
  def ft = Quantity[FT](x)
  // This general extension method, e.g., `1.in[KM]`, will also work for any units defined later.
  def in[U](implicit ev: Convertible[U, _]): Quantity[U] = Quantity[U](x)
}

scala> 1.in[KM]         // Use the general method `.in` with a type parameter.
res1: Quantity[KM] = Quantity(1.0)

scala> 10000.ft + 1.km  // Use the `.ft` and `.km` extension methods.
res2: Quantity[KM] = Quantity(4.048)

scala> 1.kg + 2.lb == 32.oz + 1.kg   // Compare values safely.
res2: Boolean = true

scala> 1.km + 2.lb      // Compile-time error: cannot add kilometers to pounds.
<console>:29: error: could not find implicit value for parameter ev2: Convertible[LB,C]
       1.km + 2.lb
            ^

scala> trait YD; implicit val cYD = Convertible[YD, KM](0.0009144)  // Add support for yards.
defined trait YD
cIN: Convertible[YD,KM] = Convertible(9.144E-4)

scala> 1.in[YD] + 1.ft // Use .in[YD] to compute 1 YD + 1 FT = 4 FT.
res3: Quantity[FT] = Quantity(4.0)
```

Figure 8.2: Implementing type-safe computations with units of length and mass.

The case class `Monoid` combines a previous `Semigroup` instance with the new method `empty`. Creating an instance of `Monoid` for a type `T` will then require having an instance of `Semigroup` for `T`:

```
implicit val c1 = Semigroup[Int](_ + _)
implicit val c2 = Monoid[Int](0)  // Works only if a ‘Semigroup[Int]‘ is available.
```

Given a `Monoid[T]` instance, how can we recover the inherited implicit value of type `Semigroup[T]`? This is achieved by redefining the `Monoid` class like this:

```
final case class Monoid[T](empty: T)(implicit val semigroup: Semigroup[T])   // Use ‘implicit val‘.
implicit def semigroupFromMonoid[T](implicit ti: Monoid[T]): Semigroup[T] = ti.semigroup
```

Another possibility of implementing typeclass inheritance is to use object-oriented inheritance of traits. This approach is often used when defining typeclasses as traits with methods:

```
trait Semigroup[T] { def combine: (T, T) => T }
trait Monoid[T] extends Semigroup[T] { def empty: T } // ‘def combine‘ is inherited from ‘Semigroup‘.
```

Creating an instance of `Monoid` for a type `T` no longer requires having an instance of `Semigroup` for `T`:

```
implicit val c: Monoid[Int] = new Monoid[Int] {
  def empty: Int =  0
  def combine: (Int, Int) => Int =  _ + _
}
```

With this approach, we cannot avoid repeating the code for `def combine...` even if we *do* already have a `Semigroup` instance. However, conversion from `Monoid[T]` to `Semigroup[T]` is now automatic due to **object-oriented inheritance**: Scala considers `Monoid[T]` a subtype of `Semigroup[T]` because the `Monoid` class is declared as `extends Semigroup`.

A problem with the object-oriented inheritance is that automatic type conversions to parent classes *cannot be disabled*. When several typeclasses inherit from the same parent typeclass, duplicate implicit instances of the parent typeclass will be present, which is a compile-time error:

```
trait TC[A]                     // Parent typeclass.
trait TC1[A] extends TC[A]      // Two different typeclasses TC1 and TC2 inherit from TC.
trait TC2[A] extends TC[A]
// The function f requires A to have both TC1 and TC2 instances and then wants to access TC instance.
def f[A: TC1 : TC2]() = {
  implicitly[TC[A]]  // Compilation fails because two implicit values of type TC[A] are found.
}
```

When typeclass inheritance is implemented by combining instances without object-oriented inheritance, conversions to parent typeclasses are not automatic: they are implemented by implicit functions that need to be imported into the current scope. The programmer's code can avoid producing duplicate instances by choosing which implicit conversions to import:

```
final case class TC[A]()
final case class TC1[A]()(implicit val tc: TC[A])                      // TC1 inherits TC.
object TC1 { implicit def toTC[A](implicit x: TC1[A]): TC[A] = x.tc }
final case class TC2[A]()(implicit val tc: TC[A])                      // TC2 inherits TC.
object TC2 { implicit def toTC[A](implicit x: TC2[A]): TC[A] = x.tc }
// The function f requires A to have both TC1 and TC2 instances and then wants to access TC instance.
def f[A: TC1 : TC2]() = {
      import TC1._  // Can import TC1._ or TC2._ but not both. If the next line is uncommented,
//    import TC2._  // compilation will fail because two implicits of type TC[A] will be found!
  implicitly[TC[A]] // This compiles successfully. One implicit instance of TC[A] can be found.
}
```

The problem of duplicate inherited instances can be solved with less work for the programmer if the typeclasses are implemented using a special, more complicated encoding.[8]

[8] See slides 5–14 of the talk by John de Goes: https://www.slideshare.net/jdegoes/scalaz-8-a-whole-new-game

9 Computations in functor blocks. I. Filterable functors and contrafunctors

Chapter 6 studied the mathematical properties of `map` and derived the concept of "functor" as a general type constructor that permits a `map` method with useful properties. In this chapter, we will study the `filter` method in a similar way. We will start from practical examples and then derive the relevant mathematical properties. This will give us a precise definition of "filterable" type constructors — those that permit useful implementations of `filter`.

9.1 Practical uses of filtering

The Scala standard library defines the `filter` method on sequences, sets, and other data structures. An example of using `filter` is the following calculation,

$$\sum_{x\in\mathbb{Z};\,0\leq x\leq 100;\,\cos x>0}\sqrt{\cos(x)}\approx 38.71\quad.$$

```
scala> (0 to 100).map(x => math.cos(x)).filter(_ > 0).map(math.sqrt).sum
res0: Double = 38.71218949848382
```

The role of `filter` in this computation is to select only the positive values of $\cos(x)$. It is safe to apply the square root function to positive values, so the code will work correctly.

The code above is a chain of methods, but the same code can be written using Scala's `for`/`yield` syntax, which is called a **functor block** in this book. The `filter` operation is represented by an `if` keyword embedded in a functor block. Compare each line of the functor block code with the corresponding line of the code written via method chains:

```
(for { x <- 0 to 100
    y = math.cos(x)
    if y > 0
  } yield math.sqrt(y)
).sum
```

```
(0 to 100).map { x =>
   math.cos(x) }.filter { y =>
   y > 0 }.map { y =>
     math.sqrt(y)
  }.sum
```

Functor blocks require the first line to have a left arrow placed before a "source", i.e., a value of a functor type (e.g., a sequence or a `Try`). Each line of a functor block can be viewed as a computation that creates an intermediate value of the same "source" functor type. The result value of the entire functor block is again a value of the same functor type. So, a functor block manipulates data wrapped by a functor without changing the type of the wrapper.

The name "functor block" is chosen because the code is a single expression (a "block") whose type is required to be a functor. The functor's type is selected by the source's type in the first line.

We have seen in Chapter 6 that the functor block syntax becomes available for a type constructor that has a `map` method (i.e., for a functor). To support the embedded `if` keyword, the type constructor must support a method called `withFilter` with the same type signature as the `filter` method. The type signatures of `map` and `withFilter` methods for a type constructor `F[_]` can be written as

```
class F[A] {  // F[A] is some type constructor that has .map and .withFilter methods.
  def map[B](f: A => B): F[B] = ...
  def withFilter(p: A => Boolean): F[A] = ...
}
```

A type constructor that supports the filtering operation is called **filterable**.

The main focus of this chapter is to explore filterable functors in detail. Programmers would intuitively expect the filtering operation to have certain properties. We will now examine the expected properties and translate them into mathematical laws for the function `withFilter`.

9.1.1 Examples and intuitions for the filtering operation

Examples of often used filterable functors defined in the Scala library are `List` and `Option`:

```
scala> List(64, 128).filter(_ > 100).map(_ * 2)
res0: List[Int] = List(256)

scala> Some(128).filter(_ > 100).map(_ * 2)
res1: Option[Int] = Some(256)

scala> Some(64).filter(_ > 100).map(_ * 2)
res2: Option[Int] = None
```

In an intuitive view, a functor wraps one or more data values, and the filtering operation may decrease the number of values wrapped. e.g., the length of the sequence `List(64, 128)` is decreased from 2 to 1 after filtering with the condition `_ > 100`. The `Option` functor can wrap at most one value, so filtering with a predicate returning `false` will return an empty `Option` value (i.e., `None`). In all cases, the resulting collection will not contain values that fail the filtering predicate.

Note that `Option[T]` is written in the type notation as a disjunctive type $\mathbb{1}+T$, while `List[T]` can be viewed as an "infinite disjunction",

$$\text{List}^T = 1 + T + T\times T + T\times T\times T + \ldots \tag{9.1}$$

Disjunctive type constructors are able to hold a different number of values of type T in different parts of the disjunction (e.g., $T\times T$ for 2 values and $T\times T\times T$ for 3 values of T).

So, we expect that a filterable functor should contain a disjunctive type supporting a different number of values of T, including *zero* values. When the filtering operation `.filter(p)` is applied, some values of type T will fail the predicate `p` and will be removed from the collection. The example

```
scala> List(64, 128).filter(_ > 100)
res0: List[Int] = List(128)
```

corresponds to mapping a disjunctive part of type $T\times T$ to a part of type T within List^T in Eq. (9.1).

Consider now a custom data type that implements a given application's business requirements:

Example 9.1.1.1 On a given week, an order (data type A) can be placed on Tuesday and/or on Friday. An order is approved under certain conditions given by a predicate `p: A => Boolean`. Can we represent the order approval by a filtering operation on a suitable data type?

The data type describing a week's orders must describe a possible order on Tuesday (`Option[A]`) and a possible order on Friday (again `Option[A]`). So, we can represent a week's orders as a product $F^A \stackrel{\text{def}}{=} (\mathbb{1}+A)\times(\mathbb{1}+A)$ and implement it as a case class having methods `map` and `withFilter`:

```
final case class Orders[A](tue: Option[A], fri: Option[A]) {
  def map[B](f: A => B): Orders[B] = Orders(tue.map(f), fri.map(f))                  // Functor.
  def withFilter(p: A => Boolean): Orders[A] = Orders(tue.filter(p), fri.filter(p))  // Filterable.
}

scala> Orders(Some(500), Some(2000)).withFilter(_ < 1000) // Approved if the amount is below $1000.
res0: Orders[Int] = Orders(Some(500),None)
```

This code applies filtering independently to both parts of the product. With this definition, we will be able to use `Orders` as a "source" of data in functor blocks:

```
scala> for {
  x <- Orders(Some(500), Some(2000))  // "Source" of type Orders[Int].
  y = x - 200     // Apply discount of $200 to each order.
```

```
  if y < 500      // Orders are approved if the amount is below $500 after discount.
} yield y * 1.10  // Add 10% tax. Result is of type Orders[Double].
res1: Orders[Double] = Orders(Some(330.0), None)
```

Suppose we are considering an additional business rule, such as:

(a) Both orders must be approved, or else no orders can be placed that week.

(b) Both orders can be placed that week if at least one of them is approved.

We could modify the code of `withFilter` to implement one of the rules **(a)** or **(b)**. Will the resulting function still be a "filtering" operation?

We cannot decide this without knowing the mathematical laws that a filtering operation must satisfy. Let us now consider what intuitive expectations we have for the concept of filtering.

9.1.2 The laws of filtering: Motivation and derivation

Computations in a functor block will "make sense" if we easily understand what the program does when we look at the code. Consider this schematic example of a functor block program that uses a filterable functor `List`:

```
1 val result = for {  // Some computations in the context of the ‘List‘ functor.
2   x <- List(...) // For each x in the given list...
3   y = f(x)       // ... compute y
4   if p1(y)       // ... impose condition p1: discard all y for which p1(y) == false
5   if p2(y)       // ... same for condition p2
6   z = g(x, y)    // ... compute z
7   if q(x, y, z)  // ... impose another condition
8 } yield          // For all x in the given list, such that all the conditions hold,
9   k(x, y, z)     // compute the values k, put them into a list, and return as the list ‘result‘.
```

There are several properties that one intuitively expects such programs to have. One property is that computing `y = f(x)` in line 3 and then checking a condition for `y`, such as "`if p1(y)`" in line 4, should be the same as checking the condition `p1(f(x))` and then computing `y = f(x)`: since the code says that `y = f(x)`, we expect the conditions `p1(y)` and `p1(f(x))` to be equivalent.

Translating this equivalence into code, we obtain the requirement that the following two expressions (`result1` and `result2`) should be equal to each other:

```
val result1 = for {
  x <- xs
  y = f(x)
  if p(y)
} yield y
 // Translating the functor block into methods:
val result1 = xs.map(f).filter(p)
```

```
val result2 = for {
  x <- xs
  if p(f(x))
  y = f(x)
} yield y
 // Translating the functor block into methods:
val result2 = xs.filter(x => p(f(x))).map(f)
```

Lines 4–5 of the listing above show two filtering operations, "`if p1(y)`" and "`if p2(y)`", applied one after another. We expect that the first filtering operation keeps only values that satisfy the condition `p1`, and the second filtering operation is applied to the results of the first one, additionally imposing the condition `p2`. So, we expect that applying these two filtering operations is equivalent to filtering by the condition "`if p1(y) && p2(y)`".

We can translate this expectation into equality of the following two code expressions:

```
val result1 = for {
  x <- xs
  if p1(x)
  if p2(x)
} yield x
 // Translating the functor block into methods:
val result1 = xs.filter(p1).filter(p2)
```

```
val result2 = for {
  x <- xs
  if (p1(x) && p2(x))
 } yield x

 // Translating the functor block into methods:
val result2 = xs.filter(x => p1(x) && p2(x))
```

When a filter predicate `p(x)` returns `true` for all `x`, the filtering operation `xs.filter(p)` will never discard any values. So we expect the result to remain the same if we *delete* the line "`if true`" from a functor block program. The corresponding code equivalence can be written as

```
xs.filter(_ => true) == xs
```

Now, suppose a predicate `p(x)` returns `false` for certain values `x`. Then we expect those values `x` to be excluded from any computations performed *after* the line "`if p(x)`". In particular, we should be able to use a partial function safely as long as that function is defined for `x` such that `p(x) == true`. To express this in code, first define a general "factory" for partial functions,

```
def if_p[A, B](p: A => Boolean)(f: A => B): A => B = x => p(x) match { case true => f(x) }
```

This "factory" takes a predicate $p^{:A\to 2}$ and a function $f^{:A\to B}$, and returns a partial function with the same signature, $A \to B$, but defined only for values x for which $p(x) = \text{true}$. Let us denote that function for brevity by $f_{|p}$. Since the `Boolean` type is equivalent to a disjunction of two "named unit types", $2 \cong 1 + 1$ (meaning "`false`" + "`true`"), we can write the code notation for the function $f_{|p}$ as

$$(f^{:A\to B})_{|p} \stackrel{\text{def}}{=} x^{:A} \to p(x) \triangleright \begin{array}{c||c} & B \\ \hline 1\ (\text{false}) & 0 \\ 1\ (\text{true}) & 1 \to f(x) \end{array} \quad . \tag{9.2}$$

The top row contains the void type 0, indicating that the partial function $f_{|p}$ will crash if applied to a value x for which $p(x) = \text{false}$.

Using the partial function $f_{|p}$, we write the last property of the filtering operation as the equality of the expressions `result1` and `result2` defined like this:

```
val result1 = for {
  x <- xs
  if p(x)
  y = f(x)
} yield y
 // Translating the functor block into methods:
val result1 = xs.filter(p).map(f)
```

```
val result2 = for {
  x <- xs
  if p(x)
  y = fp(x) // Defined as val fp = if_p(p)(f)
 } yield y
 // Translating the functor block into methods:
val result2 = xs.filter(p).map(fp)
```

We found 4 requirements for the `filter` function, written in terms of equal code fragments. These requirements are the four "laws" (i.e., equations) that any reasonable `filter` must satisfy. In the code notation, `filter` is filt_F,

$$\text{filt}_F : (A \to 2) \to F^A \to F^A \quad ,$$

and its 4 laws (called the naturality, identity, composition, and partial function laws of `filter`) are

$$\text{naturality law}: \quad f^{\uparrow F} ⨾ \text{filt}_F(p) = \text{filt}_F(f ⨾ p) ⨾ f^{\uparrow F} \quad \text{for } \forall(f^{:A\to B},\ p^{:B\to 2}) \quad . \tag{9.3}$$

$$\text{identity law}: \quad \text{filt}_F(_ \to \text{true}) = \text{id}^{:F^A\to F^A} \quad . \tag{9.4}$$

$$\text{composition law}: \quad \text{filt}_F(p_1) ⨾ \text{filt}_F(p_2) = \text{filt}_F(x \to p_1(x) \wedge p_2(x)) \quad \text{for } \forall(p_1^{:A\to 2},\ p_2^{:A\to 2}) \quad . \tag{9.5}$$

$$\text{partial function law}: \quad \text{filt}_F(p) ⨾ f^{\uparrow F} = \text{filt}_F(p) ⨾ f_{|p}^{\uparrow F} \quad \text{for } \forall(f^{:A\to B},\ p^{:A\to 2}) \quad . \tag{9.6}$$

The following type diagram illustrates the naturality law of `filter`:

$$\begin{tikzcd} F^A \arrow[r, "\text{filt}_F(f^{:A\to B} ⨾ p^{:B\to 2})"] \arrow[d, "(f^{:A\to B})^{\uparrow F}"'] & F^A \arrow[d, "(f^{:A\to B})^{\uparrow F}"] \\ F^B \arrow[r, "\text{filt}_F(p^{:B\to 2})"] & F^B \end{tikzcd}$$

A functor F is called **filterable** if there exists a function filt_F satisfying these four laws.

We may define a typeclass `Filterable` and extension methods `filter` and `withFilter` like this,

```
trait Filterable[F[_]] { def filt[A](p: A => Boolean)(fa: F[A]): F[A] }
implicit class FilterableSyntax[F[_], A](fa: F[A])(implicit ev: Filterable[F]) {
  def filter(p: A => Boolean): F[A] = ev.filt(p)(fa)
  def withFilter(p: A => Boolean): F[A] = filter(p)
}
```

It is intuitively clear why functors such as `Option` and `List` obey the filtering laws: those types can be viewed as "containers" holding zero or more items of data, and the `filter` operation removes all data that fails the filtering condition. What about the custom data type `Orders` from Example 9.1.1.1? In principle, we would need to verify all four laws symbolically, using the code of `withFilter` as we implemented it for `Orders`. Later in this chapter we will see that the four laws can be simplified, reduced to just two laws, and proved more quickly. For now, we can use the `scalacheck` library to implement randomized tests for the four filtering laws:

```
def checkFilteringLaws[F[_] : Filterable : Functor, A, B](implicit
    faEv: Arbitrary[F[A]], fbEv: Arbitrary[F[B]], abEv: Arbitrary[A => B],
    aEv: Arbitrary[A => Boolean], bEv: Arbitrary[B => Boolean]): Assertion = {
  forAll { (f: A => B, p: B => Boolean, fa: F[A]) =>             // Naturality law.
      fa.map(f).filter(p) shouldEqual fa.filter(f andThen p).map(f)
  }
  forAll { (p1: B => Boolean, p2: B => Boolean, fa: F[B]) =>     // Composition law.
      fa.filter(p1).filter(p2) shouldEqual fa.filter(b => p1(b) && p2(b))
  }
  forAll { (fb: F[B]) => fb.filter(_ => true) shouldEqual fb }   // Identity law.

  forAll { (f: A => B, p: A => Boolean, fa: F[A]) =>             // Partial function law.
      fa.filter(p).map(f) shouldEqual fa.filter(p).map[B](x => p(x) match { case true => f(x) })
  }
}
```

Creating a `Filterable` typeclass instance for `Orders` and running the tests will show no errors:

```
implicit val filterableOrders = new Filterable[Orders] {
  def filt[A](p: A => Boolean)(fa: F[A]): F[A] = fa.filter(p)
}
checkFilteringLaws[Orders, Int, String]        // Need to set type parameters to specific types.
```

9.1.3 Examples of non-filterable functors

As usual with typeclasses, the code of the `Filterable` typeclass fixes the type signature of the `filter` function but does not enforce its laws. It is up to the programmer to verify that the implementation of `filter` satisfies the laws.

If we define the filtering operation for the `Orders` data type (see Example 9.1.1.1) with the extra business rule **(a)**, we get

```
implicit val filterableOrdersRuleA = new Filterable[Orders] {
  def filt[A](p: A => Boolean)(fa: F[A]): F[A] =
    if (fa.tue.forall(p) && fa.fri.forall(p))) fa.filter(p)
    else Orders(None, None)        // Rule (a): No orders are approved unless both are approved.
}
checkFilteringLaws[Orders, Int, String]    // Tests pass.
```

However, implementing business rule **(b)** will violate some laws:

```
implicit val filterableOrdersRuleB = new Filterable[Orders] {
  def filt[A](p: A => Boolean)(fa: F[A]): F[A] =
    if (fa.tue.exists(p) || fa.fri.exists(p)) fa      // Here, the value ‘fa‘ remains unchanged.
    else Orders(None, None)   // Rule (b): Both orders are approved if at least one is approved.
}
checkFilteringLaws[Orders, Boolean, Boolean]()      // Tests will fail! A specific failing case:

scala> Orders(Some(500), Some(2000)).filter(x => x < 1000).filter(x => x > 1000)
res0: Orders[Int] = Orders(Some(500),Some(2000))

scala> Orders(Some(500), Some(2000)).filter(x => x < 1000 && x > 1000) // Composition law fails:
res1: Orders[Int] = Orders(None, None)
```

If we implement rule **(b)**, the filtering operation will not correspond to an intuitive understanding of computations in functor blocks:

```
scala> for { x <- Orders(Some(500), Some(2000))
  if x < 1000            // Intuition says that values of x must be below 1000 from now on.
  y = s"Amount: $x"      // So, the value x = 2000 should never appear in this line.
} yield y                // But the final result does not correspond to this intuition:
res2: Orders[String] = Orders(Some("Amount: 500"), Some("Amount: 2000"))
```

This computation violates the partial function law because the value `x = 2000` is not excluded from further computations despite filtering with the predicate "`x < 1000`". This happened because the code of `filter` does not remove the value `x = 2000` from the data structure in that case.

The four laws of filtering are a rigorous formulation of our intuitions about what it means to "filter data". The type `Orders` with business rule **(b)** is an example of a filtering operation that does not correspond to our intuitions and, as a consequence, violates the filtering laws. This does not mean that business rule **(b)** cannot be used in real-world programs; it only means that order approval according to rule **(b)** is not a filtering operation. For instance, applying two order approvals one after another will not give the intuitively expected results. Nevertheless, this may be acceptable in applications where only one order approval is ever applied.

Violations of the filtering laws by business rule **(b)** also does not mean that the functor `Orders` is not filterable. The same functor with business rule **(a)** has a lawful implementation of `filter`.

What are examples of functors that are not filterable? One way of implementing the type signature of the `filter` function is to use the identity function,

```
def filt[A](p: A => Boolean)(fa: F[A]): F[A] = fa  // Ignore the predicate ‘p‘; always return ‘fa‘.
```

This implementation never removes any data and so will violate the partial function law if the functor F^A wraps a value of type A that does not pass the filter. However, the identity function is the *only* possible implementation of `filter` for certain functors F, e.g., the identity functor $F^A \stackrel{\text{def}}{=} A$ or the exponential functor $F^A \stackrel{\text{def}}{=} Z \to A$. So, functors of the form $F^A \stackrel{\text{def}}{=} Z \to A$ (where Z is a fixed type) are not filterable.

The functor $F^A \stackrel{\text{def}}{=} A \times (\mathbb{1} + A)$ is not filterable because the filtering operation $\text{filt}_F(p)$ cannot remove the first value of type A when it does not pass the filter predicate p.

Functors such as $F^A \stackrel{\text{def}}{=} \mathbb{1} + A$ and $F^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$ are filterable because the function filt_F can be defined correctly; but incorrect implementations are also possible. For example, one could imagine defining `filter` for an `Option` type as

```
implicit val wrongFilterableOption = new Filterable[Option] {
  def filt[A](p: A => Boolean)(fa: Option[A]): Option[A] = None // Discard input, always return None.
}
```

This code discards information and violates the identity law: the result of filtering with an identically `true` predicate is not the identity function of type `Option[A] => Option[A]`.

Finally, one could violate the naturality law by defining `filter` in a special way when the type parameter A is set to, say, `Int`. To obey the naturality law, the `filter` function must be fully parametric and must not use hard-coded values of specific types or make decisions based on specific types.

Note that the `Boolean` type is equivalent to $\mathbb{2} \cong \mathbb{1} + \mathbb{1}$; in other words, this type can be expressed via the basic type constructions (disjunction and the `Unit` type) without using any externally defined values. For this reason, it is allowed to use the `Boolean` type in fully parametric functions.

9.1.4 Solved examples: Programming with filterable functors

Example 9.1.4.1 A cluster has 2 servers; each server needs to have valid credentials, which expire periodically. If credentials expire for one server, it may copy valid credentials from the other server. If no server has valid credentials, the cluster is down. Is this setup described by a filterable functor?

Solution Assuming that credentials have type A, we can have two possibilities: both servers have valid credentials, or the cluster is down. The corresponding data type is the functor $F^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$. This functor is filterable if we can implement a lawful `filter` function. Begin writing code for `filter`:

```
1 type F[A] = Option[(A, A)]
2 def filter[A](p: A => Boolean): F[A] => F[A] = {
3     case None              => None
4     case Some((a1, a2))    => ???
5 }
```

In line 4, we need to compute a value of type F^A using the given values `a1` and `a2`. We need to check whether the predicate `p` holds for `a1` and `a2`. What if `p(a1) == false` but `p(a2) == true`? We need to remove `a1` from the result, or else the filtering laws will not hold. But the functor $F^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$ does not allow us to keep just one of the values of type A; it requires two such values or none. So, we may return `Some((a2, a2))` or `None`.

Looking at the business requirements, we see that `p(a1) == false` means the first server's credentials expired. In that case, if `p(a2) == true`, the first server copies the second server's valid credentials, `a2`. So, we must return `Some((a2, a2))`. Other cases are handled similarly. The full code is

```
def filter[A](p: A => Boolean): F[A] => F[A] = {
    case None              => None                    // Cluster is down, no valid credentials.
    case Some((a1, a2))    => (p(a1), p(a2)) match {
        case (true, true)      => Some((a1, a2))  // Both credentials are still valid.
        case (true, false)     => Some((a1, a1))  // Server 2 copies credentials from server 1.
        case (false, true)     => Some((a2, a2))  // Server 1 copies credentials from server 2.
        case (false, false)    => None        // Both credentials expired, the cluster is down.
    }
}
```

It remains to check that the filtering laws hold. The naturality law requires that the equation $f^{\uparrow F} ⨾ \text{filt}_F(p) = \text{filt}_F(f ⨾ p) ⨾ f^{\uparrow F}$ must hold for any values $s^{:F^A}$, $f^{:A\to B}$, and $p^{:B\to 2}$. The code for `fmap` is

```
def fmap[A, B](f: A => B): F[A] => F[B] = {
    case None              => None
    case Some((a1, a2))    => Some((f(a1), f(a2)))
}
```

The code for the left-hand side of the law, $f^{\uparrow F} ⨾ \text{filt}_F(p)$, is written by composing `fmap` and `filter`:

```
fmap(f) andThen filter(p) == {
    case None              => None
    case Some((a1, a2))    => (p(f(a1)), p(f(a2))) match {
        case (true, true)      => Some((f(a1), f(a2)))
        case (true, false)     => Some((f(a1), f(a1)))
        case (false, true)     => Some((f(a2), f(a2)))
        case (false, false)    => None
    }
}
```

The code for the right-hand side of the law, $\text{filt}_F(f ⨾ p) ⨾ f^{\uparrow F}$, is

```
filter(f andThen p) = {
    case None              => None
    case Some((a1, a2))    => (p(f(a1)), p(f(a2))) match {
        case (true, true)      => Some((a1, a2))
        case (true, false)     => Some((a1, a1))
        case (false, true)     => Some((a2, a2))
        case (false, false)    => None
    }
} andThen fmap(f) = {
    case None              => None
    case Some((a1, a2))    => (p(f(a1)), p(f(a2))) match {
        case (true, true)      => Some((f(a1), f(a2)))
        case (true, false)     => Some((f(a1), f(a1)))
        case (false, true)     => Some((f(a2), f(a2)))
        case (false, false)    => None
    }
}
```

Since the code is exactly the same, the law holds.

This computation illustrates why fully parametric functions such as `filter` obey the naturality law: such functions manipulate their arguments purely as symbols of unknown types, without referring

to any specific types or values. Applying a lifted function $f^{\uparrow F}$ before `filter` is the same as inserting `f(...)` around values of type `A` at every place in the code of `filter(p)` where a value of type `A` is used. Applying a lifted function $f^{\uparrow F}$ after `filter` is the same as inserting `f(...)` at every place where a value of type `A` is *returned*; but this does not put `f(...)` around values of type `A` used by the predicate `p`. To compensate, the right-hand side of the naturality law replaces the predicate `p` by the new predicate `f andThen p`, which inserts `f(...)` into the remaining places in the code.

This turns out to be the rule: fully parametric functions always satisfy a suitably formulated naturality law. We will see many more examples of naturality laws in this book, and we will find that the naturality law of a method q often has the form $f^{\uparrow} ⨾ q$ = <modified q> $⨾ f^{\uparrow}$, i.e., the composition of a lifted function $f^{\uparrow}$ and a method q can be interchanged, possibly after some modifications.

The identity law holds because the code of `filter(p)` is the identity function when $p = (_ \to \text{true})$,

```
filter[A](_ => true) == {
    case None              => None
    case Some((a1, a2))    => Some((a1, a2))
} == identity[F[A]]
```

It takes a bit more work to show that the composition law holds. We need to consider two cases: the cluster is down, or both servers have valid credentials.

In the first case, the value of F^A is `None` and remains `None` after any filtering operation. (If the cluster is down, a check of credentials will not bring it up.) So, the composition law holds for that case.

In the second case, we have two credentials a_1, a_2 in a value $s \stackrel{\text{def}}{=} \mathbb{0} + a_1 \times a_2$. The filtering operation $s \triangleright \text{filt}_F(p_1) \triangleright \text{filt}_F(p_2)$ will produce different results according to the values of the predicates p_1 and p_2 applied to a_1 and a_2, which we can summarize as a table:

$p_1(a_1)$	$p_1(a_2)$	$p_2(a_1)$	$p_2(a_2)$	$p_{12}(a_1)$	$p_{12}(a_2)$	$s \triangleright \text{filt}_F(p_1)$	$s \triangleright \text{filt}_F(p_1) \triangleright \text{filt}_F(p_2)$	$s \triangleright \text{filt}_F(p_{12})$
true	true	true	false	true	false	$\mathbb{0} + a_1 \times a_2$	$\mathbb{0} + a_1 \times a_1$	$\mathbb{0} + a_1 \times a_1$
true	false	true	true	true	false	$\mathbb{0} + a_1 \times a_1$	$\mathbb{0} + a_1 \times a_1$	$\mathbb{0} + a_1 \times a_1$
???	???	false	false	false	false	???	$\mathbb{1} + \mathbb{0}^{:A\times A}$	$\mathbb{1} + \mathbb{0}^{:A\times A}$
true	false	false	true	false	false	$\mathbb{0} + a_1 \times a_1$	$\mathbb{1} + \mathbb{0}^{:A\times A}$	$\mathbb{1} + \mathbb{0}^{:A\times A}$
false	false	???	???	false	false	$\mathbb{1} + \mathbb{0}^{:A\times A}$	$\mathbb{1} + \mathbb{0}^{:A\times A}$	$\mathbb{1} + \mathbb{0}^{:A\times A}$

Here we denoted $p_{12} \stackrel{\text{def}}{=} x \to p_1(x) \wedge p_2(x)$ for brevity. We see that the last two columns are always equal, which verifies the composition law. We omitted some rows from the table because the filtering code is completely symmetric with respect to interchanging a_1 and a_2, and because the composition law is trivial when $p_1 = p_2$.

The partial function law holds because the code of `filter` will always remove the value `a1`, or `a2`, or both of them when the filtering predicate `p` returns `false` for any of those values.

So, we have seen that all filtering laws hold. If the program's requirements change, the `filter` function will need to be implemented differently. For instance, the first server might be the only source of credentials: the second server may copy the first server's credentials if needed, but the cluster will go down whenever the first server's credentials expire. This corresponds to the code

```
def filter[A](p: A => Boolean): F[A] => F[A] = {
    case None              => None
    case Some((a1, a2))    => (p(a1), p(a2)) match {
        case (true, true)      => Some((a1, a2))
        case (true, false)     => Some((a1, a1))
        case (false, _)        => None    // The cluster is down if credentials expired for server 1.
    }
}
```

Alternatively, we may have a requirement that copying credentials between servers is not possible:

```
def filter[A](p: A => Boolean): F[A] => F[A] = {
```

```
    case None              => None
    case Some((a1, a2))    => (p(a1), p(a2)) match {
        case (true, true)      => Some((a1, a2))              // Both credentials are valid.
        case _                 => None    // The cluster is down if any credentials expired.
    }
}
```

The filtering laws will still hold with these alternative implementations. We omit the proofs.

Example 9.1.4.2 John can have up to 3 coupons, and Jill up to 2. All of John's coupons must be valid on purchase day, while each of Jill's coupons is checked independently. Implement a filterable functor describing this situation.

Solution We use a type parameter A for the type of "coupons". A data structure holding "up to 3" values of type A is written as $\mathbb{1} + A + A \times A + A \times A \times A$ and can be implemented in Scala by

```
sealed trait JohnsCoupons[A]           // This represents the type 𝟙+A+A×A+A×A×A.
final case class John0[A]()                    extends JohnsCoupons[A]
final case class John1[A](c1: A)               extends JohnsCoupons[A]
final case class John2[A](c1: A, c2: A)        extends JohnsCoupons[A]
final case class John3[A](c1: A, c2: A, c3: A) extends JohnsCoupons[A]
```

This code models John's coupons. Jill's coupons are implemented similarly,

```
sealed trait JillsCoupons[A]    // This represents the type 𝟙+A+A×A.
final case class Jill0[A]()             extends JillsCoupons[A]
final case class Jill1[A](c1: A)        extends JillsCoupons[A]
final case class Jill2[A](c1: A, c2: A) extends JillsCoupons[A]
```

The full data type is the product of John's and Jill's coupons.

```
final case class Coupons[A](johns: JohnsCoupons[A], jills: JillsCoupons[A])
```

It is convenient to define the filtering functions separately for `JohnsCoupons` and for `JillsCoupons`:*all*

```
def filterJohn[A](p: A => Boolean): JohnsCoupons[A] => JohnsCoupons[A] = {
  case John0()             => John0()  // We return John0() unless all coupons are valid:
  case John1(c1)           => if (p(c1)) John1(c1) else John0()
  case John2(c1, c2)       => if (p(c1) && p(c2)) John2(c1, c2) else John0()
  case John3(c1, c2, c3)   => if (p(c1) && p(c2) && p(c3)) John3(c1, c2, c3) else John0()
}
def filterJill[A](p: A => Boolean): JillsCoupons[A] => JillsCoupons[A] = {
  case Jill0() => Jill0()        // We must remove each invalid coupon but keep the rest.
  case Jill1(c1) => if (p(c1)) Jill1(c1) else Jill0()
  case Jill2(c1, c2) => (p(c1), p(c2)) match {
     case (true, true)     => Jill2(c1, c2)
     case (true, false)    => Jill1(c1)
     case (false, true)    => Jill1(c2)
     case (false, false)   => Jill0()
  }
}
```

Now we can define the `filter` function for `Coupons`:

```
def filter[A](p: A => Boolean)(fa: Coupons[A]): Coupons[A] =
  Coupons(filterJohn(p)(fa.johns), filterJill(p)(fa.jills))
```

We will not prove the laws for the `filter` function because its code follows from general constructions derived later in this chapter. Running the `scalacheck` tests shows no failures:

```
implicit val filterableCoupons = new Filterable[Coupons] {
  def filt[A](p: A => Boolean)(fa: Coupons[A]): Coupons[A] = filter(p)(coupons)
}
checkFilteringLaws[Coupons, Int, String]        // Tests pass.
```

Example 9.1.4.3 A server receives a sequence of requests. Each request must be authenticated. Once a non-authenticated request is found, no further requests are accepted. Is this situation described by a filterable functor?

Solution We represent the requests by a `Seq[A]` wrapped in a `Server` type:

```
final case class Server[A](requests: Seq[A])
```

The filtering operation truncates the sequence when the predicate `p` first returns `false`:

```
def filter[A](p: A => Boolean)(s: Server[A]): Server[A] = Server(s.requests.takeWhile(p))
```

We will not prove the laws because this implementation reduces to general constructions of filterable functors (see page 319 below). Intuitively, we expect laws to hold because the `filter` function always removes values that fail the predicate `p`. The filtering function also removes other values that may or may not fail the predicate, but the filtering laws allow removing *more* values.

Example 9.1.4.4 If possible, implement a `Filterable` typeclass instance for:

(a) The functor F^T defined by the Scala code

```
final case class F[T](x: Option[T], yy: Option[(T, T)])
```

(b) $F^A \stackrel{\text{def}}{=} \text{Int} + \text{Int} \times A + \text{Int} \times A \times A + \text{Int} \times A \times A \times A$.

(c) A non-empty list functor, $F^A = \text{NEList}^A$, defined recursively as $F^A \stackrel{\text{def}}{=} A + A \times F^A$.

(d) $F^{Z,A} \stackrel{\text{def}}{=} Z + \text{Int} \times Z \times A \times A$ (with respect to the type parameter A).

(e) $F^{Z,A} \stackrel{\text{def}}{=} Z + \text{Int} \times A \times \text{List}^A$ (with respect to the type parameter A).

Solution **(a)** The functor F is written in the code notation as $F^T \stackrel{\text{def}}{=} (\mathbb{1} + T) \times (\mathbb{1} + T \times T)$. This is a product of `Option[T]`, which is filterable, and the functor $\mathbb{1} + T \times T$, which was shown to be filterable in Example 9.1.4.1. We can apply the corresponding `filter` operation to each part of the product:

```
def filter[T](p: T => Boolean): F[T] => F[T] = {
  case F(t, None)              => F(t.filter(p), None)
  case F(t, Some((t1, t2)))    => F(t.filter(p), if (p(t1) && p(t2)) Some((t1, t2)) else None)
}
```

Each part of the product type satisfies the filtering laws separately (see Statement 9.2.4.2 below).

(b) The functor F is equivalent to $F^A \cong \text{Int} \times (\mathbb{1} + A + A \times A + A \times A \times A) = \text{Int} \times \text{JohnsCoupons}^A$, where we used the functor `JohnsCoupons` from Example 9.1.4.2. So we use the same filtering operation for `JohnsCoupons` as in Example 9.1.4.2, while keeping the `Int` value unchanged:

```
final case class F[A](n: Int, johns: JohnsCoupons[A])
def filter[A](p: A => Boolean)(fa: F[A]): F[A] = fa.copy(johns = filterJohn(p)(johns))
```

An interesting alternative implementation of `filter` uses the integer value `n` for tracking the total number of data items *removed* by filtering:

```
def filter[A](p: A => Boolean)(fa: F[A]): F[A] = {
  val (removed, newJohnsCoupons) = fa.johns match {
    case John0()              => (0, John0())
    case John1(c1)            => if (p(c1)) (0, John1(c1)) else (1, John0())
    case John2(c1, c2)        => if (p(c1) && p(c2)) (0, John2(c1, c2)) else (2, John0())
    case John3(c1, c2, c3)    => if (p(c1) && p(c2) && p(c3)) (0, John3(c1, c2, c3)) else (3, John0())
  }
  F(fa.n + removed, newJohnsCoupons)
}
```

The new code still satisfies the filtering laws (we omit the proof).

(c) The type definition of `NEList` is

```
sealed trait NEList[A]
final case class Last[A](x: A)                  extends NEList[A]
final case class More[A](x: A, tail: NEList[A]) extends NEList[A]
```

We find that we *cannot* implement the type signature of `filter`:

```
def filter[A](p: A => Boolean): NEList[A] => NEList[A] = {
  case Last(x)       => if (p(x)) ??? else ???    // Need to compute a value of type NEList[A] here.
  case More(x, tail) => ???
}
```

The problem is in line 2 above when `p(x) == false`: we need to remove the value `x`, making the list empty, but the type `NEList[A]` disallows empty lists. So, line 2 must always return a list containing the value `x`, which violates the partial function law of filtering. We conclude that `NEList` is not filterable.

(d) Looking at the type expression $F^{Z,A} \stackrel{\text{def}}{=} Z + \text{Int} \times Z \times A \times A$, we need to check whether we could remove the values of type A that do not pass the filter. If none of the two values of type A within $\text{Int} \times Z \times A \times A$ pass the filter, we will need to remove both of them and to return a value of type Z. Luckily, we have a value of type Z within $\text{Int} \times Z \times A \times A$. So we can implement `filter` e.g., like this,

```
type F[A] = Either[Z, (Int, Z, A, A)]           // The type ‘Z‘ must be already defined.
def filter[A](p: A => Boolean): F[A] => F[A] = {
  case Left(z)                  => Left(z)
  case Right((n, z, a1, a2))    => if (p(a1) && p(a2)) Right((n, z, a1, a2))
                                   else Left(z) // If anything fails the filter, use ‘z‘.
}
```

The filtering laws will hold similarly to Example 9.1.4.1.

(e) The `filter` function must be able to remove values of type A from $F^{Z,A} \stackrel{\text{def}}{=} Z + \text{Int} \times A \times \text{List}^A$. What if *no* values of type A pass the filter? If we do not remove all those values of type A from the data structure, we will violate the partial function law. So, the only choice in that case is to return a value of type Z. But Z is a type parameter, and we cannot create values of type Z from scratch. We conclude that $F^{Z,A}$ is not filterable with respect to A.

```
type F[A] = Either[Z, (Int, A, List[A])]
def filter[A](p: A => Boolean): F[A] => F[A] = {
  case Left(z)              => Left(z)
  case Right((n, a, as))    =>  // What to compute in case p(a) == false and as.filter(p) == List() ?
                            ??? // In that case, we will have neither values of type A nor of type Z.
}
```

9.1.5 Exercises: Programming with filterable functors

Exercise 9.1.5.1 Confucius gave a wisdom on each of the 7 days of a week. Sometimes the wise proverbs were hard to remember. If Confucius forgets what he said on a given day, he also forgets what he said on all the *previous* days of the week. Is this situation described by a filterable functor?

Exercise 9.1.5.2 Define an extension method `evenFilter(p)` on the type `IndexedSeq[T]` such that a value `x: T` is kept in the sequence if `p(x) == true` and only if the initial sequence has an *even* total number of elements `y` for which `p(y) == false`. Does `evenFilter` define a lawful filterable functor?

Exercise 9.1.5.3 If possible, implement the `filter` function or a `Filterable` typeclass instance (law checking is optional) for:

(a) The functor $Q^{A,Z}$ with respect to the type parameter A, where $Q^{\bullet,\bullet}$ is defined by this Scala code:

```
final case class Q[A, Z](id: Long, user1: Option[(A, Z)], user2: Option[(A, Z)])
```

(b) The functor R^A defined by the Scala code

```
final case class R[A](x: Int, y: Int, z: A, data: List[A])
```

where the functor `List` already has the method `filter` defined in the standard library.

(c) $F^{P,Q,A} \stackrel{\text{def}}{=} (P \to P) + (Q \to Q) \times A \times A \times A$.

(d) The functor MyTree^A defined recursively as $\text{MyTree}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A \times \text{MyTree}^A \times \text{MyTree}^A$.

Exercise 9.1.5.4 Three times each week (on Mondays, Wednesdays, and Fridays) a data set is collected and a test is run. If a test fails, the corresponding data set may be discarded. Is this situation described by a filterable functor if we impose one of the following additional requirements:

(a) All data sets for a given week are discarded if at least *two* of the tests failed that week?

(b) All data sets for a given week are retained if at least *two* of the tests passed that week?

Exercise 9.1.5.5 Implement a `filter` function for a perfect-shaped tree R^A defined by:

(a) $R^A \stackrel{\text{def}}{=} \mathbb{1} + A + R^{A \times A}$.

(b) $R^A \stackrel{\text{def}}{=} A + R^{(\mathbb{1}+A)\times(\mathbb{1}+A)}$.
Is the simplest perfect-shaped tree R^A defined by $R^A \stackrel{\text{def}}{=} A + R^{A\times A}$ filterable?

9.2 Laws and structure

9.2.1 Simplifying the filtering laws: Motivation for `deflate`

The four laws of `filter` (Section 9.1.2) require considerable work to verify directly. Is there a shorter reformulation of the laws that is easier to understand and to verify?

To motivate such a reformulation, begin by considering a heuristic picture of a filter operation $\text{filt}_F(p) : F^A \to F^A$ that may remove some values of type A from a wrapper F^A. For example, if $F^A \stackrel{\text{def}}{=} \mathbb{1} + A$, the filtering operation may remove the value of type A and return the result $1 + \mathbb{0}$. It appears that the filtering operation, at the type level, replaces the type A by the unit type $\mathbb{1}$ whenever a value does not pass the filtering condition. In examples of non-filterable functors F, such as $F^A \stackrel{\text{def}}{=} A \times (\mathbb{1} + A)$, the type expression for F^A cannot accommodate replacing the type A by $\mathbb{1}$.

If we had the functor $F^{\mathbb{1}+A}$ instead of F^A, we would be able to replace A by $\mathbb{1}$ whenever necessary:

```
def filter[F[_]: Functor, A](p: A => Boolean): F[Option[A]] => F[Option[A]] =
    _.map(_.filter(p))    // Using the standard .filter method on Option.
```

We can always convert a value of type F^A into a value of type $F^{\mathbb{1}+A}$:

```
def inflate[F[_]: Functor, A]: F[A] => F[Option[A]] =
    _.map(x => Some(x))
```

The code notation for the function `inflate` is

$$\text{inflate}^{F,A} \stackrel{\text{def}}{=} (x^{:A} \to \mathbb{0}^{:\mathbb{1}} + x)^{\uparrow F} \quad .$$

It remains to convert $F^{\mathbb{1}+A}$ to F^A. If we could *somehow* do that, say, via a function `deflate`,

```
def deflate[F[_], A]: F[Option[A]] => F[A] = ???
```

$$\text{deflate}^{F,A} : F^{\mathbb{1}+A} \to F^A \quad ,$$

we would then express `filter` through `map` and `deflate` like this,

$$(\text{filt}_F(p))^{:F^A \to F^A} = \text{inflate} ⨾ \left(\text{filt}_{\text{Opt}}(p)\right)^{\uparrow F} ⨾ \text{deflate} \quad .$$

$$F^A \xrightarrow{\text{inflate}} F^{\mathbb{1}+A} \xrightarrow{\left(\text{filt}_{\text{Opt}}(p)\right)^{\uparrow F}} F^{\mathbb{1}+A} \xrightarrow{\text{deflate}} F^A$$

Here filt_{Opt} is the standard `filter` method defined for the `Option[_]` types.

We notice that both functions in the composition $\text{inflate} ⨾ (\text{filt}_{\text{Opt}}(p))^{\uparrow F}$ are some lifted functions in the functor F, and so we can simplify that composition to a single lifted function,

$$\begin{aligned} & \underline{\text{inflate}} ⨾ (\text{filt}_{\text{Opt}}(p))^{\uparrow F} \\ \text{definition of inflate}: \quad & = (x^{:A} \to \mathbb{0}^{:\mathbb{1}} + x)^{\uparrow F} ⨾ (\text{filt}_{\text{Opt}}(p))^{\uparrow F} \\ \text{functor composition law of } F: \quad & = \left(x \to \text{filt}_{\text{Opt}}(p)(\mathbb{0} + x)\right)^{\uparrow F} \quad . \end{aligned}$$

We will need to use this function often, so let us call it $\psi(p)$ or even shorter, ψ_p, for convenience:

```
def psi[A](p: A => Boolean): A => Option[A] =
  x => Some(x).filter(p)
```

$$\psi^A : (A \to \mathbb{2}) \to A \to \mathbb{1} + A \quad ,$$

$$\psi_p \stackrel{\text{def}}{=} \psi(p^{:A\to 2}) \stackrel{\text{def}}{=} x^{:A} \to \text{filt}_{\text{Opt}}(p)(\mathbb{0} + x) \quad .$$

Using the function ψ, we can express the `filter` operation as

$$F^A \xrightarrow{\psi_p^{\uparrow F}} F^{\mathbb{1}+A} \xrightarrow{\text{deflate}} F^A \qquad \text{filt}_F(p) = \psi_p^{\uparrow F} ⨾ \text{deflate} \quad . \tag{9.7}$$

```
def filter[A](p: A => Boolean)(fa: F[A]): F[A] = deflate(fa.map(psi(p)))
```

We derived the code for `filter` assuming that a suitable function `deflate` exists. Can we derive the code for `deflate` for a given filterable functor F, e.g., for F = `Seq`? The required type signature is `deflate: Seq[Option[T]] => Seq[T]`. The Scala library has a `flatten` method for `Seq` with exactly that type signature; it removes all empty `Option` values from a sequence. This shows how to derive the `deflate` method for any filterable F: use F's `filter` to remove the empty `Option` values from $F^{\mathbb{1}+A}$.

```
def deflate[F[_]: Filterable : Functor, A]:
                    F[Option[A]] => F[A] =
    _.filter(_.nonEmpty).map(_.get)
```

$$\text{deflate}: F^{\mathbb{1}+A} \xrightarrow{\text{filt}_F(\text{nonEmpty})} F^{\mathbb{1}+A} \xrightarrow{\text{get}^{\uparrow F}} F^A$$

$$\text{deflate}^{:F^{\mathbb{1}+A}\to F^A} = \text{filt}_F(\text{nonEmpty}) \,;\, \text{get}^{\uparrow F} \quad . \tag{9.8}$$

The method `get` is a partial function and will fail on empty `Option` values (`None`). However, it is safe to use $\text{get}^{\uparrow F}$ here, because the partial function law of filtering guarantees that `.filter(_.nonEmpty)` will prevent `get` from being applied to `None` values. Because of this, `deflate` is a *total* function when defined by Eq. (9.8), as long as F's `filter` obeys its partial function law.

The ability to express `filter` via `deflate` means that a functor's filtering logic is fully described as long as we know how to exclude empty `Option` values from $F^{\mathbb{1}+A}$ and convert the result to F^A. We can define the `Filterable` typeclass through `deflate`, providing `filter` and `deflate` as extension methods:

```
abstract class Filterable[F[_]: Functor] {    // Need a ‘Functor‘ instance to use ‘.map‘.
  def deflate[A]: F[Option[A]] => F[A]   // Typeclass instances will implement ‘deflate‘.
  def filt[A](p: A => Boolean)(fa: F[A]): F[A] = deflate(fa.map(x => Some(x).filter(p)))
} // Typeclass instances do not have to implement ‘filt‘ but may override it for performance reasons.

implicit class FilterableSyntax[F[_]: Filterable, A](fa: F[Option[A]]) {
  def deflate: F[A] = implicitly[Filterable[F]].deflate(fa)
  def withFilter(p: A => Boolean): F[A] = implicitly[Filterable[F]].filt(p)(fa)
}
```

Example 9.2.1.1 Use `deflate` to implement a `Filterable` instance for the functor $F^A \stackrel{\text{def}}{=} Z \to \text{List}^A$.

Solution The type signature of `deflate` is implemented as

```
type F[A] = Z => List[A]    // The type Z should have been defined before.
def deflateF[A](fa: F[Option[A]]): F[A] = { z => fa(z).flatten }
```

We used Scala's library method `flatten` with the type signature `Seq[Option[A]] => Seq[A]`. This method is defined for all sequences, similarly to the `flatten` method that has type `Seq[Seq[A]] => Seq[A]`.

Using this function, we implement the typeclass instance as

```
implicit val filterableF = new Filterable[F] {
  def deflate[A]: F[Option[A]] => F[A] = deflateF
}
```

Example 9.2.1.2 Implement `deflate` for the functor $F^A \stackrel{\text{def}}{=} A \times A + (Z \to Z)$.

Solution The type signature of `deflate` is $F^{\mathbb{1}+A} \to F^A$ and can be implemented e.g., as

```
type F[A] = Either[(A, A), Z => Z]    // The type Z should have been defined before.
def deflateF[A]: F[Option[A]] => F[A] = { // Pattern-match on Either[(Option[A], Option[A]), Z => Z].
  case Left((Some(a1), Some(a2)))   => Left((a1, a2))   // Both values pass the filter.
  case Left(_)                      => Right(identity)  // We can use a fixed value of type Z => Z.
  case Right(zz)                    => Right(zz)
}
```

This code implements a "greedy" filter that requires both values in the pair $A \times A$ to satisfy the predicate. Otherwise, the filter returns the special "empty" value $\mathbb{0}^{:A\times A} + \text{id}^{:Z\to Z}$ of type F^A.

Example 9.2.1.3 Use `deflate` to verify that the functor $F^A \stackrel{\text{def}}{=} A + A \times A \times \text{String}$ is not filterable.

Solution We begin by checking whether the type signature of `deflate` can be implemented:

$$\text{deflate}_F : \mathbb{1} + A + (\mathbb{1} + A) \times (\mathbb{1} + A) \times \text{String} \to A + A \times A \times \text{String} \quad .$$

An immediate problem is that we need to map all disjunctive cases, including $\mathbb{1} + 0 + 0$, into a value of type F^A, which contains values of type A in every disjunctive case. So, implementing `deflate` requires us to produce a value of type A from scratch, $\forall A. \mathbb{1} \to A$, which is impossible in a fully parametric function. Since `deflate` is not implementable, the functor F is not filterable.

These examples show that `deflate` is easier to implement and to reason about than `filter`.

9.2.2 Equivalence of `filter` and `deflate`

We have expressed `filter` through `deflate` by Eq. (9.7) and `deflate` through `filter` by Eq. (9.8). Are the types of `deflate` and `filter` equivalent? It turns out that Eqs. (9.7)–(9.8) are inverses of each other only if we assume certain laws.

Statement 9.2.2.1 Begin with a filterable functor F's function `filter`, define `deflate` through Eq. (9.8), and then define a new function `filter'` via Eq. (9.7):

$$\text{deflate} = \text{filt}_F(\text{nonEmpty}) ⨾ \text{get}^{\uparrow F} \quad , \qquad \text{filter}'(p) = \psi_p^{\uparrow F} ⨾ \text{deflate} \quad .$$

Then `filter'` is equal to `filter`, assuming that the partial function law holds.

Proof We need to show that $\text{filt}_F(p)$ is the same as $\text{filter}'(p)$ for any predicate $p^{:A\to 2}$:

$$\begin{aligned}
&\text{expect to equal } \text{filt}_F(p): && \text{filter}'(p) = \underline{\psi_p^{\uparrow F} ⨾ \text{filt}_F(\text{nonEmpty})} ⨾ \text{get}^{\uparrow F} \\
&\text{naturality law of } \text{filt}_F: && = \text{filt}_F(\psi_p ⨾ \text{nonEmpty}) ⨾ \underline{\psi_p^{\uparrow F} ⨾ \text{get}^{\uparrow F}} \\
&\text{composition law of } F: && = \text{filt}_F(\psi_p ⨾ \text{nonEmpty}) ⨾ (\psi_p ⨾ \text{get})^{\uparrow F} \quad . \qquad (9.9)
\end{aligned}$$

To proceed with the calculation, we need to simplify the two expressions $\psi_p ⨾ \text{nonEmpty}$ and $\psi_p ⨾ \text{get}$. Begin with writing the code for the standard methods `nonEmpty` and `get`, using the equivalent type `Option[Unit]` (i.e., $\mathbb{1}+\mathbb{1}$) instead of `Boolean` (i.e., $\mathbb{2}$):

```
def nonEmpty[A]: Option[A] => Option[Unit] = _.map(_ => ())   // Option[Unit] ≅ Boolean
def get[A]: Option[A] => A = { case Some(a) => a }
```

$$\text{nonEmpty} \stackrel{\text{def}}{=} \begin{array}{|c||cc|}
 & \mathbb{1}\ (\text{false}) & \mathbb{1}\ (\text{true}) \\ \hline
\mathbb{1} & \text{id} & \mathbb{0} \\
A & \mathbb{0} & (_^{:A} \to 1)
\end{array} = \begin{array}{|c||c|}
 & \mathbb{2} \\ \hline
\mathbb{1} & _ \to \text{false} \\
A & _ \to \text{true}
\end{array} \quad , \qquad \text{get}^{:\mathbb{1}+A\to A} \stackrel{\text{def}}{=} \begin{array}{|c||c|}
 & A \\ \hline
\mathbb{1} & \mathbb{0} \\
A & \text{id}
\end{array} \quad . \qquad (9.10)$$

These methods are fully parametric since their code is defined purely through the eight standard code constructions of functional programming (see Section 5.2.2), with no externally defined types such as `Int` or `String`. The function ψ is also fully parametric since we can write it as

```
def psi[A](p: A => Option[Unit]): A => Option[A] = x => p(x).map(_ => x)   // Option[Unit] ≅ Boolean
```

$$\text{view } p^{:A\to 2}(x^{:A}) \text{ as having type } \mathbb{1}+\mathbb{1}: \quad x^{:A} ▷ \psi_p \stackrel{\text{def}}{=} p(x) ▷ \begin{array}{|c||cc|}
 & \mathbb{1} & A \\ \hline
\mathbb{1}\ (\text{false}) & \text{id} & \mathbb{0} \\
\mathbb{1}\ (\text{true}) & \mathbb{0} & 1 \to x
\end{array} \quad .$$

We may write that code equivalently as $\text{nonEmpty} \stackrel{\text{def}}{=} (_^{:A} \to 1)^{\uparrow \text{Opt}}$ and $\psi_p(x) \stackrel{\text{def}}{=} x ▷ p ⨾ (1 \to x)^{\uparrow \text{Opt}}$.

Now we compute the function compositions we need. First, we show that $\psi_p ⨾ \text{nonEmpty} = p$:

$$\begin{aligned}
&\text{expect to equal } x ▷ p: && x^{:A} ▷ \psi_p ⨾ \text{nonEmpty} \\
& && = x ▷ p ⨾ (1 \to x)^{\uparrow \text{Opt}} ⨾ (_^{:A} \to 1)^{\uparrow \text{Opt}} \\
&\text{composition under } {}^{\uparrow \text{Opt}}: && = x ▷ p ⨾ \underline{(1 \to 1)}^{\uparrow \text{Opt}} \\
&\text{identity function of type } \mathbb{1} \to \mathbb{1}: && = x ▷ p ⨾ \underline{(\text{id}^{:\mathbb{1}\to\mathbb{1}})^{\uparrow \text{Opt}}} \\
&\text{identity law of Opt}: && = x ▷ p ⨾ \text{id} = x ▷ p \quad . \qquad (9.11)
\end{aligned}$$

The expression $\psi_p \,{}^{\circ}\!{;}\, \text{get}$ is simplified to the partial function $\text{id}_{|p}$ by matrix composition:

$$\text{definitions of } \psi_p \text{ and get}: \quad \underline{x^{:A} \triangleright \psi_p} \,{}^{\circ}\!{;}\, \underline{\text{get}} = p(x) \triangleright \begin{array}{|c||cc|} & \mathbb{1} & A \\ \hline \mathbb{1}\,(\text{false}) & \text{id} & \mathbb{0} \\ \mathbb{1}\,(\text{true}) & \mathbb{0} & 1 \to x \end{array} \,{}^{\circ}\!{;}\, \begin{array}{|c||c|} & A \\ \hline \mathbb{1} & \mathbb{0} \\ A & \text{id} \end{array}$$

$$\text{matrix composition}: \quad = p(x) \triangleright \begin{array}{|c||c|} & A \\ \hline \mathbb{1}\,(\text{false}) & \mathbb{0} \\ \mathbb{1}\,(\text{true}) & 1 \to x \end{array}$$

$$\text{use Eq. (9.2) as definition of } {}_{|p}: \quad = x \triangleright \text{id}_{|p} \quad . \tag{9.12}$$

The same derivation performed in Scala syntax looks like this:

```
psi(p)(x).get    // Expand the code for ‘psi‘ and ‘get‘:
== p(x) match {
     case false  => None
     case true   => Some(x)
   }) match {
     case Some(x)  => x
   }             // Compute function composition:
         == p(x) match { case true => x }      // Rewrite this code equivalently as
         == x match { case x if p(x) => x }    // x ▷ id|p
```

We can now finish the calculation in Eq. (9.9):

$$\begin{aligned}
\text{expect to equal filt}_F(p): \quad & \text{filter}'(p) = \text{filt}_F(\underline{\psi_p \,{}^{\circ}\!{;}\, \text{nonEmpty}}) \,{}^{\circ}\!{;}\, (\underline{\psi_p \,{}^{\circ}\!{;}\, \text{get}})^{\uparrow F} \\
\text{simplify using Eqs. (9.11)–(9.12)}: \quad & = \text{filt}_F(p) \,{}^{\circ}\!{;}\, (\underline{\text{id}_{|p}})^{\uparrow F} \\
\text{partial function law (9.6) of filt}_F: \quad & = \text{filt}_F(p) \,{}^{\circ}\!{;}\, \underline{(\text{id})^{\uparrow F}} \\
\text{identity law of } F: \quad & = \text{filt}_F(p) \,\underline{{}^{\circ}\!{;}\, \text{id}} = \text{filt}_F(p) \quad .
\end{aligned}$$

So, the new function `filter`′ equals the original `filter` function.

Statement 9.2.2.2 Beginning with a given `deflate` function, define the corresponding `filter` function via Eq. (9.7) and then use Eq. (9.8) to define a new function `deflate`′:

$$\text{filt}_F(p) = \psi_p^{\uparrow F} \,{}^{\circ}\!{;}\, \text{deflate} \quad , \qquad \text{deflate}' = \text{filt}_F(\text{nonEmpty}) \,{}^{\circ}\!{;}\, \text{get}^{\uparrow F} \quad .$$

Then `deflate`′ is equal to `deflate`, assuming that the naturality law (9.14) below holds.

Proof Try showing that the new function `deflate`′ is equal to `deflate`:

$$\text{deflate}' = \text{filt}_F(\text{nonEmpty}) \,{}^{\circ}\!{;}\, \text{get}^{\uparrow F} = \psi^{\uparrow F}_{\text{nonEmpty}} \,{}^{\circ}\!{;}\, \text{deflate} \,{}^{\circ}\!{;}\, \text{get}^{\uparrow F} \overset{?}{=} \text{deflate} \quad . \tag{9.13}$$

The derivation is stuck here: we cannot simplify the last expression unless we can somehow switch the order of function compositions so that $\psi^{\uparrow F}_{\text{nonEmpty}}$ and $\text{get}^{\uparrow F}$ are placed together and the functor composition law of F can be applied. To achieve that, we need a law that switches the order of lifted function compositions around `deflate`. The naturality law (9.3) of `filter` has that form, so we can try deriving a similar naturality law for `deflate`. To switch the order of composition of `deflate` with a lifted function, the law must have the form

$$\begin{tikzcd}
F^{\mathbb{1}+A} \arrow[r, "\text{deflate}"] & F^A \arrow[d, "f^{\uparrow F}"] \\
??? \arrow[r, "\text{deflate}"] & F^B
\end{tikzcd}$$

$$\text{deflate} \,{}^{\circ}\!{;}\, f^{\uparrow F} = (???)^{\uparrow F} \,{}^{\circ}\!{;}\, \text{deflate} \quad ,$$

as illustrated by the type diagram at left. The types match in the right-hand side only if the argument of `deflate` is of type $F^{\mathbb{1}+B}$. So, the law must have the form $\text{deflate} \,{}^{\circ}\!{;}\, f^{\uparrow F} = (???^{:\mathbb{1}+A \to \mathbb{1}+B})^{\uparrow F} \,{}^{\circ}\!{;}\, \text{deflate}$. The typed hole $???^{:\mathbb{1}+A \to \mathbb{1}+B}$ must be filled with a value, say, $g^{:\mathbb{1}+A \to \mathbb{1}+B}$, which is somehow related to f. The only way to obtain g is to lift the function f to the `Option` functor, i.e., to define $g \overset{\text{def}}{=} f^{\uparrow \text{Opt}}$.

So, the **naturality law** of `deflate` is

$$\begin{tikzcd} F^{\mathbb{1}+A} \arrow[r, "\text{deflate}"] \arrow[d, "(f^{\uparrow\text{Opt}})^{\uparrow F}"] & F^A \arrow[d, "f^{\uparrow F}"'] \\ F^{\mathbb{1}+B} \arrow[r, "\text{deflate}"] & F^B \end{tikzcd}$$

$$\text{deflate} ⨾ f^{\uparrow F} = f^{\uparrow\text{Opt}\uparrow F} ⨾ \text{deflate} \quad , \tag{9.14}$$

where $f^{:A\to B}$ is an arbitrary function.

Assuming that this naturality law holds for `deflate`, we can continue the derivation in Eq. (9.13) towards showing that `deflate'` equals `deflate`:

$$\begin{aligned} \text{expect to equal deflate}: \quad & \text{deflate}' = \psi^{\uparrow F}_{\text{nonEmpty}} ⨾ \underline{\text{deflate} ⨾ \text{get}^{\uparrow F}} \\ \text{naturality law (9.14) of deflate}: \quad & = \underline{\psi^{\uparrow F}_{\text{nonEmpty}} ⨾ \text{get}^{\uparrow\text{Opt}\uparrow F}} ⨾ \text{deflate} \\ \text{composition law of } F: \quad & = \left(\psi_{\text{nonEmpty}} ⨾ \text{get}^{\uparrow\text{Opt}}\right)^{\uparrow F} ⨾ \text{deflate} \quad . \end{aligned} \tag{9.15}$$

We now need to perform a separate calculation in order to simplify the function $\psi_{\text{nonEmpty}} ⨾ \text{get}^{\uparrow\text{Opt}}$, which must be a function of type $\mathbb{1} + A \to \mathbb{1} + A$ for the types to match. Writing this calculation in Scala syntax, we obtain (skipping some steps for brevity)

```
psi[Option[A]](nonEmpty)(x).map(get) == (nonEmpty(x) match {
  case false   => None
  case true    => Some(x)
}).map { case Some(y) => y } == // Expand code for ‘nonEmpty‘.
( ( x match {
      case None    => false
      case Some(_) => true
    }) match {
        case false   => None
        case true    => Some(x)  // This will be of the form Some(Some(y)).
}).map { case Some(y) => y } ==  // Compute all function compositions.
  x match {
    case None    => None
    case Some(y) => Some(y)
  } == x  // Identity function applied to ‘x‘.
```

To perform the same calculation in the code notation, we first prepare a formula for the lifting operation $^{\uparrow\text{Opt}}$ of the `Option` functor:

$$(h^{:C\to D})^{\uparrow\text{Opt}} = \begin{array}{|c||cc|} & \mathbb{1} & D \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ C & \mathbb{0} & h \end{array} \quad , \text{ so} \quad \text{get}^{\uparrow\text{Opt}} = \begin{array}{|c||cc|} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ \mathbb{1}+A & \mathbb{0} & \text{get} \end{array} = \begin{array}{|c||cc|} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ \mathbb{1} & \mathbb{0} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} \quad , \tag{9.16}$$

where we expanded the matrix to accommodate the disjunctive type $\mathbb{1} + A$. Then we compute

$$x^{:\mathbb{1}+A} ▷ \psi_{\text{nonEmpty}} = \text{nonEmpty}\,(x) ▷ \begin{array}{|c||cc|} & \mathbb{1} & \mathbb{1}+A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ \mathbb{1} & \mathbb{0} & 1 \to x \end{array}$$

$$\text{definition (9.10) of nonEmpty}: \quad = x ▷ \begin{array}{|c||cc|} & \mathbb{1} & \mathbb{1} \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & _ \to 1 \end{array} ⨾ \begin{array}{|c||cc|} & \mathbb{1} & \mathbb{1}+A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ \mathbb{1} & \mathbb{0} & 1 \to x \end{array}$$

$$\text{matrix composition}: \quad = x ▷ \begin{array}{|c||cc|} & \mathbb{1} & \mathbb{1}+A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & _ \to x \end{array} = x ▷ \begin{array}{|c||ccc|} & \mathbb{1} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} \quad ,$$

where the last matrix uses the fact that x matches the type $\mathbb{0}+A$ in the bottom row. Finally,

$$x^{:\mathbb{1}+A} \triangleright \psi_{\text{nonEmpty}} \fatsemi \text{get}^{\uparrow\text{Opt}} = x \triangleright \begin{array}{|c||ccc|} & \mathbb{1} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} \fatsemi \underline{\text{get}^{\uparrow\text{Opt}}}$$

$$\text{use Eq. (9.16)}: \quad = x \triangleright \begin{array}{|c||ccc|} & \mathbb{1} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} \fatsemi \begin{array}{|c||cc|} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ \mathbb{1} & \mathbb{0} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} = x \triangleright \begin{array}{|c||cc|} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} = x \triangleright \text{id} = x \quad . \tag{9.17}$$

So, we find $\psi_{\text{nonEmpty}} \fatsemi \text{get}^{\uparrow\text{Opt}} = \text{id}$ and thus Eq. (9.15) gives `deflate'` = `deflate`. □

We conclude that the types of `filter` and `deflate` are equivalent as long as the partial function law (9.6) holds for `filter` and the naturality law (9.14) holds for `deflate`.

Statement 9.2.2.3 The partial function law always holds for the `filter` function defined via `deflate`.

Proof Assume that a `deflate` function is given, and define `filter` through Eq. (9.7). Then the partial function law (9.6) is transformed into an equation illustrated by the following diagram:

$$\begin{tikzcd} & F^{\mathbb{1}+A} \arrow[r, "\text{deflate}"] & F^A \arrow[dr, "(f^{:A\to B})^{\uparrow F}"] & \\ F^A \arrow[ur, "\psi_p^{\uparrow F}"] \arrow[dr, "\psi_p^{\uparrow F}"'] & & & F^B \\ & F^{\mathbb{1}+A} \arrow[r, "\text{deflate}"] & F^A \arrow[ur, "(f_{\|p}^{:A\to B})^{\uparrow F}"'] & \end{tikzcd} \qquad \psi_p^{\uparrow F} \fatsemi \text{deflate} \fatsemi f^{\uparrow F} = \psi_p^{\uparrow F} \fatsemi \text{deflate} \fatsemi f_{\|p}^{\uparrow F} \quad .$$

To show that the law holds, we transform both sides using the naturality law (9.14):

$$\text{left-hand side}: \quad \psi_p^{\uparrow F} \fatsemi \underline{\text{deflate} \fatsemi f^{\uparrow F}} = \underline{\psi_p^{\uparrow F} \fatsemi f^{\uparrow\text{Opt}\uparrow F}} \fatsemi \text{deflate} = \left(\psi_p \fatsemi f^{\uparrow\text{Opt}}\right)^{\uparrow F} \fatsemi \text{deflate} \quad .$$

$$\text{right-hand side}: \quad \psi_p^{\uparrow F} \fatsemi \underline{\text{deflate} \fatsemi f_{\|p}^{\uparrow F}} = \underline{\psi_p^{\uparrow F} \fatsemi f_{\|p}^{\uparrow\text{Opt}\uparrow F}} \fatsemi \text{deflate} = \left(\psi_p \fatsemi f_{\|p}^{\uparrow\text{Opt}}\right)^{\uparrow F} \fatsemi \text{deflate} \quad .$$

It remains to show that $\psi_p \fatsemi f_{\|p}^{\uparrow\text{Opt}} = \psi_p \fatsemi f^{\uparrow\text{Opt}}$. Apply the function $\psi_p \fatsemi f^{\uparrow\text{Opt}}$ to an $x^{:A}$,

```
psi(p)(x).map(f) == (p(x) match {
  case false   => None
  case true    => Some(x)
}) match {
  case None    => None
  case Some(y) => Some(f(y))
}  == p(x) match {
  case false   => None
  case true    => Some(f(x))
}
```

The same calculation in the code notation looks like this:

$$x^{:A} \triangleright \psi_p \fatsemi f^{\uparrow\text{Opt}} = p(x) \triangleright \begin{array}{|c||cc|} & \mathbb{1} & A \\ \hline \mathbb{1}\ (\text{false}) & \text{id} & \mathbb{0} \\ \mathbb{1}\ (\text{true}) & \mathbb{0} & 1 \to x \end{array} \fatsemi \begin{array}{|c||cc|} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & f \end{array} = p(x) \triangleright \begin{array}{|c||cc|} & \mathbb{1} & A \\ \hline \mathbb{1}\ (\text{false}) & \text{id} & \mathbb{0} \\ \mathbb{1}\ (\text{true}) & \mathbb{0} & 1 \to f(x) \end{array} \quad .$$

In the last expression, the function f is applied to x only when $p(x) = \text{true}$. So, the result will be the same if we replace $f(x)$ by the partial function $f_{\|p}(x)$, which was defined by Eq. (9.2) to be equal to $f(x)$ when $p(x)$ holds. It follows that $\psi_p \fatsemi f_{\|p}^{\uparrow\text{Opt}} = \psi_p \fatsemi f^{\uparrow\text{Opt}}$, concluding the proof. □

The equivalence between `filter` and `deflate` extend to their respective naturality laws. In other words, the naturality laws for `filter` and `deflate` hold at the same time. We only need to check the

naturality law for one of these two functions; the other naturality law will then hold automatically. The following statement proves this equivalence in one direction:

Statement 9.2.2.4 If the naturality law (9.3) holds for `filter`, the law (9.14) will also hold for `deflate` if it is defined through `filter` by Eq. (9.8).

Proof Compare the two sides of the law (9.14) if `deflate` is defined by Eq. (9.8):

$$\begin{aligned}
\text{left-hand side of Eq. (9.14)}: \quad & f^{\uparrow\text{Opt}\uparrow F} \fatsemi \underline{\text{deflate}} = \underline{f^{\uparrow\text{Opt}\uparrow F} \fatsemi \text{filt}_F(\text{nonEmpty})} \fatsemi \text{get}^{\uparrow F} \\
\text{naturality law (9.3) of filt}_F: \quad & = \text{filt}_F(f^{\uparrow\text{Opt}} \fatsemi \text{nonEmpty}) \fatsemi f^{\uparrow\text{Opt}\uparrow F} \fatsemi \text{get}^{\uparrow F} \quad . \\
\text{right-hand side of Eq. (9.14)}: \quad & \underline{\text{deflate}} \fatsemi f^{\uparrow F} = \text{filt}_F(\text{nonEmpty}) \fatsemi \text{get}^{\uparrow F} \fatsemi f^{\uparrow F} \quad .
\end{aligned}$$

The two sides will be equal if we prove that $f^{\uparrow\text{Opt}} \fatsemi \text{nonEmpty} = \text{nonEmpty}$ and that $f^{\uparrow\text{Opt}} \fatsemi \text{get} = \text{get} \fatsemi f$, which can be viewed as the two naturality laws specific to these functions. Use the definitions (9.10) of `get` and `nonEmpty`, set the type parameters as needed to match the types, and compute:

$$f^{\uparrow\text{Opt}} \fatsemi \text{nonEmpty} = \begin{array}{c||cc|} & \mathbb{1} & B \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & f \end{array} \fatsemi \begin{array}{c||c|} & 2 \\ \hline \mathbb{1} & _ \to \text{false} \\ B & _ \to \text{true} \end{array} = \begin{array}{c||c|} & 2 \\ \hline \mathbb{1} & _ \to \text{false} \\ A & _ \to \text{true} \end{array} = \text{nonEmpty} \quad ,$$

$$f^{\uparrow\text{Opt}} \fatsemi \text{get} = \begin{array}{c||cc|} & \mathbb{1} & B \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & f \end{array} \fatsemi \begin{array}{c||c|} & B \\ \hline \mathbb{1} & \mathbb{0} \\ B & \text{id} \end{array} = \begin{array}{c||c|} & B \\ \hline \mathbb{1} & \mathbb{0} \\ A & f \end{array} \quad ,$$

$$\text{get} \fatsemi f = \begin{array}{c||c|} & A \\ \hline \mathbb{1} & \mathbb{0} \\ A & \text{id} \end{array} \fatsemi f^{:A\to B} = \begin{array}{c||c|} & B \\ \hline \mathbb{1} & \mathbb{0} \\ A & \text{id} \fatsemi f \end{array} = \begin{array}{c||c|} & B \\ \hline \mathbb{1} & \mathbb{0} \\ A & f \end{array} = f^{\uparrow\text{Opt}} \fatsemi \text{get} \quad .$$

□

We have just shown that the naturality law of `filter` guarantees that the naturality law of `deflate` holds. To show that the naturality laws of `filter` and `deflate` are equivalent, it remains to show that the converse statement is also true:

Statement 9.2.2.5 If the naturality law (9.14) holds for `deflate` and the function `filter` is defined via `deflate` by Eq. (9.7) then the naturality law (9.3) holds for `filter`.

Proof Begin by writing the two sides of the naturality law (9.3):

$$\begin{aligned}
\text{left-hand side of Eq. (9.3)}: \quad & f^{\uparrow F} \fatsemi \underline{\text{filt}(p)} = \underline{f^{\uparrow F} \fatsemi \psi_p^{\uparrow F}} \fatsemi \text{deflate} = (f \fatsemi \psi_p)^{\uparrow F} \fatsemi \text{deflate} \quad . \\
\text{right-hand side of Eq. (9.3)}: \quad & \text{filt}(f \fatsemi p) \fatsemi f^{\uparrow F} = \psi_{(f \fatsemi p)}^{\uparrow F} \fatsemi \underline{\text{deflate} \fatsemi f^{\uparrow F}} \\
\text{naturality law (9.14) of deflate}: \quad & = \underline{\psi_{(f \fatsemi p)}^{\uparrow F} \fatsemi f^{\uparrow\text{Opt}\uparrow F}} \fatsemi \text{deflate} = (\psi_{(f \fatsemi p)} \fatsemi f^{\uparrow\text{Opt}})^{\uparrow F} \fatsemi \text{deflate} \quad .
\end{aligned}$$

The remaining difference is in the order of composition of f and ψ. The proof will be complete if we show that, for any $f^{:A\to B}$ and $p^{:B\to 2}$,

$$f \fatsemi \psi_p = \psi_{(f \fatsemi p)} \fatsemi f^{\uparrow\text{Opt}} \quad . \tag{9.18}$$

This equation can be viewed as a naturality law specific to the function ψ. To prove Eq. (9.18), it is convenient to use a definition of ψ_p that represents p as a function of type $A \to \mathbb{1} + \mathbb{1} \cong A \to \text{Opt}^{\mathbb{1}}$,

$$x^{:A} \triangleright \psi_p \stackrel{\text{def}}{=} x^{:A} \triangleright p^{:A\to\text{Opt}^{\mathbb{1}}} \fatsemi (1 \to x)^{\uparrow\text{Opt}} \quad . \tag{9.19}$$

Using this definition of ψ_p, we can derive Eq. (9.18) by applying both sides to an $x^{:A}$,

$$\begin{aligned}
\text{left-hand side}: \quad & x \triangleright f \,;\, \psi_p = x \triangleright f \triangleright \psi_p \\
\text{use Eq. (9.19)}: \quad & = x \triangleright f \triangleright p \,;\, (1 \to x \triangleright f)^{\uparrow \text{Opt}} = x \triangleright f \,;\, p \,;\, (1 \to x \triangleright f)^{\uparrow \text{Opt}} \quad . \\
\text{right-hand side}: \quad & x \triangleright \psi_{(f \,;\, p)} \,;\, f^{\uparrow \text{Opt}} \\
\text{use Eq. (9.19)}: \quad & = x \triangleright f \,;\, p \,;\, \underline{(1 \to x)^{\uparrow \text{Opt}} \,;\, f^{\uparrow \text{Opt}}} \\
\text{composition law of Opt}: \quad & = x \triangleright f \,;\, p \,;\, \big(\underline{(1 \to x) \,;\, f}\big)^{\uparrow \text{Opt}} \\
\text{compute composition}: \quad & = x \triangleright f \,;\, p \,;\, (\underline{1 \to x \triangleright f})^{\uparrow \text{Opt}} \quad .
\end{aligned}$$

9.2.3 Motivation and laws for `liftOpt`

In several derivations we just saw, the function `deflate` was composed with $\psi_p^{\uparrow F}$ or another lifted function that mapped types as $A \to \mathbb{1} + A$. This suggests considering a more general type signature, $f^{:A \to \mathbb{1}+B}$, and composing $f^{\uparrow F}$ with `deflate` into a function that maps $F^A \to F^B$. It turns out that the resulting function, which we will call `liftOpt` and denote by liftOpt_F or simply by liftOpt,

```
def liftOpt_F[A, B](f: A => Option[B]): F[A] => F[B] = _.map(f).deflate
```

$$\text{liftOpt}_F^{A,B}(f^{:A \to \mathbb{1}+B}) \stackrel{\text{def}}{=} f^{\uparrow F} \,;\, \text{deflate}_F \quad , \tag{9.20}$$

has simpler laws and is particularly convenient for symbolic computations.

The name `liftOpt` ("lifting from `Option`") is motivated by the type signature,

$$\text{liftOpt}_F^{A,B} : (A \to \mathbb{1} + B) \to F^A \to F^B \quad ,$$

This lifts a function of type `A => Option[B]` into a function of type `F[A] => F[B]`. Except for using a "twisted" type $A \to \mathbb{1} + B$ instead of $A \to B$, this is similar to lifting via the `fmap` function,

$$\text{fmap}_F^{A,B} : (A \to B) \to F^A \to F^B \quad .$$

As this chapter will show, similarities between `liftOpt` and `fmap` go well beyond type signatures.

We will now derive some properties of `liftOpt`. The definition of `liftOpt` through `map` and `deflate` is illustrated in the diagram at left. Since f is arbitrary in Eq. (9.20), we may set $f = \text{id}^{:A \to A}$ and the type parameter $A = \mathbb{1} + B$ to find

$$\begin{tikzcd}
F^A \arrow[r, "(f^{:A \to \mathbb{1}+B})^{\uparrow F}"] \arrow[dr, "\text{liftOpt}\,(f) \stackrel{\text{def}}{=}"'] & F^{\mathbb{1}+B} \arrow[d, "\text{deflate}"] \\
& F^B
\end{tikzcd}$$

$$\text{liftOpt}^{\mathbb{1}+B,B}(\text{id}^{:\mathbb{1}+B \to \mathbb{1}+B}) = \underline{\text{id}^{\uparrow F}} \,;\, \text{deflate} = \text{deflate} \quad . \tag{9.21}$$

This expresses `deflate` through `liftOpt`. Are these two functions equivalent?

Statement 9.2.3.1 The types of functions `liftOpt` and `deflate` are equivalent via Eqs. (9.20)–(9.21), assuming that a naturality law (Eq. (9.23) below) holds for `liftOpt`.

Proof We need to show that the equivalence of `liftOpt` and `deflate` holds in both directions:

(a) Given a `deflate` function, compute a `liftOpt` function via Eq. (9.20) and then a new `deflate'` function via Eq. (9.21). Then the new `deflate'` will be the same function as the initial `deflate`.

(b) Given a `liftOpt` function, compute a `deflate` function via Eq. (9.21) and then a new `liftOpt'` function via Eq. (9.21). The new `liftOpt'` will be the same as the initial `liftOpt`, assuming Eq. (9.23).

Proof for **(a)** directly derives the formula `deflate'` = `deflate` by this calculation:

$$\begin{aligned}
\text{use Eq. (9.21) to define deflate}': \quad & \text{deflate}' = \text{liftOpt}\,(\text{id}) \\
\text{use Eq. (9.20) to define liftOpt}: \quad & = \underline{\text{id}^{\uparrow F}} \,;\, \text{deflate} = \text{deflate} \quad .
\end{aligned}$$

Proof for **(b)** begins by expressing `liftOpt'` through the initial `liftOpt`:

$$\text{liftOpt}'(f) = f^{\uparrow F} \,;\, \text{deflate} = f^{\uparrow F} \,;\, \text{liftOpt}\,(\text{id}) \quad . \tag{9.22}$$

If nothing is known about `liftOpt`, the calculation will get stuck at this point. To proceed, we need to assume that `liftOpt` obeys a law that switches the order of function compositions around `liftOpt` and allows us to pull f inside of `liftOpt` in the equation above. Laws that switch the order of lifted function compositions are often naturality laws, as we have already seen. So, let us derive a suitable naturality law for `liftOpt`, beginning with a `liftOpt` function defined via `deflate`:

$$(h^{:A\to B})^{\uparrow F} \fatsemi \text{liftOpt}^{B,C}(f^{:B\to\mathbb{1}+C}) = h^{\uparrow F} \fatsemi f^{\uparrow F} \fatsemi \text{deflate} = (h \fatsemi f)^{\uparrow F} \fatsemi \text{deflate} = \text{liftOpt}^{A,C}(h \fatsemi f) \quad .$$

$$\begin{tikzcd} F^A \arrow[d, "h^{\uparrow F}"'] \arrow[dr, "\text{liftOpt}\,(h \fatsemi f)"] & \\ F^B \arrow[r, "\text{liftOpt}\,(f)"] & F^C \end{tikzcd}$$

It follows that *if* `liftOpt` were defined via `deflate` then `liftOpt` would automatically satisfy the naturality law (see diagram at left)

$$(h^{:A\to B})^{\uparrow F} \fatsemi \text{liftOpt}_F^{B,C}(f^{:B\to\mathbb{1}+C}) = \text{liftOpt}_F^{A,C}(h^{:A\to B} \fatsemi f^{:B\to\mathbb{1}+C}) \quad . \tag{9.23}$$

This motivates *imposing* that law on `liftOpt`. We can then finish the proof, resuming from Eq. (9.22),

$$\text{expect to equal liftOpt}\,(f): \quad \text{liftOpt}'(f) = \underline{f^{\uparrow F}} \fatsemi \text{liftOpt}\,(\underline{\text{id}}) = \text{liftOpt}\,(\underline{f \fatsemi \text{id}}) = \text{liftOpt}\,(f) \quad .$$

Since `deflate` is equivalent to `filter` (Statement 9.2.2.2), it follows that `filter` is equivalent to `liftOpt`. The next step is to translate the laws of `filter` into the corresponding laws for `liftOpt`. Do the laws become simpler when formulated for `liftOpt`?

We have already seen that the partial function law of `filter` is satisfied automatically when `filter` is defined via `deflate` (Statement 9.2.2.3). It appears that `deflate` has only 3 laws while `filter` has 4. We will now show that `liftOpt` has only 2 laws, and yet they are *equivalent* to the 4 laws of `filter`.

To see that, we first express `filter` through `deflate` and then finally through `liftOpt`:

$$\text{filt}\,(p^{:A\to 2}) = \psi_p^{\uparrow F} \fatsemi \text{deflate} = \text{liftOpt}\,(\psi_p) \quad . \tag{9.24}$$

Conversely, `liftOpt` is expressed via `filter` like this,

$$\text{liftOpt}^{A,B}(f^{:A\to\mathbb{1}+B}) \stackrel{\text{def}}{=} f^{\uparrow F} \fatsemi \text{deflate} = f^{\uparrow F} \fatsemi \text{filt}\,(\psi_{\text{nonEmpty}}) \quad , \tag{9.25}$$

Identity law Let us now translate `filter`'s identity law (9.4) into the corresponding law for `liftOpt`. Begin by expressing filt (_ → true) via `liftOpt` using Eq. (9.24):

$$\text{filt}\,(_ \to \text{true}) = \text{liftOpt}\,(\psi_{(_\to\text{true})}) = \text{liftOpt}\,(x^{:A} \to \mathbb{0} + x) \quad .$$

The function $\psi_{(_\to\text{true})}$ is equivalent to a simpler function $x^{:A} \to \mathbb{0} + x$ (i.e., `x => Some(x)` in Scala),

$$\begin{aligned}
&\text{use Eq. (9.19)}: && x^{:A} \triangleright \psi_{(_\to\text{true})} = x^{:A} \triangleright (_^{:A} \to \underline{\text{true}^{:\text{Opt}^{\mathbb{1}}}}) \fatsemi (1 \to x)^{\uparrow\text{Opt}} \\
&\text{use equivalence } \mathbb{0}+1 \cong \text{true}^{:\text{Opt}^{\mathbb{1}}}: && = \underline{x^{:A} \triangleright (_^{:A} \to \mathbb{0}+1)} \fatsemi (1 \to x)^{\uparrow\text{Opt}} \\
&\text{apply function to } x: && = (\mathbb{0}+1) \triangleright (1 \to x)^{\uparrow\text{Opt}} = \left\| \begin{array}{cc} \mathbb{0} & 1 \end{array} \right| \triangleright \left| \begin{array}{c||cc} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ \mathbb{1} & \mathbb{0} & 1 \to x \end{array} \right| \\
&\text{substitute the row into the matrix}: && = \left\| \begin{array}{cc} \mathbb{0} & x \end{array} \right| = \mathbb{0} + x^{:A} \quad .
\end{aligned} \tag{9.26}$$

So, we expect the **identity law** of `liftOpt` to be

$$\text{liftOpt}^{A,A}(x^{:A} \to \mathbb{0} + x) = \text{id}^{:F^A \to F^A} \quad . \tag{9.27}$$

Statement 9.2.3.2 **(a)** If `filter` obeys its naturality law (9.3) and identity law (9.4) and `liftOpt` is defined via Eq. (9.25) then `liftOpt` obeys its identity law (9.27).

(b) If `liftOpt` obeys its identity law (9.27) then `filter`, defined via Eq. (9.24), obeys its law (9.4).

Proof **(a)** Compute the identity law of `liftOpt`:

$$\begin{aligned}
\text{expect to equal } \mathrm{id}^A: \quad & \mathrm{liftOpt}\,(x^{:A} \to \mathbb{0}+x) = \underline{(x^{:A} \to \mathbb{0}+x)^{\uparrow F} \,⨾\, \mathrm{filt}^{\mathbb{1}+A}}(\mathrm{nonEmpty}) \,⨾\, \mathrm{get}^{\uparrow F} \\
\text{naturality law (9.3)}: \quad & = \mathrm{filt}^A\big(\underline{(x^{:A} \to \mathbb{0}+x) \,⨾\, \mathrm{nonEmpty}}\big) \,⨾\, (x^{:A} \to \mathbb{0}+x)^{\uparrow F} \,⨾\, \mathrm{get}^{\uparrow F} \\
\text{compute composition}: \quad & = \underline{\mathrm{filt}^A(x^{:A} \to \mathrm{true})} \,⨾\, (x^{:A} \to \mathbb{0}+x)^{\uparrow F} \,⨾\, \mathrm{get}^{\uparrow F} \\
\text{identity law (9.4)}: \quad & = \mathrm{id}^A \,⨾\, \underline{(x^{:A} \to \mathbb{0}+x)^{\uparrow F} \,⨾\, \mathrm{get}^{\uparrow F}} = \big(\underline{(x^{:A} \to \mathbb{0}+x) \,⨾\, \mathrm{get}}\big)^{\uparrow F} \\
\text{compute composition}: \quad & = (\mathrm{id}^A)^{\uparrow F} = \mathrm{id}^A \quad .
\end{aligned}$$

(b) Compute the identity law of `filter` using Eq. (9.26):

$$\begin{aligned}
\text{use Eq. (9.24)}: \quad & \mathrm{filt}\,(_ \to \mathrm{true}) = \mathrm{liftOpt}\,(\psi_{(_\to \mathrm{true})}) \\
\text{use Eq. (9.26)}: \quad & = \mathrm{liftOpt}\,(x \to \mathbb{0}+x) \\
\text{use Eq. (9.27)}: \quad & = \mathrm{id} \quad .
\end{aligned}$$

This completes the proof.

The function $x^{:A} \to \mathbb{0}+x$ plays the role of the `pure` method for the `Option` type, if we view `Option[_]` as a pointed functor (see Section 8.3.5). Denote that `pure` method for brevity by $\mathrm{pu}_{\mathrm{Opt}}$:

$$\mathrm{pu}_{\mathrm{Opt}}^{:A\to\mathbb{1}+A} \stackrel{\text{def}}{=} x^{:A} \to \mathbb{0}+x \quad .$$

Then `liftOpt`'s identity law (9.27) is written more concisely as

$$\mathrm{liftOpt}_F\,(\mathrm{pu}_{\mathrm{Opt}}) = \mathrm{id} \quad . \tag{9.28}$$

We can combine Eq. (9.28) and naturality law (9.23) into a single law if we compose the identity law with a lifted arbitrary function $f^{:A\to B}$:

$$f^{\uparrow F} \,⨾\, \mathrm{liftOpt}_F\,(\mathrm{pu}_{\mathrm{Opt}}) = \mathrm{liftOpt}_F\,(f \,⨾\, \mathrm{pu}_{\mathrm{Opt}}) \quad .$$

So the "combined" **naturality-identity law** of `liftOpt` is

$$\mathrm{liftOpt}_F\,(f^{:A\to B} \,⨾\, \mathrm{pu}_{\mathrm{Opt}}^B) = f^{\uparrow F} \quad . \tag{9.29}$$

Composition law Next, we translate Eq. (9.5) into a corresponding law of `liftOpt`. That law needs to combine two predicates p_1 and p_2 into a new predicate $p(x) \stackrel{\text{def}}{=} p_1(x) \wedge p_2(x)$. As Eq. (9.24) shows, `liftOpt` uses a predicate p only through the function ψ_p. So, we will derive the composition law of `liftOpt` if we somehow express ψ_p as a combination of ψ_{p_1} and ψ_{p_2}. Begin by writing

```
psi(p) == { x => Some(x).filter(p) } == { x => Some(x).filter(p1).filter(p2) }
```

$$\psi_p = x^{:A} \to (\mathbb{0}+x) \triangleright \mathrm{filt}_{\mathrm{Opt}}(p) = x^{:A} \to (\mathbb{0}+x) \triangleright \mathrm{filt}_{\mathrm{Opt}}(p_1) \triangleright \mathrm{filt}_{\mathrm{Opt}}(p_2) \quad .$$

We need to transform this code into some sort of combination of the functions ψ_{p_1} and ψ_{p_2}:

```
psi(p1) == { x => Some(x).filter(p1) }
psi(p2) == { x => Some(x).filter(p2) }
```

Since the value `Some(x).filter(p1)` is still of type `Option[A]`, we may apply `.map(psi(p2))` to that value:

```
x => Some(x).filter(p1).map(y => Some(y).filter(p2)) // Type is Option[Option[A]].
```

Except for the type `Option[Option[A]]`, the result is correct: a value `x:A` will be present within the `Option[Option[A]]` wrapper only if *both* `p1(x)` and `p2(x)` return `true`. To convert the result to the required type `Option[A]`, we apply `Option`'s method `flatten`:

```
psi(p) == x => Some(x).filter(p1).map { y => Some(y).filter(p2) }.flatten    // Use flatMap instead.
       == x => Some(x).filter(p1).flatMap { y => Some(y).filter(p2) }
       == psi(p1) andThen (_.flatMap(psi(p2)))     // Using standard flatten and flatMap for Option.
```

Denote this combination of the functions ψ_{p_1} and ψ_{p_2} by the symbol $\diamond_{\text{Opt}}$, so that we may write

$$\psi_p = \psi_{p_1} \diamond_{\text{Opt}} \psi_{p_2} \stackrel{\text{def}}{=} x^{:A} \to x \triangleright \psi_{p_1} \triangleright \text{flm}_{\text{Opt}}(\psi_{p_2}) = \psi_{p_1} ⨾ (y \to y \triangleright \text{flm}_{\text{Opt}}(\psi_{p_2})) \quad .$$

We use the symbol flm_{Opt} for `Option`'s `flatMap`; the Scala code `x.flatMap(f)` is denoted by $x \triangleright \text{flm}_{\text{Opt}}(f)$ if we view flm_{Opt} as a curried function with the type signature

$$\text{flm}_{\text{Opt}} : (A \to \text{Opt}^B) \to \text{Opt}^A \to \text{Opt}^B \quad .$$

The operation $\diamond_{\text{Opt}}$ is a special kind of composition called **Kleisli**[1] **composition**. It applies to functions such as ψ_p that have type $A \to \mathbb{1} + A$; such functions cannot be composed with the ordinary function composition ($\psi_{p_1} ⨾ \psi_{p_2}$ does not type-check). It is straightforward to extend the operation $\diamond_{\text{Opt}}$ from functions of type $A \to \mathbb{1} + A$ to functions with more general types, $A \to \mathbb{1} + B$ and $B \to \mathbb{1} + C$:

```
def kleisliOpt[A, B](f: A => Option[B], g: B => Option[C]): A => Option[C] =
  { x: A => f(x).flatMap(g) }      // Using the standard flatMap for Option.
```

$$f^{:A \to \mathbb{1}+B} \diamond_{\text{Opt}} g^{:B \to \mathbb{1}+C} \stackrel{\text{def}}{=} x^{:A} \to f(x) \triangleright \text{flm}_{\text{Opt}}(g) \quad .$$

The Kleisli composition $f \diamond_{\text{Opt}} g$ yields a function of type $A \to \mathbb{1} + C$ and is similar to the ordinary composition $f ⨾ g$ except for using "twisted" types, e.g., $A \to \text{Opt}^B$ instead of $A \to B$. (The "twisted" functions cannot be composed as $f ⨾ g$ because their types do not match.)

We can now derive the composition law of `liftOpt` starting from Eq. (9.5):

$$\begin{aligned} &\text{left-hand side :} \quad \text{filt}(p_1) ⨾ \text{filt}(p_2) = \text{liftOpt}(\psi_{p_1}) ⨾ \text{liftOpt}(\psi_{p_2}) \quad . \\ &\text{right-hand side :} \quad \text{filt}(p) = \text{liftOpt}(\psi_p) = \text{liftOpt}(\psi_{p_1} \diamond_{\text{Opt}} \psi_{p_2}) \quad . \end{aligned}$$

If `filter`'s composition law holds, we obtain (without any additional assumptions) the equation

$$\text{liftOpt}(\psi_{p_1}) ⨾ \text{liftOpt}(\psi_{p_2}) = \text{liftOpt}(\psi_{p_1} \diamond_{\text{Opt}} \psi_{p_2}) \quad . \tag{9.30}$$

This looks like a law typical for a "lifting": the composition of lifted functions is equal to the lifted Kleisli composition. So, it appears useful to formulate the composition law of `liftOpt` in a more general way by allowing arbitrary $f^{:A \to \mathbb{1}+B}$ and $g^{:B \to \mathbb{1}+C}$ instead of specific functions ψ_{p_1} and ψ_{p_2}.

$$\begin{tikzcd} & F^B \arrow[dr, "\text{liftOpt}(g)"] & \\ F^A \arrow[ur, "\text{liftOpt}(f)"] \arrow[rr, "\text{liftOpt}(f \diamond_{\text{Opt}} g)"'] & & F^C \end{tikzcd}$$

The composition law of `liftOpt` is then written as

$$\text{liftOpt}_F(f^{:A \to \mathbb{1}+B}) ⨾ \text{liftOpt}_F(g^{:B \to \mathbb{1}+C}) = \text{liftOpt}_F(f \diamond_{\text{Opt}} g) \quad . \tag{9.31}$$

Because this law holds for f and g involving arbitrary types A, B, C, it is stronger than `filter`'s composition law. We will now show that `deflate`'s naturality law (9.14) and hence `filter`'s naturality law (9.3) can be derived from Eqs. (9.29)–(9.31) if we choose f and g in a special way, such that one of f or g always returns a non-empty `Option` value:

Statement 9.2.3.3 **(a)** The naturality law (9.23) of `liftOpt` follows from Eqs. (9.29)–(9.31).

(b) If `deflate` is defined via `liftOpt`, the naturality law (9.14) follows from Eqs. (9.29)–(9.31).

Proof **(a)** Choose functions $f^{:A \to \mathbb{1}+B}$ to be of the form

$$\begin{aligned} f^{:A \to \mathbb{1}+B} &\stackrel{\text{def}}{=} h^{:A \to B} ⨾ \text{pu}_{\text{Opt}}^{:B \to \mathbb{1}+B} \\ \text{or, written out in more detail :} \quad &= h^{:A \to B} ⨾ (x^{:B} \to \mathbb{0} + x) = x^{:A} \to \mathbb{0} + h(x) \quad , \end{aligned}$$

[1] The Swiss-German name Kleisli is pronounced "cli-slee" ("cli" as in "climb").

where $h^{:A\to B}$ is an arbitrary function, and use those f in Eq. (9.31):

$$\text{liftOpt}\,(f \diamond_{\text{Opt}} g) = \underline{\text{liftOpt}\,(h \,⨾\, \text{pu}_{\text{Opt}})} \,⨾\, \text{liftOpt}\,(g)$$

$$\text{use Eq. (9.29)}: \quad = h^{\uparrow F} \,⨾\, \text{liftOpt}\,(g) \quad .$$

To simplify this formula, we need to compute $f \diamond_{\text{Opt}} g$. The Kleisli composition $\diamond_{\text{Opt}}$ is defined via the standard `flatMap` method of the `Option` type. With the notation flm_{Opt} for the curried `flatMap` function, the definition of $\diamond_{\text{Opt}}$ is simplified to

$$f^{:A\to\mathbb{1}+B} \diamond_{\text{Opt}} g^{:B\to\mathbb{1}+C} \stackrel{\text{def}}{=} f \,⨾\, \text{flm}_{\text{Opt}}(g) \quad . \tag{9.32}$$

Then we compute $f \diamond_{\text{Opt}} g$ using this definition of $\diamond_{\text{Opt}}$ as

$$\text{definition of } f: \quad \underline{f} \diamond_{\text{Opt}} g = (h \,⨾\, \text{pu}_{\text{Opt}}) \underline{\diamond_{\text{Opt}}} g = h \,⨾\, \underline{\text{pu}_{\text{Opt}} \,⨾\, \text{flm}_{\text{Opt}}(g)}$$

$$\text{compute composition (see below)}: \quad = h \,⨾\, g \quad .$$

The result of composing `pure` and `flatMap` for `Option` is not obvious, but it turns out that pu_{Opt} followed by $\text{flm}_{\text{Opt}}(g)$ is equal to just g. To verify that, let us first use the syntax of Scala:

```
pure(x) == Some(x)              // By definition of ‘pure‘ for ‘Option‘.
p.flatMap(g) == p match {       // By definition of ‘flatMap‘ for ‘Option‘.
  case None       => None
  case Some(x)    => g(x)
}
pure(x).flatMap(g) == Some(x).flatMap(g) == g(x)
```

The same symbolic computation is written in the code notation like this:

$$\text{pu}_{\text{Opt}} = \left|\begin{array}{c||cc} & \mathbb{1} & A \\ \hline A & \mathbb{0} & \text{id} \end{array}\right| \quad , \qquad \text{flm}_{\text{Opt}}(g^{:A\to\mathbb{1}+B}) = \left|\begin{array}{c||c} & \mathbb{1}+B \\ \hline \mathbb{1} & 1 \to 1+\mathbb{0}^{:B} \\ A & g \end{array}\right| \quad , \tag{9.33}$$

$$\text{pu}_{\text{Opt}} \,⨾\, \text{flm}_{\text{Opt}}(g) = \left|\begin{array}{c||cc} & \mathbb{1} & A \\ \hline A & \mathbb{0} & \text{id} \end{array}\right| \,⨾\, \left|\begin{array}{c||c} & \mathbb{1}+B \\ \hline \mathbb{1} & 1 \to 1+\mathbb{0}^{:B} \\ A & g \end{array}\right|$$

$$\text{matrix composition}: \quad = \left|\begin{array}{c||c} & \mathbb{1}+B \\ \hline A & \underline{\text{id} \,⨾\, g} \end{array}\right| = g \quad . \tag{9.34}$$

Since we have now shown that $f \diamond_{\text{Opt}} g = h \,⨾\, g$, the naturality law (9.23) follows:

$$\text{liftOpt}\,(f \diamond_{\text{Opt}} g) = \text{liftOpt}\,(h \,⨾\, g) = h^{\uparrow F} \,⨾\, \text{liftOpt}\,(g) \quad .$$

This derivation is possible because the function f always returns a non-empty `Option`, which corresponds to filtering with a predicate that always returns `true`. This reduces filtering to an identity function, which simplifies the composition law (9.31) by eliminating one of the `liftOpt` functions.

(b) We keep $f^{:A\to\mathbb{1}+B}$ arbitrary but choose $g^{:B\to\mathbb{1}+C}$ to be of the form

$$g^{:B\to\mathbb{1}+C} \stackrel{\text{def}}{=} h^{:B\to C} \,⨾\, \text{pu}^{C}_{\text{Opt}} \quad ,$$

where $h^{:B\to C}$ is an arbitrary function, and substitute into `liftOpt`'s composition law (9.31):

$$\text{liftOpt}\,(f \diamond_{\text{Opt}} g) = \text{liftOpt}\,(f) \,⨾\, \text{liftOpt}\,(h \,⨾\, \text{pu}_{\text{Opt}})$$

$$\text{use Eq. (9.29)}: \quad = \text{liftOpt}\,(f) \,⨾\, h^{\uparrow F} \quad .$$

To proceed, we need to compute the Kleisli composition $f \diamond_{\text{Opt}} g$ for the chosen form of g,

$$
\begin{aligned}
& f \diamond_{\text{Opt}} \underline{g} = f \diamond_{\text{Opt}} (h \,⨾\, \text{pu}_{\text{Opt}}) \\
\text{definition (9.32) of } \diamond_{\text{Opt}} : \quad & = f \,⨾\, \text{flm}_{\text{Opt}}(h \,⨾\, \underline{\text{pu}_{\text{Opt}}}) = f \,⨾\, \text{flm}_{\text{Opt}}(y \to \mathbb{0} + h(y)) \\
\text{definition (9.33) of } \text{flm}_{\text{Opt}} : \quad & = f \,⨾\, \begin{array}{c|c} & \mathbb{1} + C \\ \hline \mathbb{1} & 1 \to 1 + \mathbb{0}^{:C} \\ B & y \to \mathbb{0} + h(y) \end{array} \\
\text{split matrix into 2 columns} : \quad & = f \,⨾\, \begin{array}{c|cc} & \mathbb{1} & C \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ B & \mathbb{0} & h \end{array} \\
\text{definition of lifting, } {}^{\uparrow\text{Opt}} : \quad & = f \,⨾\, h^{\uparrow\text{Opt}} \quad .
\end{aligned}
$$

This yields the **right naturality** law of `liftOpt`,

$$
\begin{tikzcd}
F^A \arrow[d, "\text{liftOpt}\,(f)"'] \arrow[dr, "\text{liftOpt}\,(f \,⨾\, h^{\uparrow\text{Opt}})"] & \\
F^B \arrow[r, "h^{\uparrow F}"] & F^C
\end{tikzcd}
$$

$$
\text{liftOpt}_F\,(f^{:A\to\mathbb{1}+B}) \,⨾\, h^{\uparrow F} = \text{liftOpt}_F\,(f \,⨾\, h^{\uparrow\text{Opt}}) \quad . \tag{9.35}
$$

Assuming that `deflate` is defined via `liftOpt` by Eq. (9.21), we express the two sides of `deflate`'s naturality law (9.14) through `liftOpt`:

$$
\begin{aligned}
\text{left-hand side of Eq. (9.14)} : \quad & \text{deflate} \,⨾\, f^{\uparrow F} = \text{liftOpt}\,(\text{id}) \,⨾\, f^{\uparrow F} \\
\text{use Eq. (9.35)} : \quad & = \text{liftOpt}\,(f^{\uparrow\text{Opt}}) \quad . \\
\text{right-hand side of Eq. (9.14)} : \quad & f^{\uparrow\text{Opt}\uparrow F} \,⨾\, \text{deflate} = f^{\uparrow\text{Opt}\uparrow F} \,⨾\, \text{liftOpt}\,(\text{id}) \\
\text{naturality law (9.23)} : \quad & = \text{liftOpt}\,(\underline{f^{\uparrow\text{Opt}} \,⨾\, \text{id}}) = \text{liftOpt}\,(f^{\uparrow\text{Opt}}) \quad .
\end{aligned}
$$

We are justified to use the naturality law (9.23) because we proved in part **(a)** that it holds.

If `filter` obeys the naturality law then so does `deflate`, and vice versa (Statements 9.2.2.4–9.2.2.5).

To summarize, we have proved that if `filter` is defined via `liftOpt` and the two laws (9.29), (9.31) hold for `liftOpt` then all four laws of `filter` will hold.

The converse statement also holds: if we define `liftOpt` through `filter` by Eq. (9.25), the four laws of `filter` will imply the two laws of `liftOpt`.

Exercise 9.2.3.4 Show that any function $f^{:A\to\mathbb{1}+B}$ can be expressed as $f = \psi_p \,⨾\, h_{|p}^{\uparrow\text{Opt}}$ with a suitable choice of a predicate $p^{:A\to 2}$ and a partial function $h_{|p}^{:A\to B}$. Derive explicit formulas for h and ψ in terms of f and implement them in Scala.

Exercise 9.2.3.5* Show that the law (9.31) holds if `liftOpt` is defined via `filter` using Eq. (9.25).

Hint: first derive the two naturality laws for `liftOpt`; then use Exercise 9.2.3.4 to extend the restricted composition law (9.30) to the full law (9.31).

9.2.4 Constructions of filterable functors

How can we recognize a filterable functor F^A by its type expression, without having to prove laws? One intuition is that the type F^A must be able to accommodate replacing values of A by unit values; this replacement is performed by the function `deflate`. To make this intuition more precise, it helps to perform structural analysis that systematically looks for type constructions creating new filterable functors out of existing ones while preserving the laws.

To begin, we note that `Option[_]`, `Either[L, _]`, `Try[_]`, `Seq[_]`, and `Map[K, _]` are filterable. Let us now go through all constructions available for exponential-polynomial types. To check whether a functor is filterable, it is convenient to use the `liftOpt` function and its two laws (9.29)–(9.31).

Type parameters There are three constructions that work solely by manipulating type parameters: the identity functor $\text{Id}^A \stackrel{\text{def}}{=} A$, the constant functor $\text{Const}^{Z,A} \stackrel{\text{def}}{=} Z$ (where Z is a fixed type), and the functor composition, $F^A \stackrel{\text{def}}{=} G^{H^A}$.

The identity functor is *not* filterable because `deflate` of type $\mathbb{1} + A \to A$ cannot be implemented.

The constant functor $\text{Const}^{Z,A} \stackrel{\text{def}}{=} Z$ can be viewed as a "wrapper" that never wraps any values of type A. This functor is filterable because we can define $\text{liftOpt}\,(_) \stackrel{\text{def}}{=} \text{id}^{:Z\to Z}$ (filtering is a no-op for a "wrapper" that is always empty). All laws usually hold for an identity function. To verify the laws, note that the lifting to the Const functor is also an identity function: $f^{\uparrow\text{Const}} = \text{id}^{:Z\to Z}$ for any $f^{:A\to B}$.

$$\text{check law (9.29)}: \quad \text{liftOpt}_{\text{Const}}(f \,⨾\, \text{pu}_{\text{Opt}}) = \text{id} = f^{\uparrow\text{Const}} \quad ,$$

$$\text{check law (9.31)}: \quad \text{liftOpt}_{\text{Const}}(f) \,⨾\, \text{liftOpt}_{\text{Const}}(g) = \text{id} \,⨾\, \text{id} = \text{id} = \text{liftOpt}_{\text{Const}}(f \diamond_{\text{Opt}} g) \quad .$$

The functor composition $F^A \stackrel{\text{def}}{=} G^{H^A}$ requires only H to be a filterable functor:

Statement 9.2.4.1 The functor $F^A \stackrel{\text{def}}{=} G^{H^A}$ is filterable when H^A is filterable and G^A is any functor.

Proof Assuming that liftOpt_H is available and lawful, we define liftOpt_F as

```
def liftOpt_F[A, B](f: A => Option[B]): G[H[A]] => G[H[B]] =
  { g: G[H[A]] => g.map(liftOpt_H(f)) }
```

$$\text{liftOpt}_F(f) \stackrel{\text{def}}{=} \left(\text{liftOpt}_H(f)\right)^{\uparrow G} \quad .$$

To verify the identity-naturality law of `liftOpt_F`, note that $f^{\uparrow F} = f^{\uparrow H \uparrow G}$ by definition of F:

$$\text{expect to equal } f^{\uparrow F}: \quad \text{liftOpt}_F(f \,⨾\, \text{pu}_{\text{Opt}}) = \left(\text{liftOpt}_H(f \,⨾\, \text{pu}_{\text{Opt}})\right)^{\uparrow G}$$

$$\text{law (9.29) of liftOpt}_H: \quad = \left(f^{\uparrow H}\right)^{\uparrow G} = f^{\uparrow F} \quad .$$

To verify the composition law of liftOpt_F, compute:

$$\text{expect to equal liftOpt}_F(f \diamond_{\text{Opt}} f'): \quad \text{liftOpt}_F(f) \,⨾\, \text{liftOpt}_F(f') = \left(\text{liftOpt}_H(f) \,⨾\, \text{liftOpt}_H(f')\right)^{\uparrow G}$$

$$\text{composition law of liftOpt}_H: \quad = \left(\text{liftOpt}_H(f \diamond_{\text{Opt}} f')\right)^{\uparrow G} = \text{liftOpt}_F(f \diamond_{\text{Opt}} f') \quad .$$

Products To show that the product of two filterable functors is filterable, we will use a definition of $\text{liftOpt}_{G^\bullet \times H^\bullet}$ and a proof quite similar to what we did for the product of functors (Statement 6.2.3.3).

Statement 9.2.4.2 The functor $F^A \stackrel{\text{def}}{=} G^A \times H^A$ is filterable if $G^\bullet$ and $H^\bullet$ are filterable functors.

Proof To define liftOpt_F, we use the pair product operation $\boxtimes$ similarly to Eq. (6.13),

$$\text{liftOpt}_F(p) \stackrel{\text{def}}{=} \text{liftOpt}_G(p) \boxtimes \text{liftOpt}_H(p) \quad .$$

The lifting to F is defined by Eq. (6.13) as $f^{\uparrow F} = f^{\uparrow G} \boxtimes f^{\uparrow H}$. To verify the naturality-identity law:

$$\text{expect to equal } f^{\uparrow F}: \quad \text{liftOpt}_F(f \,⨾\, \text{pu}_{\text{Opt}}) = \text{liftOpt}_G(f \,⨾\, \text{pu}_{\text{Opt}}) \boxtimes \text{liftOpt}_H(f \,⨾\, \text{pu}_{\text{Opt}})$$

$$\text{law (9.29) for } G \text{ and } H: \quad = f^{\uparrow G} \boxtimes f^{\uparrow H} = f^{\uparrow F} \quad .$$

To verify the composition law:

$$\text{expect to equal liftOpt}_F(f \diamond_{\text{Opt}} g): \quad \text{liftOpt}_F(f) \,⨾\, \text{liftOpt}_F(g)$$

$$\text{definition of liftOpt}_F: \quad = \left(\text{liftOpt}_G(f) \boxtimes \text{liftOpt}_H(f)\right) \,⨾\, \left(\text{liftOpt}_G(g) \boxtimes \text{liftOpt}_H(g)\right)$$

$$\text{composition property (9.36)}: \quad = \left(\text{liftOpt}_G(f) \,⨾\, \text{liftOpt}_G(g)\right) \boxtimes \left(\text{liftOpt}_H(f) \,⨾\, \text{liftOpt}_H(g)\right)$$

$$\text{composition laws of } G \text{ and } H: \quad = \text{liftOpt}_G(f \diamond_{\text{Opt}} g) \boxtimes \text{liftOpt}_H(f \diamond_{\text{Opt}} g)$$

$$\text{definition of liftOpt}_F: \quad = \text{liftOpt}_F(f \diamond_{\text{Opt}} g) \quad .$$

In this calculation, we used the composition property

$$(f \boxtimes g) \,⨾\, (p \boxtimes q) = (f \,⨾\, p) \boxtimes (g \,⨾\, q) \quad , \tag{9.36}$$

which follows from the definition of the pair product operation $\boxtimes$,

$$
\begin{aligned}
& (f \boxtimes g) \fatsemi (p \boxtimes q) = \big(a \times b \to f(a) \times g(b)\big) \fatsemi \big(c \times d \to p(c) \times q(d)\big) \\
\text{compute composition}: \quad & = \big(a \times b \to p(f(a)) \times q(g(b))\big) \\
\text{definition of } \boxtimes: \quad & = \big(a \to p(f(a))\big) \boxtimes \big(b \to q(g(b))\big) \\
\text{definition of the } \fatsemi \text{ operation}: \quad & = (f \fatsemi p) \boxtimes (g \fatsemi q) \quad .
\end{aligned}
$$

Co-products There are two constructions that produce new filterable functors involving disjunctive types (co-product types). The first construction is the filterable co-product $F^A \stackrel{\text{def}}{=} G^A + H^A$, where $G^\bullet$ and $H^\bullet$ are filterable functors. This is similar to the functor co-product (Statement 6.2.3.4). The second construction is $F^A \stackrel{\text{def}}{=} \mathbb{1} + A \times G^A$ where $G^\bullet$ is a filterable functor. This cannot be reduced to the first construction because $A \times G^A$ is not filterable.

Statement 9.2.4.3 The functor $F^A \stackrel{\text{def}}{=} G^A + H^A$ is filterable if G and H are filterable functors.

Proof Assuming that liftOpt_G and liftOpt_H are available, we define liftOpt_F as

```
def liftOpt_F[A, B](f: A => Option[B]): Either[G[A], H[A]] => Either[G[B], H[B]] = {
  case Left(ga)    => liftOpt_G(f)(ga)
  case Right(ha)   => liftOpt_H(f)(ha)
}
```

$$
\text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) \stackrel{\text{def}}{=} \left|\begin{array}{c||cc} & G^A & H^A \\ \hline G^A & \text{liftOpt}_G(f) & \mathbb{0} \\ H^A & \mathbb{0} & \text{liftOpt}_H(f) \end{array}\right| \quad .
$$

Lifting to the functor F^A is defined as in Statement 6.2.3.4,

$$
f^{\uparrow F} \stackrel{\text{def}}{=} \left|\begin{array}{c||cc} & G^A & H^A \\ \hline G^A & f^{\uparrow G} & \mathbb{0} \\ H^A & \mathbb{0} & f^{\uparrow H} \end{array}\right| \quad .
$$

Our matrix calculations will always have the rows and columns of type $G^A + H^A$, so we will omit the type annotations for brevity. To verify the naturality-identity law (9.29):

$$
\begin{aligned}
\text{expect to equal } f^{\uparrow F}: \quad & \text{liftOpt}_F(f \fatsemi \text{pu}_{\text{Opt}}) = \left\|\begin{array}{cc} \text{liftOpt}_G(f \fatsemi \text{pu}_{\text{Opt}}) & \mathbb{0} \\ \mathbb{0} & \text{liftOpt}_H(f \fatsemi \text{pu}_{\text{Opt}}) \end{array}\right| \\
\text{law (9.29) for } G \text{ and } H: \quad & = \left\|\begin{array}{cc} f^{\uparrow G} & \mathbb{0} \\ \mathbb{0} & f^{\uparrow H} \end{array}\right| = f^{\uparrow F} \quad .
\end{aligned}
$$

To verify the composition law (9.31):

$$
\begin{aligned}
\text{expect to equal } \text{liftOpt}_F(f \diamond_{\text{Opt}} g): \quad & \text{liftOpt}_F(f) \fatsemi \text{liftOpt}_F(g) \\
\text{definition of } \text{liftOpt}_F: \quad & = \left\|\begin{array}{cc} \text{liftOpt}_G(f) & \mathbb{0} \\ \mathbb{0} & \text{liftOpt}_H(f) \end{array}\right| \fatsemi \left\|\begin{array}{cc} \text{liftOpt}_G(g) & \mathbb{0} \\ \mathbb{0} & \text{liftOpt}_H(g) \end{array}\right| \\
\text{matrix composition}: \quad & = \left\|\begin{array}{cc} \text{liftOpt}_G(f) \fatsemi \text{liftOpt}_G(g) & \mathbb{0} \\ \mathbb{0} & \text{liftOpt}_H(f) \fatsemi \text{liftOpt}_H(g) \end{array}\right| \\
\text{law (9.31) for } G \text{ and } H: \quad & = \left\|\begin{array}{cc} \text{liftOpt}_G(f \diamond_{\text{Opt}} g) & \mathbb{0} \\ \mathbb{0} & \text{liftOpt}_H(f \diamond_{\text{Opt}} g) \end{array}\right| \\
\text{definition of } \text{liftOpt}_F: \quad & = \text{liftOpt}_F(f \diamond_{\text{Opt}} g) \quad .
\end{aligned}
$$

Statement 9.2.4.4 The functor $F^A \stackrel{\text{def}}{=} \mathbb{1} + A \times G^A$ is filterable if G is a filterable functor.

Proof Assuming that liftOpt_G is available, we define liftOpt_F as

```
def liftOpt_F[A, B](f: A => Option[B]): Option[(A, G[A])] => Option[(B, G[B])] = {
  case None            => None                    // An empty wrapper remains empty.
  case Some((a, ga))   => f(a) match {            // Does ‘a‘ pass the filtering predicate?
     case None    => None                         // No. Drop all data, return an empty wrapper.
     case Some(b) => Some((b, liftOpt_G(f)(ga)))  // Yes. Keep ‘b‘ and filter ‘ga‘ using ‘liftOpt_G‘.
  }
}
```

$$\text{liftOpt}_F(f) \stackrel{\text{def}}{=} \left\| \begin{array}{c|c} & \mathbb{1} + B \times G^B \\ \hline \mathbb{1} & 1 \to 1 + \mathbb{0}^{:B\times G^B} \\ A \times G^A & a \times g \to a \triangleright f \triangleright \left| \begin{array}{c|cc} & \mathbb{1} & B \times G \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ B & \mathbb{0} & b \to b \times \text{liftOpt}_G(f)(g) \end{array} \right| \end{array} \right\| \quad .$$

The matrix for liftOpt_F has a non-split output column (representing the disjunctive type $\mathbb{1} + B \times G^B$). This is because the code must pattern-match on $f(a)$ in order to determine whether the result is of type $\mathbb{1}$ or of type $B \times G^B$. It is inconvenient to use such matrices in calculations since the conventions of matrix products do not work unless all parts of a disjunctive type are represented by separate columns. To be able to do symbolic calculations, we need to rewrite the code in a different way.

Note that the code pattern when destructuring an `Option` value looks like this function,

```
def p[A, B, C](q: A => Option[B], r: B => C): Option[A] => Option[C] = {
  case None    => None
  case Some(a) => q(a) match {   // Destructure the result of applying the function ‘q‘.
    case None      => None
    case Some(b)   => Some(r(b)) // Apply a final transformation ‘r‘.
  }
}
```

The code of `p` can be rewritten using the standard `flatMap` method of the `Option` type:

```
def p[A, B, C](q: A => Option[B], r: B => C): Option[A] => Option[C] = _.flatMap { a => q(a).map(r) }
```

So we may write the code of `liftOpt_F` as

```
def liftOpt_F[A, B](f: A => Option[B]): Option[(A, G[A])] => Option[(B, G[B])] = _.flatMap {
     case (a, ga)   => f(a).map { b => (b, liftOpt_G(f)(ga)) }
}
```

$$\text{liftOpt}_F(f) \stackrel{\text{def}}{=} \text{flm}_{\text{Opt}}\left(a^{:A} \times g^{:G^A} \to a \triangleright f \,\fatsemi\, (b \to b \times \text{liftOpt}_G(f)(g))^{\uparrow \text{Opt}}\right) \quad . \tag{9.37}$$

To verify the laws, we first need to define the lifting to the functor F:

```
def fmap_F[A, B](f: A => B): Option[(A, G[A])] => Option[(B, G[B])] = {
  case None            => None
  case Some((a, ga))   => Some((f(a), ga.map(f)))
}
```

Again, it is convenient to rewrite the code using the standard `map` method of the `Option` type:

```
def fmap_F[A, B](f: A => B): Option[(A, G[A])] => Option[(B, G[B])] = _.map {
  case (a, ga) => (f(a), ga.map(f))
}
```

$$f^{\uparrow F} = \left(a^{:A} \times g^{:G^A} \to f(a) \times (g \triangleright f^{\uparrow G})\right)^{\uparrow \text{Opt}} = \left(f \boxtimes f^{\uparrow G}\right)^{\uparrow \text{Opt}} \quad .$$

For brevity, we omit type annotations. The naturality-identity law (9.29) for F is verified by

$$\begin{aligned}
\text{expect to equal } f^{\uparrow F}: &\quad \text{liftOpt}_F(f \fatsemi \text{pu}_{\text{Opt}}) \\
\text{definition of liftOpt}_F: &\quad = \text{flm}_{\text{Opt}}(a \times g \to a \triangleright f \fatsemi \text{pu}_{\text{Opt}} \fatsemi (b \to b \times \underline{\text{liftOpt}_G(f \fatsemi \text{pu}_{\text{Opt}})}(g))^{\uparrow \text{Opt}}) \\
\text{law (9.29) for } G: &\quad = \text{flm}_{\text{Opt}}(a \times g \to a \triangleright f \fatsemi \underline{\text{pu}_{\text{Opt}} \fatsemi (b \to b \times f^{\uparrow G}(g))^{\uparrow \text{Opt}}}) \\
\text{naturality (8.8) of pu}_{\text{Opt}}: &\quad = \text{flm}_{\text{Opt}}(a \times g \to \underline{a \triangleright f \fatsemi (b \to b \times f^{\uparrow G}(g))} \fatsemi \text{pu}_{\text{Opt}}) \\
\text{function composition}: &\quad = \text{flm}_{\text{Opt}}(\underline{a \times g \to (f(a) \times f^{\uparrow G}(g)) \triangleright} \text{pu}_{\text{Opt}}) = \underline{\text{flm}_{\text{Opt}}}((f \boxtimes f^{\uparrow G}) \fatsemi \underline{\text{pu}_{\text{Opt}}}) \\
\text{use Eq. (9.38)}: &\quad = (f \boxtimes f^{\uparrow G})^{\uparrow \text{Opt}} = f^{\uparrow F} \quad .
\end{aligned}$$

Here we used a property that applies to a composition of any $q^{:A \to B}$ with pu_{Opt} under `flatMap`:

$$\text{flm}_{\text{Opt}}(q^{:A \to B} \fatsemi \text{pu}_{\text{Opt}}^{:B \to \mathbb{1}+B}) = q^{\uparrow \text{Opt}} \quad . \tag{9.38}$$

This property is derived by applying the code for `Option`'s `flatMap` and `map` to suitable values:

```
None.flatMap(x => Some(q(x))) == None              None.map(q) == None        // pure_Opt(x) == Some(x)
Some(x).flatMap(x => Some(q(x))) == Some(q(x))     Some(x).map(q) == Some(q(x))
```

Additional work is necessary to check `liftOpt`'s composition law,

$$\text{liftOpt}_F(f) \fatsemi \text{liftOpt}_F(f') = \text{liftOpt}_F(f \diamond_{\text{Opt}} f') \quad . \tag{9.39}$$

Since $\text{liftOpt}_F(f)$ is equal to $\text{flm}_{\text{Opt}}(...)$, so we need somehow to transform $\text{flm}_{\text{Opt}}(...) \fatsemi \text{flm}_{\text{Opt}}(...)$ into an expression of the form $\text{flm}_{\text{Opt}}(...)$. To obtain such a transformation, we use a trick: consider the composition law for the $\text{liftOpt}_{\text{Opt}}$ operation of the `Option` functor (not the liftOpt_F of the functor F). Since we already know that `Option` is filterable, the composition law must hold for $\text{liftOpt}_{\text{Opt}}$:

$$\text{liftOpt}_{\text{Opt}}(f^{:A \to \text{Opt}^B}) \fatsemi \text{liftOpt}_{\text{Opt}}(g^{:B \to \text{Opt}^C}) = \text{liftOpt}_{\text{Opt}}(f \diamond_{\text{Opt}} g) \quad .$$

To express $\text{liftOpt}_{\text{Opt}}$ through known functions, recall that `Option`'s `deflate` function, with the type signature $\text{Opt}^{\text{Opt}^A} \to \text{Opt}^A$, is equal to the standard method `flatten` (denoted by ftn_{Opt} for brevity):

$$\text{liftOpt}_{\text{Opt}}(f) = f^{\uparrow \text{Opt}} \fatsemi \text{deflate}_{\text{Opt}} = f^{\uparrow \text{Opt}} \fatsemi \text{ftn}_{\text{Opt}} = \text{flm}_{\text{Opt}}(f) \quad .$$

So, `Option`'s `flatMap` method equals `liftOpt` and obeys a law similar to `liftOpt`'s composition law,

$$\text{flm}_{\text{Opt}}(f) \fatsemi \text{flm}_{\text{Opt}}(f') = \text{flm}_{\text{Opt}}(f \diamond_{\text{Opt}} f') = \text{flm}_{\text{Opt}}(f \fatsemi \text{flm}_{\text{Opt}}(f')) \quad . \tag{9.40}$$

We call Eq. (9.40) the **associativity law** of `flatMap` for reasons explained in Statement 9.4.2.1 below.

To make the calculation quicker, denote by $r_{f,g}$ a sub-expression in Eq. (9.37), so that we can write

$$r_{f,g} \stackrel{\text{def}}{=} b \to b \times \text{liftOpt}_G(f)(g) \quad , \qquad \text{liftOpt}_F(f) = \text{flm}_{\text{Opt}}(a \times g \to a \triangleright f \fatsemi r_{f,g}^{\uparrow \text{Opt}}) \quad . \tag{9.41}$$

We can now start with the left-hand side of Eq. (9.39):

$$\begin{aligned}
& \text{liftOpt}_F(f) \fatsemi \text{liftOpt}_F(f') \\
\text{use Eq. (9.41)}: &\quad = \text{flm}_{\text{Opt}}(a \times g \to a \triangleright f \fatsemi r_{f,g}^{\uparrow \text{Opt}}) \fatsemi \text{flm}_{\text{Opt}}(a' \times g' \to a' \triangleright f' \fatsemi r_{f',g'}^{\uparrow \text{Opt}}) \\
\text{use Eq. (9.40)}: &\quad = \text{flm}_{\text{Opt}}(\underline{(}a \times g \to a \triangleright f \triangleright r_{f,g}^{\uparrow \text{Opt}}\underline{)} \fatsemi \text{flm}_{\text{Opt}}(a' \times g' \to a' \triangleright f' \fatsemi r_{f',g'}^{\uparrow \text{Opt}})) \\
\triangleright\text{-notation}: &\quad = \text{flm}_{\text{Opt}}(a \times g \to a \triangleright f \fatsemi r_{f,g}^{\uparrow \text{Opt}} \fatsemi \text{flm}_{\text{Opt}}(a' \times g' \to a' \triangleright f' \fatsemi r_{f',g'}^{\uparrow \text{Opt}})) \quad .
\end{aligned} \tag{9.42}$$

Here we used a convenient "parenthesis-canceling" property of the $\triangleright$-notation:

$$(x \to x \triangleright f) \fatsemi g = (x \to \underline{g(f(x))} = (x \to x \triangleright \underline{f \triangleright g}) = (x \to x \triangleright f \fatsemi g) \quad .$$

It is not immediately clear how to proceed, so let us transform the right-hand side of Eq. (9.39):

$$\begin{aligned} & \text{liftOpt}_F(f \diamond_{\text{Opt}} f') \\ \text{definition (9.32) of } \diamond_{\text{Opt}} : \quad & = \text{liftOpt}_F(f \fatsemi \text{flm}_{\text{Opt}}(f')) \\ \text{use Eq. (9.41)} : \quad & = \text{flm}_{\text{Opt}}\big(a \times g \to a \triangleright f \fatsemi \text{flm}_{\text{Opt}}(f') \fatsemi r^{\uparrow\text{Opt}}_{f \fatsemi \text{flm}_{\text{Opt}}(f'),g}\big) \quad . \end{aligned} \tag{9.43}$$

How can we show that the last expressions in Eqs. (9.42)–(9.43) are equal? To find a way forward, let us compare these two equations and find the sub-expressions that remain different:

$$\text{sub-expression from the left-hand side :} \quad r^{\uparrow\text{Opt}}_{f,g} \fatsemi \text{flm}_{\text{Opt}}\big(a' \times g' \to a' \triangleright f' \fatsemi r^{\uparrow\text{Opt}}_{f',g'}\big) \tag{9.44}$$

$$\text{we would like that to become equal to :} \quad \text{flm}_{\text{Opt}}(f') \fatsemi r^{\uparrow\text{Opt}}_{f \fatsemi \text{flm}_{\text{Opt}}(f'),g} \quad . \tag{9.45}$$

The difference is in the lifted functions to the left and to the right of flm_{Opt}. If we somehow bring those functions inside $\text{flm}_{\text{Opt}}(...)$, we may be able to simplify them further. So, we look for properties of `Option`'s `flatMap` that should have the form

$$\begin{aligned} (p^{:A\to B})^{\uparrow\text{Opt}} \fatsemi \text{flm}_{\text{Opt}}(q^{:B\to\text{Opt}^C}) &= \text{flm}_{\text{Opt}}(???^{:A\to\text{Opt}^C}) \quad , \\ \text{flm}_{\text{Opt}}(p^{:A\to\text{Opt}^B}) \fatsemi (q^{:B\to C})^{\uparrow\text{Opt}} &= \text{flm}_{\text{Opt}}(???^{:A\to\text{Opt}^C}) \quad . \end{aligned}$$

The typed holes must be filled using the only available data (the functions p and q):

$$(p^{:A\to B})^{\uparrow\text{Opt}} \fatsemi \text{flm}_{\text{Opt}}(q^{:B\to\text{Opt}^C}) = \text{flm}_{\text{Opt}}(p \fatsemi q) \quad , \tag{9.46}$$

$$\text{flm}_{\text{Opt}}(p^{:A\to\text{Opt}^B}) \fatsemi (q^{:B\to C})^{\uparrow\text{Opt}} = \text{flm}_{\text{Opt}}(p \fatsemi q^{\uparrow\text{Opt}}) \quad . \tag{9.47}$$

We omit the proofs for these **naturality laws** of `flatMap`. With them, we transform Eqs. (9.44)–(9.45):

$$\begin{aligned} & \underline{r^{\uparrow\text{Opt}}_{f,g} \fatsemi} \text{flm}_{\text{Opt}}\big(a' \times g' \to a' \triangleright f' \fatsemi r^{\uparrow\text{Opt}}_{f',g'}\big) \\ \text{use Eq. (9.46)} : \quad & = \text{flm}_{\text{Opt}}\big(\underline{r_{f,g}} \fatsemi (a' \times g' \to a' \triangleright f' \fatsemi r^{\uparrow\text{Opt}}_{f',g'})\big) \\ \text{expand } r_{f,g} : \quad & = \text{flm}_{\text{Opt}}\big((a \to a \times \text{liftOpt}_G(f)(g)) \fatsemi (a' \times g' \to a' \triangleright f' \fatsemi r^{\uparrow\text{Opt}}_{f',g'})\big) \\ \text{compute composition} : \quad & = \text{flm}_{\text{Opt}}\big(a \to a \triangleright f' \fatsemi r^{\uparrow\text{Opt}}_{f',\text{liftOpt}_G(f)(g)}\big) \quad , \end{aligned}$$

and

$$\begin{aligned} & \text{flm}_{\text{Opt}}(f') \fatsemi \underline{r^{\uparrow\text{Opt}}_{f \fatsemi \text{flm}_{\text{Opt}}(f'),g}} = \text{flm}_{\text{Opt}}\big(\underline{f'} \fatsemi r^{\uparrow\text{Opt}}_{f \fatsemi \text{flm}_{\text{Opt}}(f'),g}\big) \\ \text{expand function } f' : \quad & = \text{flm}_{\text{Opt}}\big(a \to a \triangleright f' \fatsemi r^{\uparrow\text{Opt}}_{f \fatsemi \text{flm}_{\text{Opt}}(f'),g}\big) \quad . \end{aligned}$$

The difference between sub-expressions has become smaller: it just remains to show that

$$r_{f',\text{liftOpt}_G(f)(g)} \stackrel{?}{=} r_{f \fatsemi \text{flm}_{\text{Opt}}(f'),g} \quad .$$

Expand the definition of $r_{f,g}$ in both sides:

$$b \to b \times \underline{\text{liftOpt}_G(f')\big(\text{liftOpt}_G(f)(g)\big)} \stackrel{?}{=} b \to b \times \underline{\text{liftOpt}_G\big(f \fatsemi \text{flm}_{\text{Opt}}(f')\big)(g)} \quad .$$

Omitting the common sub-expressions, we find the remaining difference:

$$\text{liftOpt}_G(f')(\text{liftOpt}_G(f)(g)) \stackrel{?}{=} \text{liftOpt}_G(f \,⨾\, \text{flm}_{\text{Opt}}(f'))(g) \quad .$$

This is equivalent to liftOpt_G's composition law applied to the function g,

$$g \triangleright \text{liftOpt}_G(f) ⨾ \text{liftOpt}_G(f') = g \triangleright \text{liftOpt}_G(\underline{f \diamond_{\text{Opt}} f'}) = g \triangleright \text{liftOpt}_G(f \,⨾\, \text{flm}_{\text{Opt}}(f')) \quad .$$

Since the composition law of liftOpt_G is assumed to hold, we have finished the proof of Eq. (9.39).

The construction in Statement 9.2.4.4 implements a special kind of filtering where the value $a^{:A}$ in the pair of type $A \times G^A$ needs to pass the filter for any data to remain in the functor after filtering. We can use the same construction repeatedly with $G^\bullet \stackrel{\text{def}}{=} \mathbb{1}$ and obtain the type

$$L_n^A \stackrel{\text{def}}{=} \underbrace{\mathbb{1} + A \times (\mathbb{1} + A \times (\mathbb{1} + \ldots \times (\mathbb{1} + A \times \mathbb{1})))}_{\text{parameter } A \text{ is used } n \text{ times}} \quad ,$$

which is equivalent to a list of up to n elements. The construction defines a filtering operation for $L_n^\bullet$ that will delete any data beyond the first value of type A that does fails the predicate. It is clear that this filtering operation implements the standard `takeWhile` method defined on sequences. So, `takeWhile` is a lawful filtering operation (see Example 9.1.4.3 where it was used).

We can also generalize the construction of Statement 9.2.4.4 to the functor

$$F^A \stackrel{\text{def}}{=} \mathbb{1} + \underbrace{A \times A \times \ldots \times A}_{n \text{ times}} \times G^A \quad .$$

We implement the filtering operation with the requirement that *all* n values of type A in the tuple $A \times A \times \ldots \times A \times G^A$ must pass the filtering predicate, or else F^A becomes empty. Example 9.1.4.2 shows how such filtering operations may be used in practice.

Function types As we have seen in Chapter 6 (Statement 6.2.3.5), functors involving a function type, such as $F^A \stackrel{\text{def}}{=} G^A \to H^A$, require $G^\bullet$ to be a *contrafunctor* rather than a functor. It turns out that the functor $G^A \to H^A$ is filterable only if the contrafunctor $G^\bullet$ has certain properties (Eqs. (9.50)–(9.51) below) similar to properties of filterable functors. We will call such contrafunctors **filterable**.

To motivate the definition of filterable contrafunctors, consider the operation `liftOpt` for F:

$$\text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) : (G^A \to H^A) \to G^B \to H^B \quad , \qquad \text{liftOpt}_F(f) = p^{:G^A\to H^A} \to g^{:G^B} \to ???^{:H^B} \quad .$$

Assume that H is filterable, so that we have the function $\text{liftOpt}_H(f) : H^A \to H^B$. We will fill the typed hole $???^{:H^B}$ if we somehow get a value of type H^A; that is only possible if we apply $p^{:G^A\to H^A}$,

$$\text{liftOpt}_F(f) = p^{:G^A\to H^A} \to g^{:G^B} \to \text{liftOpt}_H(f)(p(???^{:G^A})) \quad .$$

The only way to proceed is to have a function $G^B \to G^A$. We cannot obtain such a function by lifting f to the contrafunctor G: that gives $f^{\downarrow G} : G^{\mathbb{1}+B} \to G^A$. So, we need to require having a function

$$\text{liftOpt}_G(f^{:A\to\mathbb{1}+B}) : G^B \to G^A \quad . \tag{9.48}$$

This function is analogous to `liftOpt` for functors, except for the reverse direction of transformation ($G^B \to G^A$ instead of $G^A \to G^B$). We can now complete the implementation of liftOpt_F:

$$\begin{aligned}
& \text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) \stackrel{\text{def}}{=} p^{:G^A\to H^A} \to g^{:G^B} \to \underline{\text{liftOpt}_H(f)(p(\text{liftOpt}_G(f)(g)))} \\
\triangleright\text{-notation}: \quad & = p^{:G^A\to H^A} \to \underline{g^{:G^B} \to g \triangleright} \text{liftOpt}_G(f) \triangleright p \triangleright \text{liftOpt}_H(f) \\
\text{omit } (g \to g \triangleright): \quad & = p \to \text{liftOpt}_G(f) ⨾ p ⨾ \text{liftOpt}_H(f) \quad .
\end{aligned} \tag{9.49}$$

Note that the last line is similar to Eq. (6.15) but with `liftOpt` instead of `map`:

$$(f^{:A\to B})^{\uparrow F} = p^{:G^A\to H^A} \to f^{\downarrow G} ⨾ p ⨾ f^{\uparrow F} = p \to \text{cmap}_G(f) ⨾ p ⨾ \text{fmap}_F(f) \quad .$$

The laws for filterable contrafunctors are chosen such that $F^A \stackrel{\text{def}}{=} G^A \to H^A$ can be shown to obey filtering laws when $H^\bullet$ is a filterable functor and $G^\bullet$ is a filterable contrafunctor.

Statement 9.2.4.5 Assume that $H^\bullet$ is a lawful filterable functor and $G^\bullet$ is a contrafunctor with a function liftOpt_G having type signature (9.48) and obeying the laws (9.50)–(9.51) shown below. Then the functor $F^A \stackrel{\text{def}}{=} G^A \to H^A$ is filterable.

Proof We will arrive at the required laws for G by trying to prove the laws for F.

Because of the function type of $F^\bullet$, it is convenient for derivations to apply both sides of the laws to an arbitrary value $p^{:G^A \to H^A}$. Consider the naturality-identity law of F:

$$\begin{aligned}
\text{expect to equal } p \triangleright f^{\uparrow F} = f^{\downarrow G} \fatsemi p \fatsemi f^{\uparrow H} : \quad & p \triangleright \text{liftOpt}_F(f \fatsemi \text{pu}_{\text{Opt}}) \\
\text{definition (9.49) of liftOpt}_F : \quad & = \text{liftOpt}_G(f \fatsemi \text{pu}_{\text{Opt}}) \fatsemi p \fatsemi \underline{\text{liftOpt}_H(f \fatsemi \text{pu}_{\text{Opt}})} \\
\text{naturality-identity law of liftOpt}_H : \quad & = \text{liftOpt}_G(f \fatsemi \text{pu}_{\text{Opt}}) \fatsemi p \fatsemi \underline{f^{\uparrow H}} \quad .
\end{aligned}$$

The only sub-expression that remains different is $\text{liftOpt}_G(f \fatsemi \text{pu}_{\text{Opt}})$, and the derivation will be finished if we assume the **naturality-identity law** of filterable contrafunctors to be

$$\text{liftOpt}_G(f \fatsemi \text{pu}_{\text{Opt}}) = f^{\downarrow G} \quad . \tag{9.50}$$

The composition law of F applied to a value to a value $p^{:G^A \to H^A}$ is

$$\begin{aligned}
\text{left-hand side of Eq. (9.31) for } F : \quad & p \triangleright \text{liftOpt}_F(f) \fatsemi \text{liftOpt}_F(g) = \underline{p \triangleright \text{liftOpt}_F(f)} \triangleright \text{liftOpt}_F(g) \\
\text{definition (9.49) of liftOpt}_F : \quad & = \left(\text{liftOpt}_G(f) \fatsemi p \fatsemi \text{liftOpt}_H(f)\right) \underline{\triangleright \text{liftOpt}_F(g)} \\
\text{definition (9.49)} : \quad & = \text{liftOpt}_G(g) \fatsemi \left(\text{liftOpt}_G(f) \fatsemi p \fatsemi \underline{\text{liftOpt}_H(f)}\right) \fatsemi \text{liftOpt}_H(g) \\
\text{composition law (9.31) of liftOpt}_H : \quad & = \text{liftOpt}_G(g) \fatsemi \text{liftOpt}_G(f) \fatsemi p \fatsemi \text{liftOpt}_H(f \diamond_{\text{Opt}} g) \quad .
\end{aligned}$$

The right-hand side of Eq. (9.31) for F is

$$p \triangleright \text{liftOpt}_F(f \diamond_{\text{Opt}} g) = \text{liftOpt}_G(f \diamond_{\text{Opt}} g) \fatsemi p \fatsemi \text{liftOpt}_H(f \diamond_{\text{Opt}} g) \quad .$$

Clearly, we need to require the **composition law** of filterable contrafunctor G to be

$$\text{liftOpt}_G(g) \fatsemi \text{liftOpt}_G(f) = \text{liftOpt}_G(f \diamond_{\text{Opt}} g) \quad . \tag{9.51}$$

Assuming that liftOpt_G satisfies this law, we obtain Eq. (9.31) for F and so conclude the proof.

Recursive types How to generalize the filtering operation from sequences to other recursive types? For motivation, we look at two examples: the filterable `List` functor defined by

$$\text{List}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times \text{List}^A \quad ,$$

and the recursive construction for ordinary functors (Statement 6.2.3.7) that requires a bifunctor $S^{\bullet,\bullet}$.

Statement 9.2.4.6 If $G^\bullet$ is a filterable functor, the recursive functor $F^\bullet$ defined by

$$F^A \stackrel{\text{def}}{=} G^A + A \times F^A$$

is filterable. With $G^A \stackrel{\text{def}}{=} \mathbb{1}$, this construction reproduces the standard filtering operation of `List`.

Proof We first need to implement the type constructor F and the function liftOpt_F:

```
sealed trait F[A]                                // Assume that the functor G was defined previously.
final case class FG[A](g: G[A]) extends F[A]
final case class FAF[A](a: A, rf: F[A]) extends F[A]
                                              // Assume that liftOpt_G is available and define liftOpt_F:
def liftOpt_F[A, B](f: A => Option[B]): F[A] => F[B] = {
   case FG(g)       => FG(liftOpt_G(f)(g))
   case FAF(a, rf)  => f(a) match {                      // Does ‘a‘ pass the filtering predicate?
      case None        => liftOpt_F(f)(rf)               // No. Drop ‘a‘ and filter ‘rf‘ recursively.
      case Some(b)     => FAF[B](b, liftOpt_F(f)(rf))  // Yes. Keep ‘b‘ and filter ‘rf‘ recursively.
   }
}
```

$$\text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) \stackrel{\text{def}}{=} \left|\begin{array}{c||c|}
 & F^B \\ \hline
G^A & g^{:G^A} \to \text{liftOpt}_G(f)(g) + \mathbb{0}^{:B\times F^B} \\
A\times F^A & a^{:A}\times r^{:F^A} \to f(a) \triangleright \left|\begin{array}{c||c|}
 & F^B \\ \hline
\mathbb{1} & 1 \to \overline{\text{liftOpt}_F}(f)(r) \\
B & b^{:B} \to \mathbb{0}^{:G^B} + b\times\overline{\text{liftOpt}_F}(f)(r)
\end{array}\right|
\end{array}\right| \quad .$$

The overline denotes recursive uses of liftOpt_F within its definition.

With this definition of liftOpt_F, it is inconvenient to use matrix composition because the matrix shown above has a single column instead of columns split by the disjunctive type $F^A = G^A + A\times F^A$. We have seen a similar problem in the proof of Statement 9.2.4.4, where we rewrote the code via flm_{Opt} and avoided using matrices with non-split columns. But it is not clear how to rewrite the code for liftOpt_F in that way. Instead, we use a more straightforward approach: apply both sides of the laws to an arbitrary value of type F^A.

The disjunctive type F^A has two cases, $G^A + \mathbb{0}^{:A\times F^A}$ and $\mathbb{0}^{:G^A} + A\times F^A$. When applied to a value $g^{:G^A} + \mathbb{0}$, the function liftOpt_F is exactly the same as liftOpt_G, so both laws are satisfied since $G^\bullet$ is a lawful filterable functor. It remains to verify the laws when applied to a value $\mathbb{0}^{:G^A} + a^{:A}\times r^{:F^A}$.

To prepare for the calculations, write the result of applying liftOpt_F to a value $\mathbb{0} + a\times r$:

$$(\mathbb{0}^{:G^A} + a^{:A}\times r^{:F^A}) \triangleright \text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) = f(a) \triangleright \left|\begin{array}{c||c|}
 & F^B \\ \hline
\mathbb{1} & 1 \to r \triangleright \overline{\text{liftOpt}_F}(f) \\
B & b^{:B} \to \mathbb{0}^{:G^B} + (b\times r) \triangleright \overline{\text{liftOpt}_F}(f)
\end{array}\right| \quad . \tag{9.52}$$

To check the naturality-identity law (9.29) of liftOpt_F, begin with the left-hand side:

$$(\mathbb{0} + a\times r) \triangleright \text{liftOpt}_F(\underline{f \fatsemi \text{pu}_{\text{Opt}}}) = (\mathbb{0} + a\times r) \triangleright \text{liftOpt}_F(x \to \mathbb{0} + f(x))$$

use Eq. (9.52) : $$= \underline{a \triangleright (f \fatsemi \text{pu}_{\text{Opt}})} \fatsemi \left|\begin{array}{c||c|}
 & \\ \hline
1 & 1 \to r \triangleright \overline{\text{liftOpt}_F}(f \fatsemi \text{pu}_{\text{Opt}})(r) \\
b^{:B} & \to \mathbb{0} + (b\times r) \triangleright \overline{\text{liftOpt}_F}(f \fatsemi \text{pu}_{\text{Opt}})
\end{array}\right|$$

evaluate function of a : $$= \big(\mathbb{0} + f(a)\big) \triangleright \left|\begin{array}{c||c|}
 & \\ \hline
1 & 1 \to r \triangleright \overline{\text{liftOpt}_F}(f \fatsemi \text{pu}_{\text{Opt}}) \\
b^{:B} & \to \mathbb{0} + (b\times r) \triangleright \overline{\text{liftOpt}_F}(f \fatsemi \text{pu}_{\text{Opt}})
\end{array}\right|$$

substitute into matrix : $$= \mathbb{0} + (f(a)\times r) \triangleright \underline{\overline{\text{liftOpt}_F}(f \fatsemi \text{pu}_{\text{Opt}})}$$

inductive assumption : $$= \mathbb{0} + (f(a)\times r) \triangleright \underline{f^{\uparrow F}} \quad .$$

The same expression is found by applying the right-hand side of Eq. (9.29) to $\mathbb{0} + a\times r$:

$$(\mathbb{0}^{:G^A} + a^{:A}\times r^{:F^A}) \triangleright (f^{:A\to B})^{\uparrow F} = \mathbb{0}^{:G^A} + f(a)\times f^{\uparrow F}(r) = \mathbb{0} + f(a)\times r \triangleright f^{\uparrow F} \quad .$$

To verify the composition law (9.31), apply its left-hand side to $\mathbb{0} + a\times r$:

$$(\mathbb{0} + a\times r) \triangleright \text{liftOpt}_F(f) \fatsemi \text{liftOpt}_F(g)$$

use Eq. (9.52) : $$= f(a) \triangleright \left|\begin{array}{c||c|}
 & \\ \hline
1 & 1 \to r \triangleright \overline{\text{liftOpt}_F}(f) \\
b^{:B} & \to \mathbb{0} + (b\times r) \triangleright \overline{\text{liftOpt}_F}(f)
\end{array}\right| \triangleright \underline{\text{liftOpt}_F(g)}$$

$$\text{apply } \text{liftOpt}_F(g): \quad = f(a) \triangleright \left\| \begin{array}{c} 1 \to r \triangleright \overline{\text{liftOpt}_F}(f) \triangleright \text{liftOpt}_F(g) \\ b^{:B} \to (\mathbb{0} + (b \times r) \triangleright \overline{\text{liftOpt}_F}(f)) \triangleright \text{liftOpt}_F(g) \end{array} \right|$$

$$\text{use Eq. (9.52)}: \quad = f(a) \triangleright \left\| \begin{array}{c} 1 \to r \triangleright \overline{\text{liftOpt}_F}(f) \fatsemi \text{liftOpt}_F(g) \\ b^{:B} \to g(b) \triangleright \left\| \begin{array}{c} 1 \to r \triangleright \overline{\text{liftOpt}_F}(f) \fatsemi \overline{\text{liftOpt}_F}(g) \\ c^{:C} \to \mathbb{0} + (c \times r) \triangleright \overline{\text{liftOpt}_F}(f) \fatsemi \overline{\text{liftOpt}_F}(g) \end{array} \right| \end{array} \right| \tag{9.53}$$

$$\text{inductive assumption}: \quad = a \triangleright f \fatsemi \left\| \begin{array}{c} 1 \to r \triangleright \overline{\text{liftOpt}_F}(f \diamond_{\text{Opt}} g) \\ b^{:B} \to g(b) \triangleright \left\| \begin{array}{c} 1 \to r \triangleright \overline{\text{liftOpt}_F}(f \diamond_{\text{Opt}} g) \\ c^{:C} \to \mathbb{0} + (c \times r) \triangleright \overline{\text{liftOpt}_F}(f \diamond_{\text{Opt}} g) \end{array} \right| \end{array} \right| . \tag{9.54}$$

We are justified in using the inductive assumption on the composition $\overline{\text{liftOpt}_F}(f) \fatsemi \text{liftOpt}_F(g)$ even though $\text{liftOpt}_F(g)$ is not a recursive call. It is sufficient that at least one term in the function composition is a recursive call to $\overline{\text{liftOpt}_F}$.

The right-hand side of Eq. (9.31) applied to $\mathbb{0} + a \times r$ gives

$$\begin{aligned} & (\mathbb{0} + a \times r) \triangleright \text{liftOpt}_F(f \diamond_{\text{Opt}} g) \\ \text{use Eq. (9.52)}: \quad & = a \triangleright (f \diamond_{\text{Opt}} g) \triangleright \left\| \begin{array}{c} 1 \to r \triangleright \overline{\text{liftOpt}_F}(f \diamond_{\text{Opt}} g) \\ c^{:C} \to \mathbb{0} + (c \times r) \triangleright \overline{\text{liftOpt}_F}(f \diamond_{\text{Opt}} g) \end{array} \right| . \end{aligned}$$

The only remaining difference compared with the last expression in Eq. (9.54) is

$$f \fatsemi \left\| \begin{array}{c} p \\ b^{:B} \to g(b) \triangleright \left\| \begin{array}{c} p \\ q \end{array} \right| \end{array} \right| \stackrel{?}{=} (f \diamond_{\text{Opt}} g) \fatsemi \left\| \begin{array}{c} p \\ q \end{array} \right| , \tag{9.55}$$

where $p \stackrel{\text{def}}{=} 1 \to r \triangleright \text{liftOpt}_F(f \diamond_{\text{Opt}} g)$ and $q \stackrel{\text{def}}{=} c \to \mathbb{0} + c \times r \triangleright \text{liftOpt}_F(f \diamond_{\text{Opt}} g)$ are some fixed functions. We can show that Eq. (9.55) holds for arbitrary functions p and q. Start from the right-hand side:

$$\text{expect the l.h.s. of Eq. (9.55)}: \quad \underline{(f \diamond_{\text{Opt}} g)} \fatsemi \left\| \begin{array}{c} p \\ q \end{array} \right| = f \fatsemi \underline{\text{flm}_{\text{Opt}}(g)} \fatsemi \left\| \begin{array}{c} p \\ q \end{array} \right| = f \fatsemi \left\| \begin{array}{c} 1 \to 1 + \mathbb{0} \\ b \to g(b) \end{array} \right| \fatsemi \left\| \begin{array}{c} p \\ q \end{array} \right|$$

$$\text{apply } \left\| \begin{array}{c} p \\ q \end{array} \right| : \quad = f \fatsemi \left\| \begin{array}{c} 1 \to \underline{(1 + \mathbb{0})} \triangleright \left\| \begin{array}{c} p \\ q \end{array} \right| \\ b \to g(b) \triangleright \left\| \begin{array}{c} p \\ q \end{array} \right| \end{array} \right| = f \fatsemi \left\| \begin{array}{c} 1 \to p \\ b \to g(b) \triangleright \left\| \begin{array}{c} p \\ q \end{array} \right| \end{array} \right| .$$

This proves Eq. (9.55) and concludes the proof of Statement 9.2.4.6.

This implementation preserves all values $x^{:A}$ except those that fail the filtering predicate (i.e., when $f(x) = 1 + \mathbb{0}$). So, it has the same filtering logic as the standard `filter` method for sequences. □

The next construction is for a functor defined via a filterable recursion scheme. The filtering logic is then different from that used in Statement 9.2.4.6.

Statement 9.2.4.7 If $S^{A,R}$ is a bifunctor that is filterable with respect to A, the recursive functor F^A defined by the recursion scheme $S^{A,R}$ (type equation $F^A \stackrel{\text{def}}{=} S^{A,F^A}$) is filterable.

Proof We follow the proof of Statement 6.2.3.7. Implement liftOpt_F recursively as

$$\text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) \stackrel{\text{def}}{=} \text{liftOpt}_S(f) \fatsemi \text{bimap}_S(\text{id})(\overline{\text{liftOpt}_F}(f)) = \text{liftOpt}_S(f) \fatsemi (\overline{\text{liftOpt}_F}(f))^{\uparrow S^{B,\bullet}} \quad .$$

where liftOpt_S is assumed to obey the laws. The lifting to F is defined (also recursively) by

$$(f^{:A\to B})^{\uparrow F} \stackrel{\text{def}}{=} \text{bimap}_S(f)(\overline{f^{\uparrow F}}) = f^{\uparrow S^{\bullet,R}} \fatsemi (\overline{f^{\uparrow F}})^{\uparrow S^{B,\bullet}} \quad .$$

To verify the naturality-identity law (9.29), compute:

$$\begin{aligned}
\text{expect to equal } f^{\uparrow F}: \quad & \text{liftOpt}_F(f \fatsemi \text{pu}_{\text{Opt}}) = \underline{\text{liftOpt}_S(f \fatsemi \text{pu}_{\text{Opt}})} \fatsemi (\overline{\text{liftOpt}_F}(f \fatsemi \text{pu}_{\text{Opt}}))^{\uparrow S^{B,\bullet}} \\
\text{law (9.29) of liftOpt}_S: \quad & = f^{\uparrow S^{\bullet,R}} \fatsemi \underline{(\overline{\text{liftOpt}_F}(f \fatsemi \text{pu}_{\text{Opt}}))^{\uparrow S^{B,\bullet}}} \\
\text{inductive assumption}: \quad & = f^{\uparrow S^{\bullet,R}} \fatsemi (\overline{f^{\uparrow F}})^{\uparrow S^{B,\bullet}} = f^{\uparrow F} \quad .
\end{aligned}$$

To verify the composition law (9.31), compute:

$$\begin{aligned}
& \text{liftOpt}_F(f) \fatsemi \text{liftOpt}_F(g) \\
& = \text{liftOpt}_S(f) \fatsemi \underline{(\overline{\text{liftOpt}_F}(f))^{\uparrow S^{B,\bullet}} \fatsemi \text{liftOpt}_S(g)} \fatsemi (\overline{\text{liftOpt}_F}(g))^{\uparrow S^{C,\bullet}} \\
\text{naturality of liftOpt}_S: \quad & = \underline{\text{liftOpt}_S(f) \fatsemi \text{liftOpt}_S(g)} \fatsemi (\overline{\text{liftOpt}_F}(f))^{\uparrow S^{C,\bullet}} \underline{\fatsemi} (\overline{\text{liftOpt}_F}(g))^{\uparrow S^{C,\bullet}} \\
\text{law (9.31) of liftOpt}_S: \quad & = \text{liftOpt}_S(f \diamond_{\text{Opt}} g) \fatsemi \underline{(\overline{\text{liftOpt}_F}(f) \fatsemi \overline{\text{liftOpt}_F}(g))^{\uparrow S^{C,\bullet}}} \\
\text{inductive assumption}: \quad & = \text{liftOpt}_S(f \diamond_{\text{Opt}} g) \fatsemi \overline{\text{liftOpt}_F}(f \diamond_{\text{Opt}} g) = \text{liftOpt}_F(f \diamond_{\text{Opt}} g) \quad .
\end{aligned}$$

We have assumed a **naturality law** of liftOpt_S with respect to the type parameter R of $S^{A,R}$:

$$(p^{:R\to R'})^{\uparrow S^{B,\bullet}} \fatsemi \text{liftOpt}_S(g^{:B\to\mathbb{1}+C}) = \text{liftOpt}_S(g^{:B\to\mathbb{1}+C}) \fatsemi (p^{:R\to R'})^{\uparrow S^{C,\bullet}} \quad .$$

$$\begin{tikzcd}
S^{B,R} \arrow[r, "\text{liftOpt}_S(g^{:B\to\mathbb{1}+C})"] \arrow[d, "(p^{:R\to R'})^{\uparrow S^{B,\bullet}}"'] & S^{C,R} \arrow[d, "(p^{:R\to R'})^{\uparrow S^{C,\bullet}}"] \\
S^{B,R'} \arrow[r, "\text{liftOpt}_S(g^{:B\to\mathbb{1}+C})"] & S^{C,R'}
\end{tikzcd}$$

We expect this naturality law to hold automatically for any fully parametric implementation of liftOpt_S (see Appendix D).

9.2.5 Filterable contrafunctors: motivation and examples

An intuitive view is that functors are "wrappers" of data, while contrafunctors "consume" data. Filterable functors permit us to exclude certain data from a wrapper; filterable contrafunctors permit us to exclude certain data from being consumed. Let us now make this intuition precise.

A simple contrafunctor is $C^A \stackrel{\text{def}}{=} A \to Z$, where Z is a constant type. This is a general form of an "extractor" — for example, a function that extracts logging information of type Z from data of an arbitrary type A. It is sometimes necessary to exclude particular kinds of information (e.g., private personal data) from logging. We can implement this by providing a predicate of type `A => Boolean` that decides, depending on the given value $x^{:A}$, whether x should be passed to the extractor. That predicate will be attached to a given extractor $c^{:C^A}$ by the `filter` operation:

```
def filter[A](p: A => Boolean): C[A] => C[A]
val extractor: C[Payload] = ???            // Original code for extracting metadata from payloads.
val noPrivateData: Payload => Boolean = ???  // Returns true only if payload has no private data.
val filtered: C[Payload] = filter(noPrivateData)(extractor)     // Will not extract private data.
```

How could filtering work when the predicate returns `false`? Even if the data (of type A) is excluded from the filtered extractor, the function of type $A \to Z$ must still return a value of type Z. A solution is to have a *default value* of type Z that the extractor will return when data is excluded from it.

There are two simple ways of implementing filtering for $C^A = A \to Z$ via a default value of type Z: First, by storing that default value in the type C^A and considering the contrafunctor $C^A \stackrel{\text{def}}{=} Z \times (A \to Z)$ instead of $A \to Z$. Second, by using a type $\mathbb{1} + Z$ instead of Z, since the type $\mathbb{1} + Z$ has a default value $1 + \mathbb{0}^{:Z}$. The following two examples show how this works.

Example 9.2.5.1 Implement `filter` for the contrafunctor $C^A \stackrel{\text{def}}{=} A \to \mathbb{1} + Z$, where Z is a fixed type.

Solution This contrafunctor "consumes" data of type A and computes values of type $\mathbb{1} + Z$. Given a value c of type C^A, which is a function that we might write like this:

```
val c: A => Option[Z] = ???
```

we need somehow to impose a filter predicate $p^{:A\to 2}$ ensuring that the function c is applied only to values that pass the predicate. The result will be a new function $d^{:A\to\mathbb{1}+Z}$ that will use its argument only if it passes the predicate. The function d could return `None` for all arguments, but that implementation would lose information. If `p(a) == true`, we may compute `c(a)`, getting a value of type $\mathbb{1} + Z$ that d should return. If `p(a) == false`, the function d must return `None`. So, the code of d must be

```
val d: A => Option[Z] = { a => if (p(a)) c(a) else None }
```

The transformation from `c` to `d` is a filtering operation for the contrafunctor C^A, implemented as

```
def filter[A](p: A => Boolean)(c: A => Option[Z]): A => Option[Z] = { a =>
  if (p(a)) c(a) else None
}                        // Equivalent code is { a => Some(a).filter(p).flatMap(c) }
```

$$\text{filt}_C(p^{:A\to 2}) \stackrel{\text{def}}{=} c^{:A\to\mathbb{1}+Z} \to \text{pu}_{\text{Opt}} \,\text{⨾}\, \text{filt}_{\text{Opt}}(p) \,\text{⨾}\, \text{flm}_{\text{Opt}}(c) = c \to \psi_p \,\text{⨾}\, \text{flm}_{\text{Opt}}(c) \quad .$$

Example 9.2.5.2 Implement `filter` for the contrafunctor $C^A \stackrel{\text{def}}{=} Z \times (A \to Z)$.

Solution The code for the `filter` function is

```
def filter[A](p: A => Boolean): ((Z, A => Z)) => (Z, A => Z) = {
  case (z, f) => (z, a => if (p(a)) f(a) else z)
}
```

$$\text{filt}_C(p^{:A\to 2}) \stackrel{\text{def}}{=} z^{:Z} \times f^{:A\to Z} \to z \times \left(a^{:A} \to p(a) \triangleright \begin{array}{|c||c|} & Z \\ \hline \mathbb{1}\,(\text{false}) & z \\ \mathbb{1}\,(\text{true}) & f(a) \end{array} \right) \quad .$$

Note that the type $Z \times (A \to Z)$ is equivalent to $\mathbb{1} + A \to Z$:

$$Z \times (A \to Z) \cong (\mathbb{1} \to Z) \times (A \to Z) \cong \mathbb{1} + A \to Z \quad .$$

For the contrafunctor $C^A \stackrel{\text{def}}{=} \mathbb{1} + A \to Z$, the `filter` function is implemented by

```
def filter[A](p: A => Boolean)(c: Option[A] => Z): Option[A] => Z = {
  case Some(a) if p(a)   => c(Some(a))          // Only apply ‘c‘ to ‘a‘ if ‘p(a) == true‘.
  case _                 => c(None)       // Return c(None) otherwise, or for empty Option.
}                                   // Equivalent code is _.filter(p).pipe(c) (Scala 2.13).
```

$$\begin{aligned} \text{filt}_C(p^{:A\to 2}) &\stackrel{\text{def}}{=} c^{:\mathbb{1}+A\to Z} \to \underline{x^{:\mathbb{1}+A} \to x} \triangleright \text{filt}_{\text{Opt}}(p) \triangleright c \\ &= c^{:\mathbb{1}+A\to Z} \to \text{filt}_{\text{Opt}}(p) \,\text{⨾}\, c \quad . \end{aligned}$$

Another motivation for filterable contrafunctors comes from the construction $F^A \stackrel{\text{def}}{=} G^A \to H^A$ (Statement 9.2.4.5): In order to assure the properties of a filterable functor for $F^\bullet$, the contrafunctor $G^\bullet$ must have the `liftOpt` function as shown in Eq. (9.48). The existence of the `liftOpt` function for a contrafunctor turns out to be equivalent to the existence of the `filter` function, as long as suitable

laws hold. To verify that equivalence, begin by defining the function `inflate`, whose role is similar to that of `deflate` for filterable functors. The type signature of `inflate` is

$$\text{inflate}_C : C^A \to C^{\mathbb{1}+A} \quad .$$

We can relate `inflate` to `filter` and `liftOpt` by the following equations (to be derived below):

$$\text{filt}_C(p) = \text{inflate}_C \,;\, (\psi_p^{:A\to\mathbb{1}+A})^{\downarrow C} \quad , \qquad \text{inflate}_C = \left(\text{get}^{:\mathbb{1}+A\to A}\right)^{\downarrow C} \,;\, \text{filt}_C(\text{nonEmpty}) \quad , \tag{9.56}$$

$$\text{inflate}_C = \text{liftOpt}_C(\text{id}^{:\mathbb{1}+A\to\mathbb{1}+A}) \quad , \qquad \text{liftOpt}_C(f^{:A\to\mathbb{1}+B}) = \text{inflate}_C \,;\, f^{\downarrow C} \quad . \tag{9.57}$$

$$\begin{tikzcd}
C^{\mathbb{1}+A} \arrow[dr, "\text{filt}_C^{\mathbb{1}+A}(\text{nonEmpty}^{:\mathbb{1}+A\to 2})"'] & C^A \arrow[l, "\text{get}^{\downarrow C}"'] \arrow[r, "\text{filt}_C^A(p^{:A\to 2})"] \arrow[d, "\text{inflate}_C"] & C^A \\
& C^{\mathbb{1}+A} \arrow[ur, "\psi_p^{\downarrow C}"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
C^B \arrow[r, "\text{inflate}_C"] \arrow[dr, "\text{liftOpt}_C(f)^{\stackrel{\text{def}}{=}}"'] & C^{\mathbb{1}+B} \arrow[d, "(f^{:A\to\mathbb{1}+B})^{\downarrow C}"] \\
& C^A
\end{tikzcd}$$

These functions have different but equivalent laws: filt_C has 4 laws, inflate_C has 3, and liftOpt_C has just 2 laws. So, liftOpt_C is the most convenient function for proving laws, while inflate_C is the easiest to implement in code (and to check whether a given contrafunctor is filterable). The laws of liftOpt_C are similar to the laws of `liftOpt` for filterable functors (we omit the derivations):

$$\begin{aligned}
\text{naturality law of liftOpt}_C : &\quad \text{liftOpt}_C(f^{:A\to B} \,;\, g^{:B\to\mathbb{1}+E}) = \text{liftOpt}_C(g) \,;\, f^{\downarrow C} \quad , \\
\text{naturality-identity law of liftOpt}_C : &\quad \text{liftOpt}_C(f^{:A\to B} \,;\, \text{pu}_{\text{Opt}}^{:B\to\mathbb{1}+B}) = f^{\downarrow C} \quad , \\
\text{composition law of liftOpt}_C : &\quad \text{liftOpt}_C(g^{:B\to\mathbb{1}+E}) \,;\, \text{liftOpt}_C(f^{:A\to\mathbb{1}+B}) = \text{liftOpt}_C(f \diamond_{\text{Opt}} g) \quad .
\end{aligned}$$

As an illustration, let us implement these functions for the contrafunctor $C^A \stackrel{\text{def}}{=} A \to \mathbb{1} + Z$.

Example 9.2.5.3 Implement `inflate` and `liftOpt` for $C^A \stackrel{\text{def}}{=} A \to \mathbb{1} + Z$, and verify Eq. (9.56).

Solution We implement the type signatures of `inflate` and `liftOpt`, preserving information:

```
def inflate[A](c: A => Option[Z]): Option[A] => Option[Z] = _.flatMap(c)
def liftOpt[A, B](f: A => Option[B])(c: B => Option[Z]): A => Option[Z] = { a => f(a).flatMap(c) }
```

$$\text{inflate}_C \stackrel{\text{def}}{=} c^{:A\to\mathbb{1}+Z} \to \text{flm}_{\text{Opt}}(c) \quad , \qquad \text{liftOpt}_C(f^{:A\to\mathbb{1}+B}) \stackrel{\text{def}}{=} c^{:B\to\mathbb{1}+Z} \to f \,;\, \text{flm}_{\text{Opt}}(c) = c \to f \diamond_{\text{Opt}} c \quad .$$

To verify Eq. (9.56), we need the code for lifting a function $g^{:A\to B}$ to the contrafunctor C:

$$(g^{:A\to B})^{\downarrow C} = c^{:B\to\mathbb{1}+Z} \to \underline{a^{:A} \to c(g(a))} = c^{:B\to\mathbb{1}+Z} \to \underline{g \,;\, c} \quad .$$

Now we use the code for filt_C from Example 9.2.5.1 and compute $\text{get}^{\downarrow C} \,;\, \text{filt}_C(\text{nonEmpty})$ as

$$\begin{aligned}
\text{expect to equal inflate}_C : &\quad \text{get}^{\downarrow C} \,;\, \text{filt}_C(\text{nonEmpty}) \\
\text{definitions of } {}^{\downarrow C} \text{ and filt}_C : &\quad = (c \to \text{get} \,;\, c) \,;\, (c \to \psi_{\text{nonEmpty}} \,;\, \text{flm}_{\text{Opt}}(c)) \\
\text{compute composition} : &\quad = c \to \psi_{\text{nonEmpty}} \,;\, \text{flm}_{\text{Opt}}(\text{get} \,;\, c) \\
\text{use naturality law (9.46) of flm}_{\text{Opt}} : &\quad = c \to \underline{\psi_{\text{nonEmpty}} \,;\, \text{get}^{\uparrow\text{Opt}}} \,;\, \text{flm}_{\text{Opt}}(c) \\
\text{use Eq. (9.17) to simplify} : &\quad = c \to \text{flm}_{\text{Opt}}(c) = \text{inflate}_C \quad .
\end{aligned}$$

In the formula (9.56), the lifted partial function $\text{get}^{\downarrow C}$ is applied *before* the filtering operation (rather than after filtering, as would be the case for filterable functors). The derivation shows how the partial function `get` is moved around due to the reverse order of contrafunctor-lifted function composition, until we find the expression $\psi_{\text{nonEmpty}} \,;\, \text{get}^{\uparrow\text{Opt}}$ where the partial function `get` is applied *after* a filter ψ (which makes it a total function due to `filter`'s partial function law). For this reason, it is correct to apply lifted partial functions before filtering in contrafunctors.

Example 9.2.5.4 Verify Eq. (9.56) for an arbitrary filterable contrafunctor $C^\bullet$, assuming needed laws.

Solution We need to check the two directions of the isomorphism in Eq. (9.56).

(a) Starting from a given filt_C, we compute inflate_C and then use that to define a new filt'_C; we must show that $\text{filt}_C = \text{filt}'_C$.

$$\text{expect to equal } \text{filt}_C(p): \quad \text{filt}'_C(p) = \text{inflate}_C \,⨾\, \psi_p^{\downarrow C} = \text{get}^{\downarrow C} \,⨾\, \text{filt}_C(\text{nonEmpty}) \,⨾\, \psi_p^{\downarrow C} \quad .$$

The computation gets stuck here: We could simplify the composition $\psi_p \,⨾\, \text{get}$ using Eq. (9.12), if only we could move these functions next to each other. It is clear that we need a law that exchanges the order of compositions of filt_C with lifted functions. Typically, that is done by naturality laws. By analogy with Eq. (9.3) and making sure types match, we write a **naturality law** of filt_C as

$$\text{filt}_C(p^{:A\to 2}) \,⨾\, (f^{:B\to A})^{\downarrow C} = f^{\downarrow C} \,⨾\, \text{filt}_C(f \,⨾\, p) \quad . \tag{9.58}$$

Assuming this law, we find that $\text{filt}'_C(p) = \text{filt}_C(p)$:

$$\begin{aligned}
\text{use naturality law of } \text{filt}_C: \quad & \text{get}^{\downarrow C} \,⨾\, \underline{\text{filt}_C(\text{nonEmpty}) \,⨾\, \psi_p^{\downarrow C}} = \underline{\text{get}^{\downarrow C} \,⨾\, \psi_p^{\downarrow C}} \,⨾\, \text{filt}_C(\underline{\psi_p \,⨾\, \text{nonEmpty}}) \\
\text{use Eqs. (9.11)–(9.12)}: \quad & = (\underline{\psi_p \,⨾\, \text{get}})^{\downarrow C} \,⨾\, \text{filt}_C(p) = \underline{\text{id}_{|p}^{\downarrow C}} \,⨾\, \text{filt}_C(p) \\
\text{partial function law of } \text{filt}_C: \quad & = \text{id}^{\downarrow C} \,⨾\, \text{filt}_C(p) = \text{filt}_C(p) \quad .
\end{aligned}$$

Here we assumed the partial function law in the form similar to that for functors,

$$f_{|p}^{\downarrow C} \,⨾\, \text{filt}_C(p) = f^{\downarrow C} \,⨾\, \text{filt}_C(p) \quad .$$

The lifted partial function $f_{|p}^{\downarrow C}$ is applied *before* filtering, as appropriate for filterable contrafunctors.

(b) Starting from a given inflate_C, we compute filt_C and then use that to define a new $\text{inflate}'_C$; we must show that $\text{inflate}_C = \text{inflate}'_C$.

$$\text{expect to equal } \text{inflate}_C: \quad \text{inflate}'_C = \text{get}^{\downarrow C} \,⨾\, \text{filt}_C(\text{nonEmpty}) = \text{get}^{\downarrow C} \,⨾\, \text{inflate}_C \,⨾\, \psi_{\text{nonEmpty}}^{\downarrow C} .$$

The calculation cannot proceed unless we can exchange lifted functions around `inflate`. By analogy with Eq. (9.14) and making changes suitable for contrafunctors, we obtain the **naturality law** for the inflate_C function:

$$\begin{tikzcd}
C^A \arrow[r, "\text{inflate}_C"] \arrow[d, "(f^{:B\to A})^{\downarrow C}"'] & C^{\mathbb{1}+A} \arrow[d, "(f^{:B\to A})^{\uparrow\text{Opt}\downarrow C}"] \\
C^B \arrow[r, "\text{inflate}_C"] & C^{\mathbb{1}+B}
\end{tikzcd}$$

$$(f^{:B\to A})^{\downarrow C} \,⨾\, \text{inflate}_C = \text{inflate}_C \,⨾\, f^{\uparrow\text{Opt}\downarrow C} \quad . \tag{9.59}$$

With help of this law, we can finish the derivation:

$$\begin{aligned}
\text{use the naturality law}: \quad & \underline{\text{get}^{\downarrow C} \,⨾\, \text{inflate}_C} \,⨾\, \psi_{\text{nonEmpty}}^{\downarrow C} = \text{inflate}_C \,⨾\, \underline{\text{get}^{\uparrow\text{Opt}\downarrow C} \,⨾\, \psi_{\text{nonEmpty}}^{\downarrow C}} \\
\text{composition lifted to } C: \quad & = \text{inflate}_C \,⨾\, (\underline{\psi_{\text{nonEmpty}} \,⨾\, \text{get}^{\uparrow\text{Opt}}})^{\downarrow C} \\
\text{use Eq. (9.17)}: \quad & = \text{inflate}_C \,⨾\, \underline{\text{id}^{\downarrow C}} = \text{inflate}_C \quad .
\end{aligned}$$

Exercise 9.2.5.5 Implement `inflate` and `liftOpt` for the filterable contrafunctor $C^A \stackrel{\text{def}}{=} \mathbb{1} + A \to Z$.

Exercise 9.2.5.6* Verify Eq. (9.57) for an arbitrary filterable contrafunctor $C^\bullet$, assuming needed laws.

9.2.6 Constructions of filterable contrafunctors

How to build up a filterable contrafunctor from parts? Structural analysis produces a number of type constructions guaranteed to create lawful filterable contrafunctors.

The `Filterable` typeclass is inductive (see Section 8.5.4) if formulated via `filter` or via `liftOpt`, because those methods return the type C^A itself. So, we expect that the product, the exponential, and the recursive constructions will apply to filterable contrafunctors (as they do to filterable functors).

Type parameters Constant contrafunctors $C^A \stackrel{\text{def}}{=} Z$ are "trivially" filterable: all methods are identity functions, so all laws hold.

Further constructions that work with type parameters are functor compositions. The composition $P \circ Q \stackrel{\text{def}}{=} P^{Q^\bullet}$ is a contrafunctor when P is a functor and Q is a contrafunctor, or vice versa. The contrafunctor $P^{Q^\bullet}$ is filterable if $Q^\bullet$ (whether it is a functor or a contrafunctor) is filterable:

Statement 9.2.6.1 **(a)** If $P^\bullet$ is any contrafunctor and $Q^\bullet$ is a filterable functor then $P^{Q^\bullet}$ is filterable.

(b) If $P^\bullet$ is any functor and $Q^\bullet$ is a filterable contrafunctor then $P^{Q^\bullet}$ is filterable.

Proof We follow the proof of Statement 9.2.4.1 with necessary modifications.

(a) We define the `liftOpt` operation for $P \circ Q$ as $\text{liftOpt}_{P\circ Q}(f^{:A\to\mathbb{1}+B}) \stackrel{\text{def}}{=} (\text{liftOpt}_Q(f))^{\downarrow P}$.

```
def liftOpt_PQ[A, B](f: A => Option[B]): P[Q[B]] => P[Q[A]] = _.contramap(liftOpt_Q(f))
```

To verify the naturality-identity law (9.50):

$$\begin{aligned}
\text{expect to equal } f^{\downarrow(P\circ Q)}: \quad & \text{liftOpt}_{P\circ Q}(f \,⨾\, \text{pu}_{\text{Opt}}) = \underline{(\text{liftOpt}_Q(f \,⨾\, \text{pu}_{\text{Opt}}))}^{\downarrow P} \\
\text{naturality-identity law (9.29) of } Q: \quad & = f^{\uparrow Q \downarrow P} = f^{\downarrow(P\circ Q)} \quad .
\end{aligned}$$

To verify the composition law (9.51), we show that its left-hand side equals $\text{liftOpt}_{P\circ Q}(f \diamond_{\text{Opt}} g)$:

$$\begin{aligned}
\text{definition of liftOpt}_{P\circ Q}: \quad & \text{liftOpt}_{P\circ Q}(g) \,⨾\, \text{liftOpt}_{P\circ Q}(f) = (\text{liftOpt}_Q(g))^{\downarrow P} \,⨾\, (\text{liftOpt}_Q(f))^{\downarrow P} \\
\text{composition law of } P: \quad & = (\text{liftOpt}_Q(f) \,⨾\, \text{liftOpt}_Q(g))^{\downarrow P} \\
\text{composition law (9.31) of } Q: \quad & = (\text{liftOpt}_Q(f \diamond_{\text{Opt}} g))^{\downarrow P} = \text{liftOpt}_{P\circ Q}(f \diamond_{\text{Opt}} g) \quad .
\end{aligned}$$

(b) We define the `liftOpt` operation for $P \circ Q$ as $\text{liftOpt}_{P\circ Q}(f^{:A\to\mathbb{1}+B}) \stackrel{\text{def}}{=} (\text{liftOpt}_Q(f))^{\uparrow P}$.

```
def liftOpt_PQ[A, B](f: A => Option[B]): P[Q[B]] => P[Q[A]] = _.map(liftOpt_Q(f))
```

To verify the naturality-identity law (9.50):

$$\begin{aligned}
\text{expect to equal } f^{\downarrow(P\circ Q)}: \quad & \text{liftOpt}_{P\circ Q}(f \,⨾\, \text{pu}_{\text{Opt}}) = \underline{(\text{liftOpt}_Q(f \,⨾\, \text{pu}_{\text{Opt}}))}^{\uparrow P} \\
\text{naturality-identity law (9.29) of } Q: \quad & = f^{\downarrow Q \uparrow P} = f^{\downarrow(P\circ Q)} \quad .
\end{aligned}$$

To verify the composition law (9.51), we show that its left-hand side equals $\text{liftOpt}_{P\circ Q}(f \diamond_{\text{Opt}} g)$:

$$\begin{aligned}
\text{definition of liftOpt}_{P\circ Q}: \quad & \text{liftOpt}_{P\circ Q}(g) \,⨾\, \text{liftOpt}_{P\circ Q}(f) = (\text{liftOpt}_Q(g))^{\uparrow P} \,⨾\, (\text{liftOpt}_Q(f))^{\uparrow P} \\
\text{composition law of } P: \quad & = (\text{liftOpt}_Q(g) \,⨾\, \text{liftOpt}_Q(f))^{\uparrow P} \\
\text{composition law (9.51) of } Q: \quad & = (\text{liftOpt}_Q(f \diamond_{\text{Opt}} g))^{\uparrow P} = \text{liftOpt}_{P\circ Q}(f \diamond_{\text{Opt}} g) \quad .
\end{aligned}$$

Composition of two filterable contrafunctors is a filterable *functor* (Exercise 9.3.2.1).

Products and co-products If $G^\bullet$ and $H^\bullet$ are filterable contrafunctors, the product $G^A \times H^A$ and the co-product $G^A + H^A$ will also be filterable contrafunctors. Proofs are analogous to the case of filterable functors and are delegated to Exercise 9.3.2.11.

Functions We have a construction similar to that of Statement 9.2.4.5 for filterable functors:

Statement 9.2.6.2 The contrafunctor $F^A \stackrel{\text{def}}{=} G^A \to H^A$ is filterable for any filterable functor G^A and any filterable contrafunctor H^A.

Proof We define the `liftOpt` operation for $F^\bullet$ by

```
def liftOpt_F[A, B](f: A => Option[B])(p: G[B] => H[B]): G[A] => H[A] =
  { ga => liftOpt_H(f)(p(liftOpt_G(f)(ga))) }
```

To obtain a clearer code formula, rewrite the Scala code using the ▷-notation and then simplify:

$$\text{liftOpt}_F(f) \stackrel{\text{def}}{=} p^{:G^B\to H^B} \to \underline{g^{:G^A} \to g} \triangleright \text{liftOpt}_G(f) \triangleright p \triangleright \text{liftOpt}_H(f)$$

$$\text{simplify } (x \to x \triangleright y) = y: \quad = p^{:G^B\to H^B} \to \text{liftOpt}_G(f) ⨾ p ⨾ \text{liftOpt}_H(f) \quad .$$

To verify the naturality-identity law (9.50), apply both its sides to an arbitrary $p^{:G^B\to H^B}$:

$$\text{expect to equal } p \triangleright f^{\downarrow F}: \quad p \triangleright \text{liftOpt}_F(f ⨾ \text{pu}_{\text{Opt}}) = \text{liftOpt}_G(f ⨾ \text{pu}_{\text{Opt}}) ⨾ p ⨾ \text{liftOpt}_H(f ⨾ \text{pu}_{\text{Opt}})$$

$$\text{use laws (9.50) and (9.29)}: \quad = f^{\uparrow G} ⨾ p ⨾ f^{\downarrow H} = p \triangleright f^{\downarrow F} \quad .$$

To verify the composition law (9.51), show that its left-hand side equals $p \triangleright \text{liftOpt}_F(f \diamond_{\text{Opt}} g)$:

$$\triangleright\text{-notation}: \quad p \triangleright \text{liftOpt}_F(g) \underline{⨾} \text{liftOpt}_F(f) = \left(p \triangleright \text{liftOpt}_F(g)\right) \triangleright \underline{\text{liftOpt}_F(f)}$$

$$\text{definition of liftOpt}_F(f): \quad = \text{liftOpt}_G(f) ⨾ \underline{\left(p \triangleright \text{liftOpt}_F(g)\right)} ⨾ \text{liftOpt}_H(f)$$

$$\text{definition of liftOpt}_F(g): \quad = \underline{\text{liftOpt}_G(f) ⨾ \text{liftOpt}_G(g)} ⨾ p ⨾ \underline{\text{liftOpt}_H(g) ⨾ \text{liftOpt}_H(f)}$$

$$\text{composition laws of } G \text{ and } H: \quad = \text{liftOpt}_G(f \diamond_{\text{Opt}} g) ⨾ p ⨾ \text{liftOpt}_H(f \diamond_{\text{Opt}} g)$$

$$\text{definition of liftOpt}_F(f \diamond_{\text{Opt}} g): \quad = p \triangleright \text{liftOpt}_F(f \diamond_{\text{Opt}} g) \quad .$$

Example 9.2.6.3 (the search functor) A non-trivial application of Statement 9.2.6.2 is the **search functor** $S^\bullet_Z$ defined by $S^A_Z \stackrel{\text{def}}{=} (A \to \mathbb{1} + Z) \to \mathbb{1} + A$, where Z is a fixed type. This functor is filterable because it is a function from the filterable contrafunctor $A \to \mathbb{1} + Z$ to the filterable functor $\mathbb{1} + A$. The simplest case of the search functor is found by setting $Z \stackrel{\text{def}}{=} \mathbb{1}$, which gives the type constructor

$$S^A_{\mathbb{1}} \stackrel{\text{def}}{=} (A \to \mathbb{2}) \to \mathbb{1} + A \quad .$$

Values of type $S^A_{\mathbb{1}}$ may be viewed as "searchers" taking a predicate $q^{:A\to\mathbb{2}}$ and looking for a value of type A that satisfies the predicate. A searcher will return either a suitable value of type $\mathbb{0} + A$, or an empty value $1 + \mathbb{0}$ ("not found"). Applying a filter with an a predicate $p^{:A\to\mathbb{2}}$ to a searcher will exclude values of type A from the search unless they satisfy p.

Another function-type construction is a generalization of the filterable contrafunctor $A \to \mathbb{1} + Z$.

Statement 9.2.6.4 If a contrafunctor $H^\bullet$ is filterable, so is the contrafunctor $F^A \stackrel{\text{def}}{=} A \to \mathbb{1} + H^A$.

Proof We extend the implementation of `liftOpt` from Example 9.2.5.3:

```
def liftOpt_F[A, B](f: A => Option[B])(c: B => Option[H[B]]): A => Option[H[A]] =
  { a => f(a).flatMap(c).map(liftOpt_H(f)) }
```

$$\text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) \stackrel{\text{def}}{=} p^{:B\to\mathbb{1}+H^B} \to \underline{a^{:A} \to a} \triangleright f \triangleright \text{flm}_{\text{Opt}}(p) \triangleright \left(\text{liftOpt}_H(f)\right)^{\uparrow\text{Opt}}$$

$$= p \to f ⨾ \text{flm}_{\text{Opt}}(p) ⨾ \left(\text{liftOpt}_H(f)\right)^{\uparrow\text{Opt}} \quad .$$

To verify the naturality-identity law (9.50), apply both sides to an arbitrary $p^{:B\to\mathbb{1}+H^B}$:

$$\text{expect to equal } p \triangleright f^{\downarrow F}: \quad p \triangleright \text{liftOpt}_F(f ⨾ \text{pu}_{\text{Opt}}) = f ⨾ \underline{\text{pu}_{\text{Opt}} ⨾ \text{flm}_{\text{Opt}}(p)} ⨾ \left(\text{liftOpt}_H(f ⨾ \text{pu}_{\text{Opt}})\right)^{\uparrow\text{Opt}}$$

$$\text{use Eq. (9.34)}: \quad = f ⨾ p ⨾ \underline{\left(\text{liftOpt}_H(f ⨾ \text{pu}_{\text{Opt}})\right)}^{\uparrow\text{Opt}}$$

$$\text{law (9.50) of } H: \quad = f ⨾ p ⨾ \left(f^{\downarrow H}\right)^{\uparrow\text{Opt}} = p \triangleright f^{\downarrow F} \quad .$$

To verify the composition law (9.51), we apply its right-hand side to an arbitrary $p^{:C\to\mathbb{1}+H^C}$ and

transform the result until it is equal to $p \triangleright \text{liftOpt}_F(f \diamond_{\text{Opt}} g)$:

$$\begin{aligned}
\triangleright\text{-notation}: \quad & p \triangleright \text{liftOpt}_F(g) \fatsemi \text{liftOpt}_F(f) = (p \triangleright \text{liftOpt}_F(g)) \triangleright \text{liftOpt}_F(f) \\
\text{definition of liftOpt}_F(f): \quad & = f \fatsemi \text{flm}_{\text{Opt}}(p \triangleright \text{liftOpt}_F(g)) \fatsemi (\text{liftOpt}_H(f))^{\uparrow\text{Opt}} \\
\text{definition of liftOpt}_F(g): \quad & = f \fatsemi \text{flm}_{\text{Opt}}(g \fatsemi \text{flm}_{\text{Opt}}(p) \fatsemi (\text{liftOpt}_H(g))^{\uparrow\text{Opt}}) \fatsemi (\text{liftOpt}_H(f))^{\uparrow\text{Opt}} \\
\text{Eqs. (9.40) and (9.47)}: \quad & f \fatsemi \text{flm}_{\text{Opt}}(g) \fatsemi \text{flm}_{\text{Opt}}(p) \fatsemi (\text{liftOpt}_H(g) \fatsemi \text{liftOpt}_H(f))^{\uparrow\text{Opt}} \\
\text{definition (9.32) of } f \diamond_{\text{Opt}} g: \quad & = (f \diamond_{\text{Opt}} g) \fatsemi \text{flm}_{\text{Opt}}(p) \fatsemi (\text{liftOpt}_H(f \diamond_{\text{Opt}} g))^{\uparrow\text{Opt}} \\
\text{definition of liftOpt}_F: \quad & = p \triangleright \text{liftOpt}_F(f \diamond_{\text{Opt}} g) \quad .
\end{aligned}$$

Recursive types To define a contrafunctor via type recursion, we need to use a recursion scheme $S^{A,R}$ that is contravariant in A and covariant in R. In other words, the recursion scheme $S^{A,R}$ must be a profunctor (see Section 6.4.1 for the definition of a profunctor).

Statement 9.2.6.5 If $S^{A,R}$ is contravariant in A and covariant in R, and additionally $S^{\bullet,R}$ is filterable (with the type parameter R fixed), then the recursive contrafunctor $F^A \stackrel{\text{def}}{=} S^{A,F^A}$ is filterable.

Proof The recursive contrafunctor $F^\bullet$ is implemented by wrapping S in a case class:

```
type S[A, R] = ...
final case class F[A](s: S[A, F[A]])
```

The code of the function `liftOpt` for $F^\bullet$ is recursive and uses the `xmap` method of the profunctor S.

```
def liftOpt_F[A, B](f: A => Option[B]): F[B] => F[A] = { case F(sbfb) => F(
  liftOpt_S(f)(sbfb).xmap_S(identity)(liftOpt_F(f))
)}
```

$$\begin{tikzcd}
S^{B,F^B} \arrow[r, "\text{liftOpt}_S(f^{:A\to\mathbb{1}+B})"] \arrow[dr, "\text{liftOpt}_F(f) \stackrel{\text{def}}{=}"'] & S^{A,F^B} \arrow[d, "(\overline{\text{liftOpt}_F}(f))^{\uparrow S^{A,\bullet}}"] \\
& S^{A,F^A}
\end{tikzcd}$$

Note that $F^B \cong S^{B,F^B}$. As before, we use an overline to mark recursive calls to the same function:

$$\begin{aligned}
\text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) & \stackrel{\text{def}}{=} \text{liftOpt}_S(f) \fatsemi (\overline{\text{liftOpt}_F}(f))^{\uparrow S^{A,\bullet}} \\
& = \text{liftOpt}_S(f) \fatsemi \text{xmap}_S(\text{id})(\overline{\text{liftOpt}_F}(f)) \quad .
\end{aligned}$$

To verify the laws, we need the code for lifting to the contrafunctor F:

```
def cmap_F[A, B](f: A => B): F[B] => F[A] = { case F(sbfb) => F( sbfb.xmap_S(f)(cmap_F(f)) ) }
```

$$f^{\downarrow F} \stackrel{\text{def}}{=} \text{xmap}_S(f)(\overline{f^{\downarrow F}}) = f^{\downarrow S^{\bullet,F^B}} \fatsemi (\overline{f^{\downarrow F}})^{\uparrow S^{A,\bullet}} \quad .$$

The naturality-identity law (9.50) is verified by

$$\begin{aligned}
\text{expect to equal } f^{\downarrow F}: \quad & \text{liftOpt}_F(f \fatsemi \text{pu}_{\text{Opt}}) = \text{liftOpt}_S(f \fatsemi \text{pu}_{\text{Opt}}) \fatsemi (\overline{\text{liftOpt}_F}(f \fatsemi \text{pu}_{\text{Opt}}))^{\uparrow S^{A,\bullet}} \\
\text{law (9.50) for } S^{\bullet,B}: \quad & = f^{\downarrow S^{\bullet,B}} \fatsemi (\overline{\text{liftOpt}_F}(f \fatsemi \text{pu}_{\text{Opt}}))^{\uparrow S^{A,\bullet}} \\
\text{inductive assumption}: \quad & = f^{\downarrow S^{\bullet,B}} \fatsemi (\overline{f^{\downarrow F}})^{\uparrow S^{A,\bullet}} = f^{\downarrow F} \quad .
\end{aligned}$$

To verify the composition law (9.51):

$$\begin{aligned}
\text{expect liftOpt}_F(f \diamond_{\text{Opt}} g): \quad & \text{liftOpt}_F(g^{:B\to\mathbb{1}+C}) \fatsemi \text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) \\
\text{definition of liftOpt}_F: \quad & = \text{liftOpt}_S(g) \fatsemi (\overline{\text{liftOpt}_F}(g))^{\uparrow S^{B,\bullet}} \fatsemi \text{liftOpt}_S(f) \fatsemi (\overline{\text{liftOpt}_F}(f))^{\uparrow S^{A,\bullet}} \\
\text{law (9.60) of liftOpt}_S: \quad & = \text{liftOpt}_S(g) \fatsemi \text{liftOpt}_S(f) \fatsemi (\overline{\text{liftOpt}_F}(g))^{\uparrow S^{A,\bullet}} \fatsemi (\overline{\text{liftOpt}_F}(f))^{\uparrow S^{A,\bullet}} \\
\text{law (9.51) of liftOpt}_S: \quad & = \text{liftOpt}_S(f \diamond_{\text{Opt}} g) \fatsemi (\overline{\text{liftOpt}_F}(g) \fatsemi \overline{\text{liftOpt}_F}(f))^{\uparrow S^{A,\bullet}} \\
\text{inductive assumption}: \quad & = \text{liftOpt}_S(f \diamond_{\text{Opt}} g) \fatsemi (\overline{\text{liftOpt}_F}(f \diamond_{\text{Opt}} g))^{\uparrow S^{A,\bullet}} = \text{liftOpt}_F(f \diamond_{\text{Opt}} g) \quad .
\end{aligned}$$

In this derivation, we have used the naturality law of liftOpt_S with respect to lifting in the type parameter A of $S^{A,R}$:

$$\begin{tikzcd} S^{B,R} \arrow[r, "\text{liftOpt}_S(f^{:A\to\mathbb{1}+B})"] \arrow[d, "(h^{:R\to R'})^{\uparrow S^{B,\bullet}}"] & S^{A,R} \arrow[d, "h^{\uparrow S^{A,\bullet}}"'] \\ S^{B,R'} \arrow[r, "\text{liftOpt}_S(f)"] & S^{A,R'} \end{tikzcd}$$

$$\text{liftOpt}_S(f^{:A\to\mathbb{1}+B}) \fatsemi (h^{:R\to R'})^{\uparrow S^{A,\bullet}} = h^{\uparrow S^{B,\bullet}} \fatsemi \text{liftOpt}_S(f) \quad . \tag{9.60}$$

We expect this naturality law to hold for fully parametric functions, as discussed in Section 6.2.2.

9.3 Summary

What can we do with the techniques of this chapter?

- Use functor blocks to manipulate data wrapped in filterable functors.
- Decide whether a given filtering behavior satisfies the laws of `filter`.
- Decide whether a given type constructor (functor or contrafunctor) is filterable. If so, implement a `filter` or `liftOpt` function that satisfies the appropriate laws.
- Use constructions to derive the code of `filter` without trial and error.

What cannot be done with these techniques?

- Given a filterable type constructor, generate the code for `filter` or `liftOpt` automatically.

This is because most non-trivial type constructors have many lawful but *inequivalent* implementations of `filter`. When we say "a type constructor F is filterable" we mean that there is at least one lawful implementation. It is not possible to choose a "preferred" implementation automatically, since different applications may need different filtering behavior. While in most cases the standard library provides a generally useful implementations of filtering (e.g., the `filter` or `takeWhile` methods on sequences), in some situations the programmer will need to provide a custom implementation of `filter` for a custom data type. The programmer must examine the given requirements and decide whether those requirements can be implemented as a lawful `filter` function.

9.3.1 Solved examples

Example 9.3.1.1 Show that the functor $F^A \stackrel{\text{def}}{=} G^A \to A$ is not filterable (for any contrafunctor G^A).

Solution Try to implement `deflate` : $F^{\mathbb{1}+A} \to F^A$, writing out its full type signature:

$$\text{deflate}_F : (G^{\mathbb{1}+A} \to \mathbb{1}+A) \to G^A \to A \quad , \qquad \text{deflate}_F = p^{:G^{\mathbb{1}+A}\to\mathbb{1}+A} \to g^{:G^A} \to ???^{:A} \quad .$$

We cannot extract a value of type A from $g^{:G^A}$ since the contrafunctor G^A does not wrap any values of A. So, the only hope of filling the typed hole $???^{:A}$ is to apply the function p to an argument of type $G^{\mathbb{1}+A}$. Even if we are able to map $G^A \to G^{\mathbb{1}+A}$ (e.g., if $G^\bullet$ is filterable), the result of applying p will be a value of type $\mathbb{1}+A$. We cannot compute a value of type A out of that. So, the type signature of `deflate` for $F^\bullet$ is not implementable. We conclude that $F^\bullet$ is not filterable.

Example 9.3.1.2 Use known filterable constructions to show that

$$F^A \stackrel{\text{def}}{=} \text{Int}\times\text{String} \to \mathbb{1} + \text{Int}\times A + A\times(\mathbb{1}+A) + (\text{Int}\to\mathbb{1}+A+A\times A\times\text{String})$$

is a filterable functor. (Using the constructions avoids the need for proofs.)

Solution We need to analyze the structure of the functor $F^\bullet$ to decide which constructions we may use. Define some auxiliary functors that represents sub-expressions in F^A,

$$R_1^A \stackrel{\text{def}}{=} \text{Int} \times \text{String} \to A \quad , \qquad R_2^A \stackrel{\text{def}}{=} \text{Int} \to A \quad ,$$
$$G^A \stackrel{\text{def}}{=} \mathbb{1} + \text{Int} \times A + A \times (\mathbb{1} + A) \quad , \qquad H^A \stackrel{\text{def}}{=} \mathbb{1} + A + A \times A \times \text{String} \quad .$$

Now we can rewrite the type F^A as

$$F^A = R_1^{L^A} \quad , \qquad L^A \stackrel{\text{def}}{=} G^A + R_2^{H^A} \quad .$$

The type of $G^\bullet$ is a co-product, so we need to check which of the two co-product constructions (Statements 9.2.4.3 or 9.2.4.4) might apply. The first of them does not apply because the functors $\text{Int} \times A$ and $A \times (\mathbb{1} + A)$ are not filterable. But the second construction applies if we write G^A in the form $G^A = \mathbb{1} + A \times K^A$ where $K^A \stackrel{\text{def}}{=} \mathbb{1} + \text{Int} + A$. Since K^A is the co-product of the `Option` functor and a constant functor (the fixed type `Int`), $K^\bullet$ is filterable by Statement 9.2.4.3. So, G^A is filterable.

Similarly, we find that H^A is filterable by Statement 9.2.4.4 if we write $H^A = \mathbb{1} + A \times (\mathbb{1} + A \times \text{String})$, where the functor $\mathbb{1} + A \times \text{String}$ is filterable by the same construction.

The functor $R_2^{H^\bullet}$ is filterable since it is a functor composition (Statement 9.2.4.1) and $H^\bullet$ is filterable. The co-product $L^A \stackrel{\text{def}}{=} G^A + R_2^{H^A}$ is filterable by Statement 9.2.4.3, and $R_1^{L^A}$ by Statement 9.2.4.1.

Each construction gives a specific code for the corresponding `liftOpt` function, and so we could derive the code for liftOpt_F that is guaranteed to obey the filter laws. However, keep in mind that there are several inequivalent ways of implementing a lawful `liftOpt` for this functor. For instance, the filtering operation for $H^\bullet$ could be defined similarly to that for `JillsCoupons` in Example 9.1.4.2 and not through a co-product construction. The constructions give one possibility out of many. The programmer needs to choose the implementation according to the business requirements at hand.

Example 9.3.1.3 (identity law of `deflate`) The function `deflate` : $F^{\mathbb{1}+A} \to F^A$ is available only for filterable functors; but the function with the inverse type signature, `inflate` : $F^A \to F^{\mathbb{1}+A}$, can be implemented for any functor F. (See definition of `inflate` in Section 9.2.1.) Assuming that a given functor F is filterable, show that `deflate` is a **left inverse** of `inflate`:

$$\text{inflate}_F \,⨾\, \text{deflate}_F = \text{id} \quad . \tag{9.61}$$

Also show that it is not a right inverse: $\text{deflate}_F \,⨾\, \text{inflate}_F \neq \text{id}$ for some functors F.

Solution We may assume that F satisfies the filtering laws. The function `inflate` can be equivalently written as

$$\text{inflate}_F = (x^{:A} \to \mathbb{0} + x)^{\uparrow F} = \text{pu}_{\text{Opt}}^{\uparrow F} = \left(\psi_{(_\to\text{true})}\right)^{\uparrow F} \quad .$$

Now we can use the identity law (9.4) to derive

$$\text{filt}_F(_ \to \text{true}) = \psi_{(_\to\text{true})}^{\uparrow F} \,⨾\, \text{deflate}_F = \text{inflate}_F \,⨾\, \text{deflate}_F \quad .$$

Since the identity law says $\text{filt}_F(_ \to \text{true}) = \text{id}$, we obtain $\text{inflate}_F \,⨾\, \text{deflate}_F = \text{id}$.

To show that the inverse equation does not always hold, we need to find an explicit example: a specific functor F and a value $x^{:F^{\mathbb{1}+A}}$ such that $x \triangleright \text{deflate}_F \,⨾\, \text{inflate}_F \neq x$. Looking at the simplest filterable functors, we find that the constant functor $F^A \stackrel{\text{def}}{=} Z$ is not a suitable example because all its methods are identity functions. The next nontrivial example is the `Option` functor, $F^A \stackrel{\text{def}}{=} \mathbb{1} + A = \text{Opt}^A$. With the equivalence $\text{Opt}^{\text{Opt}^A} \cong \mathbb{1} + \mathbb{1} + A$, we write the methods `inflate` and `deflate` as

```
def inflate[A]: Option[A] => Option[Option[A]] = _.map(x => Some(x))
def deflate[A]: Option[Option[A]] => Option[A] = _.flatten
```

$$\text{inflate}_{\text{Opt}} \stackrel{\text{def}}{=} \text{pu}_{\text{Opt}}^{\uparrow\text{Opt}} = \begin{array}{c||ccc} & \mathbb{1} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} \quad , \qquad \text{deflate}_{\text{Opt}} \stackrel{\text{def}}{=} \text{ftn}_{\text{Opt}} = \begin{array}{c||cc} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} \quad .$$

The composition $\text{deflate}_{\text{Opt}} ⨾ \text{inflate}_{\text{Opt}}$ is computed as

$$\text{deflate}_{\text{Opt}} ⨾ \text{inflate}_{\text{Opt}} = \begin{array}{|c||cc|} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} ⨾ \begin{array}{|c||ccc|} & \mathbb{1} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array}$$

$$\text{matrix composition}: \quad = \begin{array}{|c||ccc|} & \mathbb{1} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} \\ \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} \neq \text{id} = \begin{array}{|c||ccc|} & \mathbb{1} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} \\ \mathbb{1} & \mathbb{0} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} .$$

The result differs from an identity matrix in the second row. A value $x^{:\mathbb{1}+\mathbb{1}+A} \stackrel{\text{def}}{=} \mathbb{0}^{:\mathbb{1}} + 1 + \mathbb{0}^{:A}$ will give a non-void result in the second row of the matrix, showing the difference:

$$(\mathbb{0}+1+\mathbb{0}) ▷ \begin{array}{|c||ccc|} & \mathbb{1} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} \\ \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} = \left| \begin{array}{ccc} \mathbb{0} & 1 & \mathbb{0} \end{array} \right| ▷ \left\| \begin{array}{ccc} \text{id} & \mathbb{0} & \mathbb{0} \\ \text{id} & \mathbb{0} & \mathbb{0} \\ \mathbb{0} & \mathbb{0} & \text{id} \end{array} \right| = \left| \begin{array}{ccc} 1 & \mathbb{0} & \mathbb{0} \end{array} \right| \neq \left| \begin{array}{ccc} \mathbb{0} & 1 & \mathbb{0} \end{array} \right| .$$

In Scala code, this value x is `Some(None)`, so the calculation corresponds to

```
inflate(deflate(Some(None))) == None  // Would have been Some(None) if the function were an identity.
```

Example 9.3.1.4 Assume that a given functor $H^A \stackrel{\text{def}}{=} \mathbb{1} + K^A$ is filterable (but $K^\bullet$ is not necessarily filterable). The functor $H^\bullet$ is a "data wrapper" with a fixed empty value, $1 + \mathbb{0}^{:K^A}$. Show that an empty wrapper remains empty after any filtering: the function filt_H satisfies, for any $p^{:A\to 2}$,

$$(1 + \mathbb{0}^{:K^A}) ▷ \text{filt}_H(p) = 1 + \mathbb{0}^{:K^A} \quad . \tag{9.62}$$

Solution We know nothing about $H^\bullet$ and $K^\bullet$ other than the fact that filt_H obeys the filtering laws. Rewrite Eq. (9.62) via the simpler function `deflate`, which is equivalent to filt_H:

$$(1 + \mathbb{0}^{:K^A}) ▷ \psi_p^{\uparrow H} ⨾ \text{deflate}_H \stackrel{?}{=} 1 + \mathbb{0}^{:K^A} \quad .$$

Any function lifted to $H^\bullet$ works separately for the two parts of the disjunctive type $H^A = \mathbb{1} + K^A$. So

$$(1 + \mathbb{0}^{:K^A}) ▷ f^{\uparrow H} = 1 + \mathbb{0}^{:K^B} \quad ,$$

regardless of the choice of $f^{:A\to B}$. Setting $f \stackrel{\text{def}}{=} \psi_p$, we obtain

$$(1 + \mathbb{0}^{:K^A}) ▷ \psi_p^{\uparrow H} = 1 + \mathbb{0}^{:K^{\mathbb{1}+A}} \quad . \tag{9.63}$$

It remains to show that

$$(1 + \mathbb{0}^{:K^{\mathbb{1}+A}}) ▷ \text{deflate}_H \stackrel{?}{=} 1 + \mathbb{0}^{:K^A} \quad . \tag{9.64}$$

We can proceed only if we use some law of `deflate`; Example 9.3.1.3 shows a suitable law. Since Eq. (9.63) holds for all predicates p, we can choose the predicate p to be identically `true` and get

$$(1 + \mathbb{0}^{:K^A}) ▷ \psi_{(_\to\text{true})}^{\uparrow H} = (1 + \mathbb{0}^{:K^A}) ▷ \text{inflate}_H = 1 + \mathbb{0}^{:K^{\mathbb{1}+A}} \quad . \tag{9.65}$$

Substituting this into Eq. (9.64), we find

$$\begin{aligned}
\text{expect to equal } 1+\mathbb{0}^{:K^A}: \quad & \underline{(1+\mathbb{0}^{:K^{\mathbb{1}+A}})} \triangleright \text{deflate}_H \\
\text{use Eq. (9.65)}: \quad & = (1+\mathbb{0}^{:K^A}) \triangleright \underline{\text{inflate}_H \,\text{⨾}\, \text{deflate}_H} \\
\text{identity law (9.61) of deflate}_H: \quad & = (1+\mathbb{0}^{:K^A}) \triangleright \underline{\text{id}} = 1+\mathbb{0}^{:K^A} \quad .
\end{aligned}$$

This completes the proof of Eq. (9.64) and of the property (9.62). The same property can be expressed in terms of liftOpt_H as

$$(1+\mathbb{0}^{:K^B}) \triangleright \text{liftOpt}_H(f^{:A\to\mathbb{1}+B}) = 1+\mathbb{0}^{:K^B} \quad . \tag{9.66}$$

Example 9.3.1.5 Assuming that $G^\bullet$ and $H^\bullet$ are filterable functors and $H^\bullet$ is of the form $H^A \stackrel{\text{def}}{=} \mathbb{1}+K^A$ (where $K^\bullet$ is not necessarily filterable), prove that the functor $F^\bullet \stackrel{\text{def}}{=} G^{K^\bullet}$ is filterable.

Solution We need to define liftOpt_F and verify its laws, assuming that liftOpt_G and liftOpt_H,

$$\text{liftOpt}_G(f^{:A\to\mathbb{1}+B}) : G^A \to G^B \quad , \qquad \text{liftOpt}_H(f^{:A\to\mathbb{1}+B}) : \mathbb{1}+K^A \to \mathbb{1}+K^B \quad ,$$

are already available and obey the same laws. We need to implement the type signature

$$\text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) : G^{K^A} \to G^{K^B} \quad .$$

We can map G^{K^A} to G^{K^B} using $\text{liftOpt}_G^{K^A,K^B}(k)$ if we supply k of type $K^A \to \mathbb{1}+K^B$ and use the type parameters K^A, K^B instead of A, B. We can compute the function k as $\text{liftOpt}_H(f^{:A\to\mathbb{1}+B})$ if we extend the argument to $\mathbb{1}+K^A$ instead of K^A using the function pu_{Opt} with type parameter K^A:

$$\text{pu}_{\text{Opt}}^{K^A} : K^A \to \mathbb{1}+K^A \quad , \qquad \text{pu}_{\text{Opt}}^{K^A} \,\text{⨾}\, \text{liftOpt}_H^{K^B,K^B}(f^{:A\to\mathbb{1}+B}) : K^A \to \mathbb{1}+K^B \quad .$$

Now we are ready to write the code for liftOpt_F as

$$\text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) \stackrel{\text{def}}{=} \text{liftOpt}_G^{K^A,K^B}\left(\text{pu}_{\text{Opt}}^{K^A} \,\text{⨾}\, \text{liftOpt}_H(f)\right) \quad .$$

It remains to verify the laws. The naturality-identity law (9.29) for liftOpt_F:

$$\begin{aligned}
\text{expect to equal } f^{\uparrow F}: \quad & \text{liftOpt}_F(f \,\text{⨾}\, \text{pu}_{\text{Opt}}) = \text{liftOpt}_G^{K^A,K^B}\left(\text{pu}_{\text{Opt}}^{K^A} \,\text{⨾}\, \underline{\text{liftOpt}_H(f \,\text{⨾}\, \text{pu}_{\text{Opt}})}\right) \\
\text{law (9.29) of liftOpt}_H: \quad & = \text{liftOpt}_G^{K^A,K^B}\left(\text{pu}_{\text{Opt}}^{K^A} \,\text{⨾}\, \underline{f^{\uparrow H}}\right) \\
\text{lifting } {}^{\uparrow H} \text{ expressed via } {}^{\uparrow K}: \quad & = \text{liftOpt}_G^{K^A,K^B}\left(\underline{\text{pu}_{\text{Opt}}^{K^A} \,\text{⨾}\, f^{\uparrow K \uparrow \text{Opt}}}\right) \\
\text{naturality (8.8) of pu}_{\text{Opt}}: \quad & = \text{liftOpt}_G^{K^A,K^B}\left(f^{\uparrow K} \,\text{⨾}\, \text{pu}_{\text{Opt}}^{K^B}\right) \\
\text{law (9.29) of liftOpt}_G: \quad & = \underline{(f^{\uparrow K})^{\uparrow G}} = f^{\uparrow F} \quad .
\end{aligned}$$

To verify the composition law (9.31) for liftOpt_F:

$$\begin{aligned}
\text{expect to equal liftOpt}_F(f \diamond_{\text{Opt}} g): \quad & \text{liftOpt}_F(f) \,\text{⨾}\, \text{liftOpt}_F(g) \\
\text{definition of liftOpt}_F: \quad & = \underline{\text{liftOpt}_G}\left(\text{pu}_{\text{Opt}} \,\text{⨾}\, \text{liftOpt}_H(f)\right) \underline{\,\text{⨾}\, \text{liftOpt}_G}\left(\text{pu}_{\text{Opt}} \,\text{⨾}\, \text{liftOpt}_H(g)\right) \\
\text{law (9.31) for liftOpt}_G: \quad & = \text{liftOpt}_G\left(\left(\text{pu}_{\text{Opt}} \,\text{⨾}\, \text{liftOpt}_H(f)\right) \underline{\diamond_{\text{Opt}} \left(\text{pu}_{\text{Opt}} \,\text{⨾}\, \text{liftOpt}_H(g)\right)}\right) \\
\text{simplify using Eq. (9.67)}: \quad & = \text{liftOpt}_G\left(\text{pu}_{\text{Opt}} \,\text{⨾}\, \underline{\text{liftOpt}_H(f) \,\text{⨾}\, \text{liftOpt}_H(g)}\right) \\
\text{law (9.31) for liftOpt}_H: \quad & = \text{liftOpt}_G\left(\text{pu}_{\text{Opt}} \,\text{⨾}\, \text{liftOpt}_H(f \diamond_{\text{Opt}} g)\right) = \text{liftOpt}_F(f \diamond_{\text{Opt}} g) \quad .
\end{aligned}$$

In this derivation, we used a property that simplifies the Kleisli composition with pu_{Opt}:

$$f^{:K^A\to\mathbb{1}+K^B}\diamond_{\text{Opt}}(\text{pu}_{\text{Opt}}^{K^B}\,;\,\text{liftOpt}_H^{\mathbb{1}+K^B,\mathbb{1}+K^C}(g^{:B\to\mathbb{1}+C}))=f\,;\,\text{liftOpt}_H(g)\quad.\tag{9.67}$$

This simplification depends on the property (9.66) of liftOpt_H shown in Example 9.3.1.4 and does *not* work for arbitrary functions p, q having the same type signatures as in Eq. (9.67),

$$p^{:A\to\mathbb{1}+B}\diamond_{\text{Opt}}(\text{pu}_{\text{Opt}}^{:B\to\mathbb{1}+B}\,;\,q^{:\mathbb{1}+B\to\mathbb{1}+C})\neq p\,;\,q\quad.$$

To prove Eq. (9.67), use the code for $\diamond_{\text{Opt}}$ and pu_{Opt} from Eq. (9.33):

$$\begin{aligned}
&\text{left-hand side of Eq. (9.67)}: && f\underline{\diamond_{\text{Opt}}}(\text{pu}_{\text{Opt}}^{K^B}\,;\,\text{liftOpt}_H(g))\\
&\text{definition (9.32) of }\diamond_{\text{Opt}}: && =f\,;\,\text{flm}_{\text{Opt}}(\underline{\text{pu}_{\text{Opt}}}^{K^B}\,;\,\text{liftOpt}_H(g))\\
&\text{definition of }\text{pu}_{\text{Opt}}: && =f\,;\,\underline{\text{flm}_{\text{Opt}}}(x^{:K^B}\to(\mathbb{0}+x)\triangleright\text{liftOpt}_H(g))\\
&\text{use Eq. (9.33)}: && =f^{:K^A\to\mathbb{1}+K^B}\,;\,\left|\begin{array}{c||c|}
 & \mathbb{1}+K^C\\
\hline
\mathbb{1} & 1\to 1+\mathbb{0}^{:K^C}\\
K^B & x\to(\mathbb{0}+x)\triangleright\text{liftOpt}_H(g)
\end{array}\right|\quad.
\end{aligned}$$

We expect the last expression to equal the right-hand side of Eq. (9.67), which is $f\,;\,\text{liftOpt}_H(g)$. To be able to compare these expressions, we rewrite $\text{liftOpt}_H(g)$ equivalently as a matrix:

$$\text{liftOpt}_H(g^{:B\to\mathbb{1}+C})=\left|\begin{array}{c||c|}
 & \mathbb{1}+K^C\\
\hline
\mathbb{1} & 1\to(1+\mathbb{0}^{:K^C})\triangleright\text{liftOpt}_H(g)\\
K^B & x\to(\mathbb{0}+x)\triangleright\text{liftOpt}_H(g)
\end{array}\right|\quad.$$

Usually, such an equivalent rewriting gives no advantages; but in this case, Eq. (9.66) simplifies the first row of the matrix to $1\to 1+\mathbb{0}^{:K^C}$. This makes both sides of Eq. (9.66) equal.

Example 9.3.1.6 Prove that the recursive functor $F^A\stackrel{\text{def}}{=}\mathbb{1}+A\times A+A\times(\mathbb{1}+A)\times F^A$ is filterable.

Solution Rather than proving the laws by hand (as we did in a similar case in Statement 9.2.4.6), we will use a trick that will make calculations shorter. The trick is to "unroll" the recursive equation and to reduce F^A to the `List` functor, which has a standard filtering operation.

The unrolling trick gives, for any recursive definition of the form $F^A\stackrel{\text{def}}{=}P^A+Q^A\times F^A$:

$$\text{if}\quad F^A\stackrel{\text{def}}{=}P^A+Q^A\times F^A\quad\text{then}\quad F^A\cong P^A\times\text{List}^{Q^A}\quad,\tag{9.68}$$

where $P^\bullet$ and $Q^\bullet$ are arbitrary functors. The functor F^A given in this example is of this form with the functors $P^A\stackrel{\text{def}}{=}\mathbb{1}+A\times A$ and $Q^A\stackrel{\text{def}}{=}A\times(\mathbb{1}+A)$. Comparing with the definition of the `List` functor,

$$\begin{aligned}
\text{List}^A\stackrel{\text{def}}{=}\mathbb{1}+A\times\text{List}^A&\cong\mathbb{1}+A\times(\mathbb{1}+A\times(\mathbb{1}+...(\mathbb{1}+A\times\text{List}^A)))\\
&\cong\mathbb{1}+A+A\times A+A\times A\times A+...+\underbrace{A\times...\times A}_{n\text{ times}}\times\text{List}^A\quad,
\end{aligned}$$

we find that the definition (9.68) of F^A can be "unrolled" n times as

$$\begin{aligned}
F^A\stackrel{\text{def}}{=}P^A+Q^A\times F^A&\cong P^A+Q^A\times(P^A+Q^A\times(P^A+...(P^A+Q^A\times F^A)))\\
&\cong P^A+Q^A\times P^A+Q^A\times Q^A\times P^A+...+\underbrace{Q^A\times...\times Q^A}_{n\text{ times}}\times F^A\\
&\cong P^A\times\big(\mathbb{1}+Q^A+Q^A\times Q^A+...\underbrace{Q^A\times...\times Q^A}_{n-1\text{ times}}\big)+\underbrace{Q^A\times...\times Q^A}_{n\text{ times}}\times F^A\quad.
\end{aligned}$$

The type equivalence (9.68), $F^A \cong P^A \times \text{List}^{Q^A}$, follows by induction on the number of "unrolled" functors F. We can now use the functor product construction for $F^\bullet$ if we show that $\text{List}^{Q^\bullet}$ is filterable. This does not follow by functor composition because $Q^A \stackrel{\text{def}}{=} A \times (\mathbb{1} + A)$ is *not* filterable (we saw that in Section 9.1.3). However, we can derive from Statement 9.2.4.4, setting $G^A \stackrel{\text{def}}{=} \mathbb{1} + A$, that

$$H^A \stackrel{\text{def}}{=} \mathbb{1} + A \times (\mathbb{1} + A) = \mathbb{1} + Q^A$$

is filterable. So, we can use the result of Example 9.3.1.5 and conclude that $\text{List}^{Q^\bullet}$ is filterable.

Example 9.3.1.7 Show that the contrafunctor $C^A \stackrel{\text{def}}{=} A \to Z$ is not filterable (where the fixed type Z does not have a known default value).

Solution Try to implement the function $\text{inflate}_C : C^A \to C^{\mathbb{1}+A}$,

$$\text{inflate}_C : (A \to Z) \to \mathbb{1} + A \to Z \quad , \qquad \text{inflate}_C = c^{:A\to Z} \to p^{:\mathbb{1}+A} \to ???^{:Z} \quad .$$

The only way to fill the type hole $???^{:Z}$ is to apply c to an argument of type A. However, we do not have values of type A; we only have p of type $\mathbb{1} + A$, and p might be the "empty" value, $1 + \mathbb{0}$. So, it is impossible to implement inflate_C, and we conclude that C is not filterable.

Example 9.3.1.8 Given a filterable functor $F^\bullet$, show that the type $F^{\mathbb{1}} \to F^{\mathbb{0}}$ is *not* void.

Solution A filterable functor must have a `deflate` function with type signature $F^{\mathbb{1}+A} \to F^A$. Set the type parameter $A = \mathbb{0}$ in the code of `deflate` to obtain the code for a function of type $F^{\mathbb{1}} \to F^{\mathbb{0}}$.

9.3.2 Exercises

Exercise 9.3.2.1 Implement a `Filterable` instance for the functor `F[T] = G[H[T]]` assuming that the contrafunctor `H[_]` already has a `Filterable` instance and `G[_]` is an arbitrary contrafunctor. Verify the laws of filterable functor rigorously (by symbolic derivations, not tests).

Exercise 9.3.2.2 Implement a `Filterable` instance for `F[T] = Option[Int => Option[(T, T)]]`. Show that the laws hold by using known constructions (avoiding explicit proofs or tests).

Exercise 9.3.2.3 Prove rigorously (not by tests) that $\text{flm}_{\text{Opt}}(\text{pu}_{\text{Opt}}) = \text{id}^{:\mathbb{1}+A\to\mathbb{1}+A}$.

Exercise 9.3.2.4 Show that one can define `deflate` : $C^{\mathbb{1}+A} \to C^A$ for any contrafunctor C^A (not necessarily filterable). Prove that *in case* $C^\bullet$ is filterable, the **identity law** will hold:

$$\text{inflate}_C \,⨾\, \text{deflate}_C = \text{id}^{:C^A\to C^A} \quad .$$

Show that the inverse equation does not hold in general: $\text{deflate}_C \,⨾\, \text{inflate}_C \neq \text{id}$.

Exercise 9.3.2.5 Assuming that $G^\bullet$ is a filterable functor, prove rigorously that the recursive functor $F^A \stackrel{\text{def}}{=} G^A + \text{Int} \times A \times A \times A \times F^A$ is filterable. Implement a `Filterable` instance for $F^\bullet$.

Exercise 9.3.2.6 Show that the functor $F^A \stackrel{\text{def}}{=} A + (\text{Int} \to A)$ is not filterable.

Exercise 9.3.2.7 Prove that $F^A \stackrel{\text{def}}{=} \mathbb{1} + A \times G^A$ is in general not filterable if G^A is an arbitrary (non-filterable) functor; give an example of a suitable G^A. Since $F^{\mathbb{1}} \to F^{\mathbb{0}} \cong \mathbb{1} + G^{\mathbb{1}} \to \mathbb{1} \cong \mathbb{1}$, this will demonstrate that Example 9.3.1.8 gives a necessary but not a sufficient condition for a functor $F^\bullet$ to be filterable.

Exercise 9.3.2.8 Show that $F^A \stackrel{\text{def}}{=} \mathbb{1} + G^A + H^A$ is filterable if $\mathbb{1} + G^A$ and $\mathbb{1} + H^A$ are filterable (even when $G^\bullet$ and $H^\bullet$ are not filterable).

Exercise 9.3.2.9 Show that $C^A \stackrel{\text{def}}{=} A + A \times A \to \mathbb{1} + Z$ is a filterable contrafunctor (no law checking).

Exercise 9.3.2.10 Verify Eqs. (9.46)–(9.47) in Scala syntax and in the code notation.

Exercise 9.3.2.11 If G^A and H^A are filterable contrafunctors, prove that the contrafunctors $G^A \times H^A$ and $G^A + H^A$ are also filterable.

Exercise 9.3.2.12 Show that the contrafunctor $C^A \stackrel{\text{def}}{=} A \times F^A \to Z$ is *not* filterable for any functor $F^\bullet$ and any fixed type Z that does not have a known default value.

Exercise 9.3.2.13 Show that a *necessary* condition for a contrafunctor $C^\bullet$ to be filterable is that a function of type $C^{\mathbb{0}} \to C^{\mathbb{1}}$ can be implemented (i.e., the type $C^{\mathbb{0}} \to C^{\mathbb{1}}$ is not void).

Exercise 9.3.2.14* Show that a *polynomial* functor $F^\bullet$ is filterable (in some way) if and only if the type $F^{\mathbb{1}} \to F^{\mathbb{0}}$ is not void. Find an example of a non-filterable polynomial functor $F^\bullet$ that violates the condition $F^{\mathbb{1}} \to F^{\mathbb{0}} \not\cong \mathbb{0}$.

9.4 Further developments

9.4.1 Naturality laws and natural transformations

While deriving various laws, we often need to interchange the order of compositions that involve lifted functions. For instance, in the derivation of Example 9.2.5.4, we needed to move $\psi_p^{\downarrow C}$ to the left of filt_C in the expression $\text{get}^{\downarrow C} ⨾ \text{filt}_C(\text{nonEmpty}) ⨾ \psi_p^{\downarrow C}$, or else we could make no progress with the calculations. The required interchange was possible by using the law (9.58),

$$\text{filt}_C(p) ⨾ f^{\downarrow C} = f^{\downarrow C} ⨾ \text{filt}_C(f ⨾ p) \quad .$$

In this and previous chapters, we discovered a number of laws of that form, e.g.:

Eq. (8.8) :	$\text{pu}_F ⨾ f^{\uparrow F} = f ⨾ \text{pu}_F$	for	$\text{pu}_F : A \to F^A$
Eq. (8.11) :	$\text{ex}_F ⨾ f = f^{\uparrow F} ⨾ \text{ex}_F$	for	$\text{ex}_F : F^A \to A$
Eq. (8.14) :	$\text{ex}_S ⨾ f = \text{bimap}_S(f)(f) ⨾ \text{ex}_S$	for	$\text{ex}_S : S^{A,A} \to A$
Eq. (9.3) :	$f^{\uparrow F} ⨾ \text{filt}_F(p) = \text{filt}_F(f ⨾ p) ⨾ f^{\uparrow F}$	for	$\text{filt}_F : (A \to \mathbb{2}) \to F^A \to F^A$
Eq. (9.14) :	$\text{deflate} ⨾ f^{\uparrow F} = f^{\uparrow \text{Opt} \uparrow F} ⨾ \text{deflate}$	for	$\text{deflate}_F : F^{\mathbb{1}+A} \to F^A$
Eq. (9.18) :	$f ⨾ \psi_p = \psi_{f ⨾ p} ⨾ f^{\uparrow \text{Opt}}$	for	$\psi : (A \to \mathbb{2}) \to A \to \mathbb{1} + A$
Eq. (9.23) :	$h^{\uparrow F} ⨾ \text{liftOpt}_F(f) = \text{liftOpt}_F(h ⨾ f)$	for	$\text{liftOpt}_F : (A \to \text{Opt}^B) \to F^A \to F^B$
Eq. (9.35) :	$\text{liftOpt}_F(f) ⨾ h^{\uparrow F} = \text{liftOpt}_F(f ⨾ h^{\uparrow \text{Opt}})$	for	$\text{liftOpt}_F : (A \to \text{Opt}^B) \to F^A \to F^B$
Eq. (9.58) :	$\text{filt}_C(p) ⨾ f^{\downarrow C} = f^{\downarrow C} ⨾ \text{filt}_C(f ⨾ p)$	for	$\text{filt}_C : (A \to \mathbb{2}) \to C^A \to C^A$
Eq. (9.59) :	$f^{\downarrow C} ⨾ \text{inflate}_C = \text{inflate}_C ⨾ f^{\uparrow \text{Opt} \downarrow C}$	for	$\text{inflate}_C : C^A \to C^{\mathbb{1}+A}$

We called all these laws "naturality laws", although they were derived from different premises and do not look similar at first sight. Is there a common pattern for these laws? For a given function, can we guess the form of its naturality law?

Looking at the examples shown above, we find that each law follows one of the four patterns shown in the following table, where we use arbitrary functors F, G, H and contrafunctors C:

Pattern	Type signature	Naturality law
natural transformation	$t^A : F^A \to G^A$	$(f^{:A\to B})^{\uparrow F} ⨾ t^B = t^A ⨾ f^{\uparrow G}$
parameterized transformation	$t^A : C^A \to F^A \to G^A$	$f^{\uparrow F} ⨾ t^B(c) = t(c ▸ f^{\downarrow C}) ⨾ f^{\uparrow G}$
generalized lifting; A-naturality	$t^{A,B} : (A \to G^B) \to F^A \to F^B$	$(f^{:A\to B})^{\uparrow F} ⨾ t^{B,C}(p) = t^{A,C}(f ⨾ p)$
generalized lifting; B-naturality	$t^{A,B} : (A \to G^B) \to F^A \to F^B$	$t^{A,B}(p) ⨾ (g^{:B\to C})^{\uparrow F} = t^{A,C}(p ⨾ g^{\uparrow G})$

Let us now look at each of these patterns in detail.

Natural transformations This pattern covers functions with type signatures of the form $F^A \to G^A$, where F and G must be both functors or both contrafunctors. Functions of that kind, i.e., fully parametric functions with type signatures of the form $t^A : F^A \to G^A$, are called **natural transformations** between F and G. Heuristically, we may view t^A as a function that copies data of type A from one

"wrapper" to another and may rearrange that data in a way that does not depend on the type A. An example is the `headOption` method defined in Scala on various sequence types such as `List`,

$$\text{headOpt} : \text{List}^A \to \text{Opt}^A \quad .$$

We can write an equivalent Scala code for this function as

```
def headOption[A]: List[A] => Option[A] = {
  case List()   => None
  case x :: _   => Some(x)
}
```

It is clear that this code works in the same way for all types A. This property is formulated mathematically as the requirement that we may first transform a list with a lifted function $(f^{:A\to B})^{\uparrow\text{List}}$ and then apply `headOption`; or we may first apply `headOption` and then transform the data with a lifted function f (and we will then need to lift f to the `Option` functor rather than to the `List` functor); the results will be equal. We write this requirement as an equation called the naturality law of `headOption`:

$$\begin{tikzcd} \text{List}^A \arrow[r, "\text{headOpt}^A"] \arrow[d, "(f^{:A\to B})^{\uparrow\text{List}}"'] & \text{Opt}^A \arrow[d, "f^{\uparrow\text{Opt}}"] \\ \text{List}^B \arrow[r, "\text{headOpt}^B"] & \text{Opt}^B \end{tikzcd}$$

$$(f^{:A\to B})^{\uparrow\text{List}} ⨾ \text{headOpt}^B = \text{headOpt}^A ⨾ f^{\uparrow\text{Opt}} \quad .$$

It is important to keep in mind that this law does not depend on the fact that `headOption` extracts the *first* element of a list. The same naturality law will hold for any fully parametric function of type $\text{List}^A \to \text{Opt}^A$, e.g., the functions `_.lastOption` or `_.drop(2).headOption`. The naturality law only expresses the property that `headOption` works in the same way for all types without examining any specific values in the list.

Other examples of natural transformations are the functions `pure`, `deflate`, and `inflate`, whose naturality laws we have already seen. All these naturality laws are captured by the "natural transformation" pattern, which we can formulate for arbitrary functors $F^\bullet$ and $G^\bullet$ as the following law:

$$\begin{tikzcd} F^A \arrow[r, "t^A"] \arrow[d, "(f^{:A\to B})^{\uparrow F}"'] & G^A \arrow[d, "f^{\uparrow G}"] \\ F^B \arrow[r, "t^B"] & G^B \end{tikzcd}$$

$$(f^{:A\to B})^{\uparrow F} ⨾ t^{:F^B\to G^B} = t^{:F^A\to G^A} ⨾ f^{\uparrow G} \quad . \tag{9.69}$$

Once we recognize that a given function $t^A : F^A \to G^A$ has the type signature of a natural transformation, how can we remember its naturality law? A naturality law always involves an arbitrary function $f^{:A\to B}$ with two type parameters. Types will match only if the function f is lifted to the functor F when f is applied before t, and to the functor G when f is applied after t. So, the naturality law is $f^{\uparrow F} ⨾ t = t ⨾ f^{\uparrow G}$ with appropriate type parameters.

The analogous naturality law for natural transformations $t : C^A \to D^A$ between contrafunctors C and D has exactly the same form, but the order of type parameters must be swapped:

$$\begin{tikzcd} C^A \arrow[r, "t^A"] \arrow[d, "(f^{:B\to A})^{\downarrow C}"'] & D^A \arrow[d, "f^{\downarrow D}"] \\ C^B \arrow[r, "t^B"] & D^B \end{tikzcd}$$

$$(f^{:B\to A})^{\downarrow C} ⨾ t^{:C^B\to D^B} = t^{:C^A\to D^A} ⨾ f^{\downarrow D} \quad . \tag{9.70}$$

Examples of this pattern are the naturality law (8.15) for pu_D (where we need to set $C^A \stackrel{\text{def}}{=} \mathbb{1}$) and the naturality law (9.59) for inflate_C.

Parameterized transformations The second pattern, which may be called "parameterized transformation", is a curried function that takes a first argument of type C^A and returns a natural transformation of type $F^A \to G^A$. This pattern is recognized by a type signature $C^A \to F^A \to G^A$ where C is a contrafunctor while F and G are functors (or vice versa, C is a functor while F and G are contrafunctors). Examples of parameterized transformations are the methods `filter`, `takeWhile`, and `find`, defined in the Scala library for sequence-like type constructors (e.g., `List`, `Vector`, or `Array`).

For functions of this kind, naturality laws must modify the argument of type C^A when changing the order of lifted functions. In order to formulate the naturality law for a parameterized transformation $t^A : C^A \to F^A \to G^A$, use an arbitrary function $f^{:A\to B}$ and first transform the argument of type F^A into F^B using $f^{\uparrow F}$ before applying $t^B(c)$:

$$f^{\uparrow F} ⨾ t^B(c^{:C^B}) = ???^{:F^A\to G^B} \quad .$$

It is clear that we must choose an arbitrary value c of type C^B (rather than C^A) for all types to match. It remains to fill the typed hole, which must be of the form $t(...) \fatsemi f^{\uparrow G}$,

$$f^{\uparrow F} \fatsemi t^B(c^{:C^B}) = t^A(???^{:C^A}) \fatsemi f^{\uparrow G} \quad .$$

The argument of t of type C^A is obtained by applying $f^{\downarrow C}$ to c. So, the law is

$$f^{\uparrow F} \fatsemi t(c) = t(c \triangleright f^{\downarrow C}) \fatsemi f^{\uparrow G} \quad . \tag{9.71}$$

A similar law can be derived for the case when $C^\bullet$ is a functor and $F^\bullet, G^\bullet$ are contrafunctors.

Parameterized transformations $t : C^A \to F^A \to G^A$ are reduced to natural transformations if we swap the order of curried arguments to $F^A \to C^A \to G^A$ and define the functor $H^A \stackrel{\text{def}}{=} C^A \to G^A$. Then $F^A \to C^A \to G^A = F^A \to H^A$, which is a type signature of a natural transformation. Denoting by $\tilde{t} : F^A \to H^A$ the function t with its arguments swapped, we can write the naturality law of $\tilde{t}$ as

$$f^{\uparrow F} \fatsemi \tilde{t} = \tilde{t} \fatsemi f^{\uparrow H} \quad , \quad \text{where} \quad \tilde{t} \stackrel{\text{def}}{=} p^{:F^A} \to c^{:C^A} \to p \triangleright t(c) \quad .$$

To show that this law is equivalent to Eq. (9.71), we use the definition of $^{\uparrow H}$,

$$f^{\uparrow H} \stackrel{\text{def}}{=} h^{:H^A} \to c^{:C^B} \to c \triangleright f^{\downarrow C} \fatsemi h \fatsemi f^{\uparrow G} \quad .$$

Substituting this into the naturality law of $\tilde{t}$, we obtain the naturality law (9.71):

$$\begin{aligned}
\text{left-hand side applied to } p^{:F^A} : \quad & p \triangleright f^{\uparrow F} \fatsemi \underline{\tilde{t}} = \underline{p \triangleright f^{\uparrow F}} \fatsemi (\underline{p} \to c \to \underline{p} \triangleright t(c)) \\
\text{apply function to } p \triangleright f^{\uparrow F} : \quad & = c \to p \triangleright f^{\uparrow F} \triangleright t(c) = c \to p \triangleright f^{\uparrow F} \fatsemi t(c) \quad , \\
\text{left-hand side applied to } p^{:F^A} : \quad & \underline{p \triangleright (\underline{p} \to c \to p \triangleright t(c))} \fatsemi f^{\uparrow H} = (c \to p \triangleright t(c)) \triangleright \underline{f^{\uparrow H}} \\
\text{definition of } f^{\uparrow H} : \quad & = c \to \underline{c \triangleright f^{\downarrow C} \fatsemi} (c \to p \triangleright t(\underline{c})) \fatsemi f^{\uparrow G} \\
\text{apply function to } c \triangleright f^{\downarrow C} : \quad & = c \to p \triangleright t(c \triangleright f^{\downarrow C}) \fatsemi f^{\uparrow G} \quad .
\end{aligned}$$

So, we have reduced both sides of $\tilde{t}$'s law to the two sides of the law (9.71) applied to an arbitrary $p^{:F^A}$ and considered as functions of $c^{:C^B}$.

Reduction to natural transformations works similarly when $C^\bullet$ is a functor and $F^\bullet, G^\bullet$ are contrafunctors. A naturality law of $t : C^A \to F^A \to G^A$ can then be derived from t's type signature.

Liftings A "lifted" function $f^{\uparrow F}$ is the result of applying a functor F's `map` operation to a function f. The type signatures of `flatMap` and `liftOpt` are similar to that of `map` except for using the function type $A \to \text{Opt}^B$ instead of $A \to B$:

$$\begin{aligned}
&\text{fmap}_F : (A \to B) \to F^A \to F^B \quad , \\
&\text{flm}_{\text{Opt}} : (A \to \text{Opt}^B) \to \text{Opt}^A \to \text{Opt}^B \quad , \\
&\text{liftOpt}_F : (A \to \text{Opt}^B) \to F^A \to F^B \quad .
\end{aligned}$$

Replacing Opt by an arbitrary functor G, we obtain the type signature of a "generalized lifting",

$$\text{lift}^{A,B}_{G,F} : (A \to G^B) \to F^A \to F^B \quad ,$$

which we can view as a lifting of functions with a "twisted" type $A \to G^B$ (which we call **Kleisli functions**) to functions of type $F^A \to F^B$. We will look at properties of generalized liftings in the next subsection. Here we focus on the naturality laws for generalized liftings.

A generalized lifting has two type parameters and two naturality laws. Looking at the two naturality laws (9.23), (9.35) for `liftOpt` or at the two laws (9.46), (9.47) for flm_{Opt}, we notice that each naturality law replaces one of the type parameters but keeps the other type parameter unchanged.

This motivates us to fix one of the type parameters in the type signature of $\text{lift}^{A,B}_{G,F}$. For fixed A, the function $\text{lift}^{A,B}_{G,F}$ is a natural transformation between functors $A \to G^\bullet$ and $F^A \to F^\bullet$. With a fixed B, the function $\text{lift}^{A,B}_{G,F}$ is a natural transformation between contrafunctors $\bullet \to G^B$ and $F^\bullet \to F^B$. One can show that the corresponding naturality laws for these two natural transformations are equivalent to the two naturality laws of a generalized lifting.

We have reduced the four patterns of naturality laws to the laws of natural transformations, Eq. (9.69) for functors and Eq. (9.70) for contrafunctors, which are easier to understand.

Parametricity theorem It turns out that the naturality law of a natural transformation $t : F^A \to G^A$ will *always hold* if the code of the function t is fully parametric. More precisely, naturality holds if the code of t is a combination of the eight standard code constructions (shown in Section 5.2.2) together with recursion. This is a consequence of the "parametricity theorem", which is beyond the scope of this chapter.[2] So, we do *not* need to verify naturality laws for functions whose code is known to be fully parametric. This saves a significant amount of work, since every method of every typeclass will have one naturality law per type parameter. Until now, we have been systematically deriving and checking all naturality laws; but we will not check those laws in the rest of the book.

Even if naturality laws hold automatically, it is important to be able to recognize their form and to use them in derivations where they are frequently needed. The mnemonic recipe for naturality laws (9.69)–(9.70) is that an arbitrary function f is lifted to the functor F at the left side of $t : F^A \to G^A$ and to the functor G at the right side of t, matching the two sides of the type signature $F^A \to G^A$.

All methods of typeclasses considered in this book are covered by the natural transformation recipe. However, not all type signatures of fully parametric functions can be reduced to natural transformations. For example, $t^A : (A \to A) \to A$ is not of the form $F^A \to G^A$ where F, G are either functors or contrafunctors. The parametricity theorem will still produce naturality laws for such functions; but we will not be able to write those laws via the natural transformation recipe. (A general procedure that works for all type signatures is given in Section D.1.1.)

9.4.2 Generalizing the laws of liftings. Kleisli functions

As we have seen in this chapter, the laws of filtering may be formulated equivalently via the `filter`, `deflate`, or `liftOpt` methods. These methods and their laws are equivalent but play different roles: `filter` is the most convenient to use in program code; `deflate` is the easiest type signature to implement and to reason about, especially in order to demonstrate that a functor is not filterable; `liftOpt` has the fewest laws and is most convenient for proofs of general type constructions.

If we put the naturality laws aside, `liftOpt` has the laws of identity (9.27) and composition (9.31). It is notable how these two laws are similar to the functor laws (6.2)–(6.3):

$$\text{liftOpt}_F(\text{pu}_F) = \text{id} \quad , \qquad \text{liftOpt}_F(f^{:A\to\mathbb{1}+B}) ⨾ \text{liftOpt}_F(g^{:B\to\mathbb{1}+C}) = \text{liftOpt}_F(f \diamond_{\text{Opt}} g) \quad .$$
$$\text{fmap}_F(\text{id}) = \text{id} \quad , \qquad \text{fmap}_F(f^{:A\to B}) ⨾ \text{fmap}_F(g^{:B\to C}) = \text{fmap}_F(f ⨾ g) \quad .$$

The only difference between these laws is in replacing $\text{id}^{:A\to A}$ by $\text{pu}_F^{:A\to F^A}$ and the function composition $f ⨾ g$ by the Kleisli composition $f \diamond_{\text{Opt}} g$. We will now focus on the analogy between these laws, which goes far beyond the superficial similarity of form.

Kleisli functions $f^{:A\to\mathbb{1}+B}$ and $g^{:B\to\mathbb{1}+C}$ cannot be composed as $f ⨾ g$ with the ordinary function composition. If we instead use the Kleisli composition, $f \diamond_{\text{Opt}} g$, the properties of Kleisli functions with respect to composition become completely analogous to the properties of the ordinary functions, except that the $\text{pu}_{\text{Opt}} : A \to \mathbb{1} + A$ plays the role of the identity ($\text{id}^{:A\to A}$).

[2] Formulations and proofs sufficient for the scope of this book are given in Appendix D.

Statement 9.4.2.1 The Kleisli composition $\diamond_{\text{Opt}}$ obeys the identity and the associativity laws:

$$\text{identity laws}: \quad \text{pu}_{\text{Opt}}^{:A\to\text{Opt}^A} \diamond_{\text{Opt}} g^{:A\to\text{Opt}^B} = g \quad , \qquad f^{:A\to\text{Opt}^B} \diamond_{\text{Opt}} \text{pu}_{\text{Opt}}^{:B\to\text{Opt}^B} = f ,$$
$$\text{associativity law}: \quad \left(f^{:A\to\text{Opt}^B} \diamond_{\text{Opt}} g^{:B\to\text{Opt}^C}\right) \diamond_{\text{Opt}} h^{:C\to\text{Opt}^D} = f \diamond_{\text{Opt}} \left(g \diamond_{\text{Opt}} h\right) \quad .$$

Proof Use the definitions (9.32) and Eqs. (9.34), (9.67), (9.40) derived previously in this chapter:

$$\text{use Eq. (9.34)}: \quad \text{pu}_{\text{Opt}} \diamond_{\text{Opt}} g = \text{pu}_{\text{Opt}} ⨾ \text{flm}_{\text{Opt}}(g) = g \quad ,$$
$$\text{use Exercise 9.3.2.3}: \quad f \diamond_{\text{Opt}} \text{pu}_{\text{Opt}} = f ⨾ \underline{\text{flm}_{\text{Opt}}(\text{pu}_{\text{Opt}})} = f ⨾ \text{id} = f \quad ,$$
$$\text{expect to equal } f \diamond_{\text{Opt}} \left(g \diamond_{\text{Opt}} h\right): \quad \left(f \diamond_{\text{Opt}} g\right) \diamond_{\text{Opt}} h = f ⨾ \underline{\text{flm}_{\text{Opt}}(g) ⨾ \text{flm}_{\text{Opt}}(h)}$$
$$\text{use Eq. (9.40)}: \quad = f ⨾ \text{flm}_{\text{Opt}}(g \diamond_{\text{Opt}} h) = f \diamond_{\text{Opt}} \left(g \diamond_{\text{Opt}} h\right) \quad .$$

This calculation motivates the name "associativity law" for Eq. (9.40).

The function `liftOpt` can be viewed as a "generalized lifting" from Kleisli functions $A \to \mathbb{1} + B$ to functions $F^A \to F^B$, just as `fmap` is a lifting from ordinary functions $A \to B$ to functions $F^A \to F^B$; the laws of composition and the laws of liftings are analogous. The close analogy between ordinary functions and Kleisli functions means that any proofs of properties of ordinary liftings can be mechanically translated into proofs of the corresponding properties of generalized liftings. Indeed, replacing `fmap` by `liftOpt` and id by pu_{Opt} where appropriate, we can translate the proof of Statement 6.2.3.3 (functor product), written using the pair product operation $\boxtimes$, into the proof of Statement 9.2.4.2 (filterable functor product). The same holds for the proofs of functor co-product and functor composition constructions.

The similarity between these proofs means, in the mathematical sense, that we have been proving essentially the same statements twice but did not look at the appropriate level of abstraction to see that. While programmers may accept the work of writing these proofs twice, a mathematician would prefer to define a "generalized lifting" that replaces Opt by a functor M,

$$\text{lift}_{M,F} : (A \to M^B) \to F^A \to F^B \quad ,$$
$$\text{pu}_M : A \to M^A \quad , \qquad \diamond_M : (A \to M^B) \to (B \to M^C) \to (A \to M^C) \quad ,$$

and postulating the required properties as the set of identity, associativity, and composition laws,

$$\text{lift}_{M,F}(\text{pu}_M^{:A\to M^A}) = \text{id}^{:F^A\to F^A} \quad , \qquad \text{lift}_{M,F}(f) ⨾ \text{lift}_{M,F}(g) = \text{lift}_{M,F}(f \diamond_M g) \quad ,$$
$$\text{pu}_M \diamond_M g = g \quad , \qquad f \diamond_M \text{pu}_M = f \quad , \qquad \left(f \diamond_M g\right) \diamond_M h = f \diamond_M \left(g \diamond_M h\right) \quad .$$

Now the two sets of proofs can be replaced by a single set of proofs formulated for an "M**-filterable**" functor F, where M could be later set to the identity functor or the Opt functor.

Not all functors M support the Kleisli composition $\diamond_M$ with the required laws. We will study such functors M, which are known as **monads**, in Chapter 10.

9.4.3 Motivation for using category theory

In this chapter, we have seen four examples of operations that have the form of a "lifting":

$$\text{for functors } F: \quad \text{fmap}_F : (A \to B) \to (F^A \to F^B) \quad ,$$
$$\text{for contrafunctors } C: \quad \text{cmap}_C : (B \to A) \to (C^A \to C^B) \quad ,$$
$$\text{for } M\text{-filterable functors } F: \quad \text{lift}_{M,F} : (A \to M^B) \to (F^A \to F^B) \quad ,$$
$$\text{for } M\text{-filterable contrafunctors } C: \quad \text{lift}_{M,F} : (B \to M^A) \to (C^A \to C^B) \quad .$$

All these operations obey similar laws of naturality, identity, and composition but differ in the type of functions being lifted: the ordinary function $A \to B$, the "reversed" type $B \to A$, the Kleisli function $A \to M^B$, and the "reversed" Kleisli function $B \to M^A$. In turn, all these function types obey the laws of identity and composition. (For the types to match, composition of reversed functions needs to be performed in the reverse order.)

In order to avoid writing essentially the same proofs multiple times, we use a more abstract view of this situation: a new notion of a functor that can work with modified function types (e.g., $A \to M^B$ or $F^A \to F^B$) instead of the ordinary function types ($A \to B$). This notion of functor is provided by **category theory**, which turns out to be a convenient language for describing various laws and types of operations in functional programming.

Category theory generalizes functions of type $A \to B$ to **morphisms** $A \rightsquigarrow B$, which can be in any relation to A and B as long as the identity and composition laws hold. The morphism types, the "identity morphism" of type $A \rightsquigarrow A$, and the composition operation must be chosen appropriately. These choices, together with a definition of the admissible types $A, B, ...$, define a **category**, e.g.:

Category	**Types** $A, B, ...$	**Morphisms** $f : A \rightsquigarrow B$	**Identity morphism**	**Composition**
"plain"	Int, String, ...	$f : A \to B$	$\text{id}^{:A\to A}$	$f \fatsemi g$
"reversed"	Int, String, ...	$f : B \to A$	$\text{id}^{:A\to A}$	$g \fatsemi f$
"M-Kleisli"	Int, String, ...	$f : A \to M^B$	$\text{pu}_M : A \to M^A$	$f \diamond_M g$
"reversed M-Kleisli"	Int, String, ...	$f : B \to M^A$	$\text{pu}_M : A \to M^A$	$g \diamond_M f$
"F-lifted"	$F^{\text{Int}}, F^{\text{String}}, ...$	$f : F^A \to F^B$	$\text{id}^{:F^A\to F^A}$	$f \fatsemi g$

Definition 9.4.3.1 (category) A category C is a collection of **objects**[3] $\{A, B, ...\}$ and a collection of morphisms $\{f_1, f_2, ...\}$, where each morphism is labeled by two objects A, B as $f^{:A\rightsquigarrow B}$ (the objects A and B do not need to be different). For any object A, the identity morphism $\text{id}_C^{:A\rightsquigarrow A}$ must exist. For any two morphisms $f^{:A\rightsquigarrow B}$ and $g^{:B\rightsquigarrow C}$, the composition $f \fatsemi_C g$ must exist as a morphism labeled $A \rightsquigarrow C$. Additionally, the identity and the associativity laws must hold:

$$f \fatsemi_C \text{id}_C = f = f \fatsemi_C \text{id}_C \quad , \qquad (f \fatsemi_C g) \fatsemi_C h = f \fatsemi_C (g \fatsemi_C h) \quad .$$

The category laws clearly hold for the "plain" category whose objects are types (`String`, `Option[Int]`, etc.) and morphisms are functions available in the programming language. The F-lifted category is a subset of the "plain" category where we only allow types of the form F^A; so the category laws also hold. Most applications of category theory to functional programming will use categories whose objects are types (or, sometimes, type constructors), and whose morphisms are functions of specific types, e.g., $A \to M^B$ or $F^A \to F^B$ as we have seen. However, the definition of category is general and does not require that morphisms be functions or that objects be types.

In functional programming, a "functor" is a type constructor $F^\bullet$ with a lawful lifting of functions $A \to B$ to functions $F^A \to F^B$. Category theory defines a functor more generally — as a lawful lifting *from one category to another*. We will use the phrase "**categorical functor**" to distinguish the notion of functor in category theory from the programmer's notion (a type constructor with `map`.[4])

Definition 9.4.3.2 (categorical functor) Given two categories C and $\mathcal{D}$, a functor $\mathcal{F} : C \to \mathcal{D}$ is a mapping of each type A in C to the corresponding type $\mathcal{F}(A)$ in $\mathcal{D}$, as well as a mapping of each morphism $f : A \rightsquigarrow_C B$ from C to a corresponding morphism $\mathcal{F}(f) : \mathcal{F}(A) \rightsquigarrow_\mathcal{D} \mathcal{F}(B)$ in $\mathcal{D}$. Additionally, the laws of identity and composition must hold according to the rules of each category. That is, identity morphisms id_C from the category C must be mapped to those of the category $\mathcal{D}$; and the composition ($\fatsemi_C$) of any two morphisms in the category C must be mapped to the composition

[3]The "objects" in category theory are *not* related to "object-oriented programming".

[4]In category theory, programmer's functors are called **endofunctors** — categorical functors from a category C to itself.

$(\fatsemi_{\mathcal{D}})$ of morphisms in the category $\mathcal{D}$.

$$\text{identity law of categorical functor}: \quad \mathcal{F}(\text{id}_C^{:A\rightsquigarrow_C A}) = \text{id}_{\mathcal{D}}^{:\mathcal{F}(A)\rightsquigarrow_{\mathcal{D}}\mathcal{F}(A)} \quad ,$$
$$\text{composition law of categorical functor}: \quad \mathcal{F}(f^{:A\rightsquigarrow_C B})\fatsemi_{\mathcal{D}}\mathcal{F}(g^{:B\rightsquigarrow_C C}) = \mathcal{F}(f\fatsemi_C g) \quad .$$

To clarify this definition, consider two examples: a contrafunctor C and a filterable functor F.

Example 9.4.3.3 A contrafunctor C is specified via a lawful function $\text{cmap} : (B \to A) \to C^A \to C^B$. To formulate this in category theory, we say that there must exist a categorical functor from the reversed category to the C-lifted category. To define that functor, we need to specify how types and morphisms are mapped from the first category to the second. Each type A of the reversed category is mapped to the corresponding type C^A of the C-lifted category. Each $(A \rightsquigarrow B)$-morphism $f^{:B\to A}$ of the reversed category is mapped to the morphism $f^{\downarrow C} : C^A \to C^B$ of the C-lifted category.

To formulate the laws of identity and composition for the (categorical) functor, we look up the definitions of the identity morphisms and the composition operation in each category:

$$\text{for the reversed category}: \quad \text{id}^{:A\to A} \quad \text{and} \quad g\fatsemi f \quad ,$$
$$\text{for the } C\text{-lifted category}: \quad \text{id}^{:C^A\to C^A} \quad \text{and} \quad f\fatsemi g \quad .$$

Now we require that the first category's identity morphism is mapped to the second category's identity morphism, and that a composition of any two morphisms (as defined in the first category) is mapped to a composition as defined in the second category:

$$\text{identity law}: \quad \left(\text{id}^{:A\to A}\right)^{\downarrow C} = \text{id}^{:C^A\to C^A} \quad ,$$
$$\text{composition law}: \quad (g\fatsemi f)^{\downarrow C} = f^{\downarrow C}\fatsemi g^{\downarrow C} \quad .$$

We derived these laws previously as the laws of contrafunctors. □

Example 9.4.3.4 A filterable functor F is specified via a lawful function $\text{liftOpt}_F : (A \to \text{Opt}^B) \to F^A \to F^B$. To formulate the categorical definition for this situation, we say that there must exist a functor from the Opt-Kleisli category to the F-lifted category.

To define that (categorical) functor, we need to specify how types and morphisms are mapped from the first category to the second. Each type A of the Opt-Kleisli category is mapped to the corresponding type F^A of the F-lifted category. Each $(A \rightsquigarrow B)$-morphism $f^{:A\to\mathbb{1}+B}$ of the Opt-Kleisli category is mapped to the morphism $\text{liftOpt}_F(f) : F^A \to F^B$ of the F-lifted category.

Formulate the laws of identity and composition for the (categorical) functor using the definitions of the identity morphisms and the composition operation in each category:

$$\text{for the Opt-Kleisli category}: \quad \text{pu}_{\text{Opt}}^{:A\to\mathbb{1}+A} \quad \text{and} \quad f\diamond_{\text{Opt}} g \quad ,$$
$$\text{for the } F\text{-lifted category}: \quad \text{id}^{:F^A\to F^A} \quad \text{and} \quad f\fatsemi g \quad .$$

Now we require that the first category's identity morphism is mapped to the second category's identity morphism, and that a composition of any two morphisms (as defined in the first category) is mapped to a composition as defined in the second category:

$$\text{identity law}: \quad \text{liftOpt}_F(\text{pu}_{\text{Opt}}) = \text{id}^{:F^A\to F^A} \quad ,$$
$$\text{composition law}: \quad \text{liftOpt}_F(f\diamond_{\text{Opt}} g) = \text{liftOpt}_F(f)\fatsemi\text{liftOpt}_F(g) \quad .$$

We derived these laws previously as the laws of `liftOpt`. □

In both examples, the laws of the suitably defined categorical functors turned out to be equivalent to the typeclass laws we derived previously. This gives us assurance that we have correctly guessed the relevant laws. The choice of typeclass laws is not self-evident. For example, Section 9.1.2 derived the four laws of the `filter` function from heuristic ideas. The laws of `filter` have been formulated

differently by different people, also starting from heuristic considerations.[5] It is not obvious whether we correctly guessed the relevant laws of `filter` and did not assume more laws than necessary. In contrast, the two laws of the categorical functor are general and appear time and again in different areas of mathematics. This gives us confidence that these laws are correctly chosen and will be useful in a wide range of contexts. Proving that the four laws of `filter` from Section 9.1.2 are equivalent to the two laws of a categorical functor gives assurance that our choice of filtering laws is mathematically consistent and is likely to prove useful in applications.

Having looked at the laws of `liftOpt`, we noticed that we can reduce the number of different proofs if we generalize the Opt-Kleisli category to an M-Kleisli category with a suitable functor M. It turns out that one can prove general theorems about products, co-products, and composition of (categorical) functors that map between any categories. In this way, we can replace four theorems (say, for the product of functors, contrafunctors, filterable functors, and filterable contrafunctors) by a single but more abstract theorem about the product of (categorical) functors being a functor between suitably defined categories. We will not look at these proofs here; Chapters 6–9 already worked through a few almost identical proofs that show the required techniques.

The categorical view also shows us two directions for developing the theory further, hoping to find useful applications. First, we can look for functors M (called "monads") that admit the Kleisli composition with the properties (the identity and the associativity laws) required by an M-Kleisli category. Second, having found some new monads M, we can look for "M-filterable" functors or contrafunctors F that admit an operation $\text{lift}_{M,F}$ similar to liftOpt_F but adapted to the monad M instead of `Option`. We will see some examples of M-filterable contrafunctors later in this book.

To summarize, using the category theory's notion of functor brings the following advantages:

- We are assured that we found a correct set of laws of a typeclass. We can derive the formulation of those laws from the standard laws of categories and functors, without guessing.
- We may find some promising directions for obtaining more general type constructions.
- Several proofs may be replaced by a single proof for properties of some (categorical) functors.
- We can formulate general constructions (e.g., functor product) that work in the same way for many different typeclasses.

In this way, we find that category theory is a useful tool for reasoning about abstract constructions that work with different typeclasses (functor, contrafunctor, filterable, etc.). Category theory views many typeclasses in a similar way and gives a systematic guidance for deriving the typeclass laws.

What does category theory (CT) *not* do for functional programming?

- CT defines many abstract constructions but does not say which of these constructions will have useful applications in practical programming.
- For verifying specific laws of specific functions (e.g., the laws of the `Filterable` typeclass for the `takeWhile` function), CT gives neither proofs nor proof techniques that programmers could use.
- CT does not help determine whether a given type constructor (say, `type F[A] = Option[(A, A)]`) will belong to a specific typeclass (e.g., a filterable functor, a pointed functor, or a monad).
- Even if we know that, say, a lawful `Filterable` instance actually exists for the type constructor `F[A] = Option[(A, A)]`, CT does not help in writing correct code for that typeclass instance.
- CT does not say whether there exists a natural transformation between two given type constructors, or how to implement one if it exists and how to verify the suitable naturality laws.

Performing these tasks requires certain techniques of symbolic derivation adapted to *applied* (that is, practically relevant) functional programming. Developing such techniques and selecting the necessary theoretical material is one of the main themes of this book.

[5] E.g., with an additional "empty-value" law here: https://github.com/fantasyland/fantasy-land#filterable

10 Computations in functor blocks. II. Semimonads and monads

In a Scala programmer's view, functors are type constructors with a `map` method, filterable type constructors have a `filter` method, and **semimonads** are functors that have a `flatMap` method. This chapter begins by developing an intuition for the behavior of `flatMap`.

10.1 Practical use of monads

10.1.1 Motivation for semimonads: Nested iteration

How can we translate into code a computation that contains nested iterations, such as:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\frac{1}{1+i+j+k}=? \tag{10.1}$$

Recall that a "flat" (non-nested) iteration is translated into the `map` method applied to a sequence:

```
(1 to n).map { i => 1.0 / (1 + i) }.sum
```

$$\sum_{i=1}^{n}\frac{1}{1+i} \quad .$$

The mathematical notation combines summation and iteration in one symbol, but the code separates these two steps: first, a sequence is computed as `(1 to n).map { i => 1.0 / (1 + i) }`, holding the values $\frac{1}{1+i}$ for $i=1,...,n$, and only then the `sum` function is applied to the sequence. This separation is useful because it gives us full flexibility to transform or aggregate the sequence.

So, we will treat nested iterations in a similar way: first, compute a sequence of values that result from nested iterations, and then apply transformations or aggregations to that sequence.

If we use nested `map` operations, we will obtain a nested data structure, e.g., a vector of vectors:

```
scala> (1 to 5).map(i => (1 to i).map(j => i * j))
res0: IndexedSeq[IndexedSeq[Int]] = Vector(Vector(1), Vector(2, 4), Vector(3, 6, 9), Vector(4, 8,
    12, 16), Vector(5, 10, 15, 20, 25))
```

We need to "flatten" this nested structure into a simple, non-nested sequence. The standard method for that is `flatten`, and its combination with `map` can be replaced by `flatMap`:

```
scala> (1 to 4).map(i => (1 to i).map(j => i * j)).flatten
res1: IndexedSeq[Int] = Vector(1, 2, 4, 3, 6, 9, 4, 8, 12, 16)

scala> (1 to 4).flatMap(i => (1 to i).map(j => i * j))     // Same result as above.
res2: IndexedSeq[Int] = Vector(1, 2, 4, 3, 6, 9, 4, 8, 12, 16)
```

To represent more nesting, we use more `flatMap` operations. For example, to implement Eq. (10.1):

```
def example(n: Int): Double = (1 to n).flatMap { i =>
  (1 to n).flatMap { j =>
    (1 to n).map { k =>
      1.0 / (1.0 + i + j + k) }
  }
}.sum

scala> example(10)
res3: Double = 63.20950497687006
```

These examples show that converting nested iterations into a simple iteration means replacing all `map` functions by `flatMap` except for the last `map` call (which by itself produces a non-nested sequence).

The `for`/`yield` syntax (or "functor block") is an easier way of flattening nested iterations: just use a new source line for each level of nesting. Compare the two following code fragments line by line:

```
(for { i <- 1 to n
    j <- 1 to n
    k <- 1 to n
  } yield 1.0 / (1.0 + i + j + k)
).sum
```

```
(1 to n).flatMap { i =>
   (1 to n).flatMap { j =>
     (1 to n).map { k =>
       1.0 / (1.0 + i + j + k)
  }}}.sum
```

The left arrows visually suggest that the variables `i`, `j`, `k` will iterate over the given sequences. All left arrows except the last one are replaced by `flatMap`s; the last left arrow is replaced by a `map`. These replacements are performed automatically by the Scala compiler.

A functor block with source lines and conditionals corresponds to the mathematical notation for creating sets of values. An example of using that notation is the formula:

```
val t = for {
  x <- p
  y <- q
  z <- r
  if f(x, y, z) == 0
} yield x + y + z
```

$$T = \{x + y + z \mid x \in P,\ y \in Q,\ z \in R,\ f(x, y, z) = 0\} \quad .$$

Here, P, Q, R are given sets of numbers, and the result is a set T of numbers obtained by adding some x from P, some y from Q, and some z from R such that $f(x, y, z) = 0$. A direct implementation of this formula is the code shown at left. Here, `p`, `q`, `r` are given collections (say, arrays) and the result `t` is again an array. Just like the mathematical formula's result is a collection of some $x + y + z$ values, the functor block's result is a collection of values computed after the `yield` keyword.

To develop more intuition about using functor blocks with multiple left arrows, look at this code:

```
val result = for {
  i <- 1 to m
  j <- 1 to n
  x = f(i, j)
  k <- 1 to p
  y = g(i, j, k)
} yield h(x, y)
```

```
val result = {
  (1 to m).flatMap { i =>
    (1 to n).flatMap { j =>
      val x = f(i, j)
      (1 to p).map { k =>
        val y = g(i, j, k)
        h(x, y)  } } } }
```

One can imagine that each line (which we can read as "for all i in $[1, ..., m]$", "for all j in $[1, ..., n]$", etc.) will produce an intermediate sequence of the same type. Each next line continues the calculation from the previous intermediate sequence.

If this intuition is correct, we should be able to refactor the code by cutting the calculation at any place and continuing in another functor block, without changing the result value:

```
val result = for {
  i <- 1 to m
  j <- 1 to n
// We will cut the block here, making i and j
// available for further computations.
  x = f(i, j)
  k <- 1 to p
  y = g(i, j, k)
} yield h(x, y)
// The ‘result‘ is equal to ‘res2‘ at right.
```

```
val res1 = for {
  i <- 1 to m
  j <- 1 to n
} yield (i, j) // Intermediate sequence ‘res1‘.
val res2 = for {
  (i, j) <- res1   // Continue from ‘res1‘.
  x = f(i, j)
  k <- 1 to p
  y = g(i, j, k)
} yield h(x, y)
```

This example illustrates the two features of functor blocks that often cause confusion:

- Each "source line" computes an intermediate collection of the same type, so all values to the right of `<-` must use *the same* type constructor (or its subtypes).
- The entire functor block's result is again a collection using the same type constructor. The result is *not* the expression under `yield`; instead, it is a collection of those expressions.

So far, we have been using sequences as the main type constructor. However, functor blocks with several left arrows will work with any other type constructor that has `map` and `flatMap` methods. In the next sections, we will see how to use functor blocks with different type constructors.

Functors having `flatMap` methods are called **semimonads** in this book.[1] In practice, most semimonads also have a `pure` method (i.e., belong to the `Pointed` typeclass, see Section 8.3.5). Semimonads with a `pure` method (and obeying the appropriate laws) are called **monads**. This chapter will study semimonads and monads in detail. For now, we note that the functor block syntax does not require functors to have a `pure` method; it works just as well with semimonads.

If a functor has a `withFilter` method, Scala's functor block will also support the `if` operation (see Section 9.1). So, the full functionality of functor blocks can be used with *filterable semimonads*.

10.1.2 List-like monads

List-like monads are types that model a collection of data values. Examples are `Seq` and its subtypes, `Stream`, `Array`, non-empty lists, sets, and dictionaries. These data types make different choices of lazy or eager evaluation and memory allocation, but their `flatMap` methods work similarly: they "flatten" nested collections.

All list-like monads are also filterable. A typical task for list-like monads is to find a set of solutions of a combinatorial problem by first taking all combinations and then filtering out unwanted ones.

Example 10.1.2.1 Compute all permutations of the three letters `"a"`, `"b"`, `"c"`.

Solution We will compute a *sequence* of all permutations by nested iteration. First attempt:

```
scala> for {
         x <- Seq("a", "b", "c")
         y <- Seq("a", "b", "c")
         z <- Seq("a", "b", "c")
       } yield x + y + z
res0: Seq[String] = List(aaa, aab, aac, aba, abb, abc, aca, acb, acc, baa, bab, bac, bba, bbb, bbc,
     bca, bcb, bcc, caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc)
```

To obtain all permutations and nothing else, we need to exclude all repeated subsequences such as `"aab"`. To achieve that, we must make `y` iterate over letters that do not include the current value of `x`:

```
val xs = Seq("a", "b", "c")

scala> for {
      x <- xs
      xsWithoutX = xs.filter(_ != x)
      y <- xsWithoutX
      xsWithoutXY = xsWithoutX.filter(_ != y)
      z <- xsWithoutXY
} yield x + y + z
res1: Seq[String] = List(abc, acb, bac, bca, cab, cba)
```

Example 10.1.2.2 Compute the set of all subsets of `xs = Set("a", "b", "c")`.

Solution We aim to write the code as a nested iteration. Begin by choosing one element, say `"a"`. Some subsets of `xs` will contain `"a"` and other subsets will not. So, let `xa` iterate over two possibilities: either an empty set, `Set()`, or a set containing just `"a"`. Then each subset of `xs` is the union of `xa` and some subset that does not contain `"a"`. We repeat the same logic for `"b"` and `"c"`. The code is:

```
val empty = Set[String]()

scala> for {
         xa <- Set(empty, Set("a"))
         xb <- Set(empty, Set("b"))
         xc <- Set(empty, Set("c"))
       } yield xa ++ xb ++ xc        // p ++ q is the union of the sets p and q.
```

[1] There is no single accepted name. The libraries `scalaz` and `cats` call the relevant typeclasses `Bind` and `FlatMap` respectively.

```
res0: Set[Set[String]] = Set(Set(), Set(a, b), Set(b, c), Set(a, c), Set(a, b, c), Set(b), Set(c),
    Set(a))
```

Example 10.1.2.3 Compute all sub-arrays of length 3 in a given array. Type signature and a test:

```
def subarrays3[A](input: IndexedSeq[A]): IndexedSeq[IndexedSeq[A]] = ???

scala> subarrays3(IndexedSeq("a", "b", "c", "d"))
res3: IndexedSeq[IndexedSeq[String]] = Vector(Vector(a, b, c), Vector(a, b, d), Vector(a, c, d),
    Vector(b, c, d))
```

Solution What are the indices of the elements of the required sub-arrays? Suppose n is the length of the given array. In any sub-array of length 3, the first element must have an index i such that $0 \leq i < n$. The second element must have an index j such that $i < j < n$, and the third element's index k must satisfy $j < k < n$. So we can iterate over the indices like this:

```
def subarrays3[A](input: IndexedSeq[A]): IndexedSeq[IndexedSeq[A]] = {
  val n = input.length
  for {
    i <- 0 until n                    // Iterate over 0,1,...,n−1.
    j <- i + 1 until n
    k <- j + 1 until n
  } yield IndexedSeq(input(i), input(j), input(k))
}
```

Example 10.1.2.4 Generalize examples 10.1.2.1–10.1.2.3 to support an arbitrary size n instead of 3.

Solution **(a)** The task is to compute the set of all permutations of n letters. We note that the solution in Example 10.1.2.1 used three source lines (such as `x <- xs`), one for each letter. To generalize that code to n letters, we would need to write a functor block with n source lines. However, we cannot do that if n is a run-time parameter not fixed in advance. So, the functor block must use recursion in n. Begin implementing `permutations(xs)` as a recursive function:

```
def permutations(xs: Seq[String]): Seq[String] = for {
  x <- xs
  xsWithoutX = xs.filter(_ != x)
  ??? permutations(xsWithoutX) ??? // We need to use a recursive call somehow.
```

It is promising to use a recursive call of `permutations` with the sub-sequence `xsWithoutX` that does not contain a chosen letter `x`. It remains to formulate the code as nested iteration. Let us visualize the computation for `xs == Seq("a","b","c")`. While iterating over `xs`, we start with `x == "a"`, which gives `xsWithoutX == Seq("b","c")`. Iterating over `permutations(xsWithoutX)`, we obtain the permutations `"bc"` and `"cb"`. These permutations need to be concatenated with `x == "a"`, yielding `"abc"` and `"acb"`, which is the correct part of the final answer. So, we write a nested iteration and concatenate the results:

```
def permutations(xs: Seq[String]): Seq[String] = for {
  x <- xs
  xsWithoutX = xs.filter(_ != x)
  rest <- permutations(xsWithoutX)
} yield x + rest

scala> permutations(Seq("a", "b", "c", "d"))
res0: Seq[String] = List()
```

The code is wrong: it always returns an empty list! The reason is that we provided no base case for the recursion. Eventually the intermediate value `xsWithoutX` becomes empty. A nested iteration with an empty list always makes the final result also an empty list. To fix this, add a base case:

```
def permutations(xs: Seq[String]): Seq[String] = if (xs.length == 1) xs else for {
  x <- xs
  xsWithoutX = xs.filter(_ != x)
  rest <- permutations(xsWithoutX)
} yield x + rest
```

```
scala> permutations(Seq("a", "b", "c", "d"))
res1: Seq[String] = List(abcd, abdc, acbd, acdb, adbc, adcb, bacd, badc, bcad, bcda, bdac, bdca,
     cabd, cadb, cbad, cbda, cdab, cdba, dabc, dacb, dbac, dbca, dcab, dcba)
```

(b) To find all subsets of a set via nested iteration, we cannot directly extend the code from Example 10.1.2.2 because we cannot write an unknown number of source lines in a functor block. As in part **(a)**, we need to refactor the code and write a functor block that uses recursion:

```
def subsets[A](xs: Set[A]): Set[Set[A]] = for {
  x <- xs
  xsWithoutX = xs - x           // Use the difference operation for sets.
  ??? subsets(xsWithoutX) ???   // We need to use a recursive call somehow.
```

If `xs == Set("a", "b", "c")` and `x == "a"` during an iteration, we get `xsWithoutX == Set("b", "c")`. Once we compute `subsets(xsWithoutX)`, we need to use all those subsets together with `x` and also without adding `x`. We also should not forget to write the base case (an empty set `xs`). So, the code becomes:

```
def subsets[A](xs: Set[A]): Set[Set[A]] = if (xs.isEmpty) Set(xs) else for {
  x <- xs
  xsWithoutX = xs - x
  rest <- subsets(xsWithoutX)   // Recursive call.
  maybeX <- Set(Set(x), Set())
} yield maybeX ++ rest

scala> subsets(Set("a", "b", "c", "d"))
res0: Set[Set[String]] = Set(Set(), Set(a, c, d), Set(a, b), Set(b, c), Set(a, d), Set(a, b, d),
     Set(b, c, d), Set(a, c), Set(c, d), Set(a, b, c), Set(d), Set(b), Set(b, d), Set(a, b, c, d),
     Set(c), Set(a))
```

(c) To compute all sub-arrays of length n from a given array, we rewrite the solution in Example 10.1.2.3 via a recursive function that computes the sequence of indices:

```
def subindices(begin: Int, end: Int, n: Int): IndexedSeq[IndexedSeq[Int]] =
  if (n == 0) IndexedSeq(IndexedSeq()) else for {
    i <- begin until end
    rest <- subindices(i + 1, end, n - 1)   // Recursive call.
  } yield IndexedSeq(i) ++ rest

scala> subindices(0, 4, 2)
res0: IndexedSeq[IndexedSeq[Int]] = Vector(Vector(0, 1), Vector(0, 2), Vector(0, 3), Vector(0, 4),
     Vector(1, 2), Vector(1, 3), Vector(1, 4), Vector(2, 3), Vector(2, 4), Vector(3, 4))
```

The sequence of subarrays is easy to compute once the sequence of subarray indices is known:

```
def subarrays[A](n: Int, input: IndexedSeq[A]): IndexedSeq[IndexedSeq[A]] =
  subindices(0, input.length, n).map(_.map(i => input(i)))

scala> subarrays(4, IndexedSeq("a", "b", "c", "d", "e"))
res1: IndexedSeq[IndexedSeq[String]] = Vector(Vector(a, b, c, d), Vector(a, b, c, e), Vector(a, b,
     d, e), Vector(a, c, d, e), Vector(b, c, d, e))
```

The solutions **(a)–(c)** are not tail recursive because recursive calls within a source line in a functor block are translated into recursive calls not in tail positions but *inside* `map` or `flatMap` methods. Achieving tail recursion in functor blocks requires techniques beyond the scope of this chapter.

Example 10.1.2.5 Find all solutions of the "8 queens" problem.

Solution The 8 queens must be placed on an 8×8 chess board so that no queen threatens any other queen. To make our work easier, we note that each queen must be placed in a different row. So, it is sufficient to find the column index for each queen. A solution is a sequence of 8 indices.

Begin by iterating over all possible combinations of column indices:

```
val solutions = for {
  x0 <- 0 until 8              // Queen 0 has coordinates (x0, 0).
  x1 <- 0 until 8              // Queen 1 has coordinates (x1, 1).
  x2 <- 0 until 8              // Queen 2 has coordinates (x2, 2).
```

It remains to filter out invalid positions. We should start filtering as early as possible, since the total number of combinations grows quickly during nested iterations:

```
val solutions = for {
  x0 <- 0 until 8              // Queen 0 has coordinates (x0, 0).
  x1 <- 0 until 8              // Queen 1 has coordinates (x1, 1).
  if noThreat(x1, Seq(x0))     // Queen 1 does not threaten queen 0.
  x2 <- 0 until 8              // Queen 2 has coordinates (x2, 2).
  if noThreat(x2, Seq(x0, x1)) // Queen 2 does not threaten queens 0 and 1.
  ...
} yield Seq(x0, x1, x2, ...)
```

Here, `noThreat` is a helper function that decides whether a new queen threatens previous ones:

```
def noThreat(otherX: Int, prev: Seq[Int]): Boolean = {
  val otherY = prev.length
  prev.zipWithIndex.forall { case (x, y) =>   // Check the vertical and the two diagonals.
    x != otherX && x - y != otherX - otherY && x + y != otherX + otherY
  }
}
```

We used a feature of Scala allowing us to pass a sequence of arguments via the syntax `Int*`, which means a variable number of arguments of type `Int`. We can now complete the code and test it:

```
val column = 0 until 8
val solutions = for {
  x0 <- column                 // Queen 0 has coordinates (x0, 0).
  x1 <- column                 // Queen 1 has coordinates (x1, 1).
  if noThreat(x1, Seq(x0))     // Queen 1 does not threaten queen 0.
  x2 <- column                 // Queen 2 has coordinates (x2, 2).
  if noThreat(x2, Seq(x0, x1)) // Queen 2 does not threaten queens 0 and 1.
  x3 <- column
  if noThreat(x3, Seq(x0, x1, x2))
  x4 <- column
  if noThreat(x4, Seq(x0, x1, x2, x3))
  x5 <- column
  if noThreat(x5, Seq(x0, x1, x2, x3, x4))
  x6 <- column
  if noThreat(x6, Seq(x0, x1, x2, x3, x4, x5))
  x7 <- column
  if noThreat(x7, Seq(x0, x1, x2, x3, x4, x5, x6))
} yield Seq(x0, x1, x2, x3, x4, x5, x6, x7)

scala> solutions.take(5)    // First 5 solutions.
res0: IndexedSeq[Seq[Int]] = Vector(List(0, 4, 7, 5, 2, 6, 1, 3), List(0, 5, 7, 2, 6, 3, 1, 4),
     List(0, 6, 3, 5, 7, 1, 4, 2), List(0, 6, 4, 7, 1, 3, 5, 2), List(1, 3, 5, 7, 2, 0, 6, 4))
```

Example 10.1.2.6 Generalize Example 10.1.2.5 to solve the "n queens" problem.

Solution In this problem, n queens must be placed on an $n \times n$ chess board. We need to find and count all solutions.[2] As in the 8-queens problem, each queen must be placed in a different row, and so we represent a solution by a sequence of n column indices, each index between 0 and $n - 1$.

Begin by writing code that expects to use itself recursively:

```
def nQueens(n: Int): Seq[Seq[Int]] = for {
  x0 <- 0 until n
  ??? nQueens(n - 1) ???
```

Possible positions of new queens depend on the chosen positions for all previous queens. So, the recursive function must receive that information. Write an auxiliary recursive function `nQueensPartial` that computes all remaining positions given a sequence of (less than n) previously found queens:

```
def nQueensPartial(n: Int, prev: Seq[Int]): Seq[Seq[Int]] = for {
```

[2] There is no known general formula for the number of solutions of the n-queens problem. See a discussion here, https://math.stackexchange.com/questions/1872444/

```
  x <- 0 until n
  if noThreat(x, prev)          // The new queen does not threaten any previous queens.
  rest <- nQueensPartial(n - 1, prev +: x)    // Find positions with n - 1 new queens.
} yield x +: rest                // Prepend the new queen to the other queen positions.
```

This code still has two problems: first, the base case ($n = 0$) is not covered; second, the recursive function must be initially called with correct arguments. The complete code is:

```
def nQueens(n: Int): Seq[Seq[Int]] = {
    def nQueensPartial(m: Int, prev: Seq[Int]): Seq[Seq[Int]] = if (m == 0) Seq(Seq()) else for {
        x <- 0 until n
        if noThreat(x, prev)
        rest <- nQueensPartial(m - 1, prev :+ x)
      } yield x +: rest
    nQueensPartial(n, Seq())
}

scala> (nQueens(8).length, nQueens(9).length, nQueens(10).length, nQueens(11).length)
res0: (Int, Int, Int, Int) = (92,352,724,2680)
```

Example 10.1.2.7* Formulas of Boolean logic may be transformed by "expanding brackets",

$$\text{expand brackets with a conjunction inside}: \quad (a \wedge b) \vee c = (a \vee c) \wedge (b \vee c) \quad ,$$
$$\text{expand brackets with a disjunction inside}: \quad (a \vee b) \wedge c = (a \wedge c) \vee (b \wedge c) \quad .$$

Implication is transformed as $(a \Rightarrow b) = ((\neg a) \vee b)$. These transformations (expressed in Scala at left)

```
(a && b) || c  ==  (a || c) && (b || c)
(a || b) && c  ==  (a && c) || (b && c)
(a implies b)  ==  (!a) || b
```

allow us to rewrite any Boolean formula as a conjunction of disjunctions with no more nested conjunctions inside, e.g., `(a || b) && ((!c) || d || e)`. This form is called the **conjunctive normal form**[3] (CNF) of a Boolean formula. We can also rewrite Boolean formula into a *disjunction* of conjunctions with no more nesting inside, e.g., `(p && q && !r) || ((!x) && y)`. This is called the **disjunctive normal form**[4] (DNF) of a Boolean formula. The task is to implement functions that convert formulas from CNF to DNF and back.

Solution We begin by designing a data type to represent CNFs. Let the type parameter A stand for the type of elementary Boolean formulas that we denoted by `a`, `b`, `!c`, ..., i.e., Boolean formulas that contain no conjunctions or no disjunctions. Then we may represent a conjunction as a `Set[A]`, and a disjunction of conjunctions as a `Set[Set[A]]`. For instance, `(a || b) && (c || d || e)` is represented by `Set(Set(a, b), Set(c, d, e))`. Because we are using `Set`, disjunction of two disjunctions is easily implemented as a union of sets. Since `Set` eliminates repeated elements, the representation based on `Set` will also automatically perform simplification of formulas such as `(a || b || a) == (a || b)`.

Conjunction of two conjunctions may be implemented as a union of sets for the same reason.

We also need to figure out how to represent the Boolean constants `true` and `false` in the CNF. When we compute a conjunction such as `(a || b) && c && true`, the result must be just `(a || b) && c`. Since `(a || b) && c` is represented as `Set(Set(a, b), Set(c))`, and since we compute conjunctions via unions of sets, we expect to have the equation:

```
Set(Set(a, b), Set(c)) union Set(???) == Set(Set(a, b), Set(c))
```

This must hold for any formulas; so we must represent `true` as an *empty* set of disjunctions, `Set()`.

```
final case class CNF[A](s: Set[Set[A]])
def trueCNF[A]     = CNF[A](Set())
def falseCNF[A]    = CNF[A](Set(Set()))
```

The value `false` should have the property that for any group g of disjunctions, e.g., `g == (x || y)`, we must have `g || false == g`. So, `false` corresponds to an empty disjunction. However, CNF represents disjunctions by nested sets contained within conjunctions. Thus, `Set(Set())` must stand for `false`.

[3] https://en.wikipedia.org/wiki/Conjunctive_normal_form
[4] https://en.wikipedia.org/wiki/Disjunctive_normal_form

Similar arguments hold for representing DNFs. We again use nested sets, but this time the conjunctions and disjunctions are swapped, so `Set(Set(a, b), Set(c, d, e))` stands for the DNF formula `(a && b) || (c && d && e)`. The value `Set()` represents an empty disjunction and stands for the constant `false`, while the constant `true` is `Set(Set())`, a disjunction containing an empty conjunction.

```
final case class DNF[A](s: Set[Set[A]])
def trueDNF[A]    = DNF[A](Set(Set()))
def falseDNF[A]   = DNF[A](Set())
```

It is easy to make mistakes when reasoning about the constants `true` and `false` in these representations. We need to test the code to make sure our implementations of `true` and `false` are correct.

Before writing the full code, let us implement the DNF-to-CNF conversion for a short example:

```
     (a && b) || (c && d)         ---->          (a || c) && (a || d) && (b || c) && (b || d)
                                // The same formulas are written in the set-based representation as:
DNF(Set(Set(a, b), Set(c, d)))    ---->      CNF(Set(Set(a, c), Set(a, d), Set(b, c), Set(b, d)))
```

A simple expression implementing this transformation for nested sets could look like this:

```
for {
  x <- Set("a", "b")
  y <- Set("c", "d")
} yield Set(x) ++ Set(y)   // The result is Set(Set(a, c), Set(a, d), Set(b, c), Set(b, d)).
```

Generalizing this code to an arbitrary number of nested sets, we get a first draft of a solution:

```
def toCNF[A](dnf: Set[Set[A]]): Set[Set[A]] = for {
  x  <- dnf.head            // The ‘head‘ of a set; corresponds to Set("a", "b") in the example above.
  ys <- toCNF(dnf.tail)     // The ‘tail‘ of a set; corresponds to Set(Set("c", "d")) in the example.
                            // Converted to CNF, this corresponds to Set(Set("c"), Set("d")).
} yield Set(x) ++ ys
```

This code is not yet fully correct: we have no base case for the recursion, and the `head` method will crash for an empty set. To make the pattern matching on a `Set` easier, let us implement a function:

```
def headTailOption[A](xs: Set[A]): Option[(A, Set[A])] =     // ‘List‘-like pattern matching for Set.
  if (xs.isEmpty) None else Some((xs.head, xs.tail))
```

With that function, we can write the final code as:

```
def toCNF[A](dnf: DNF[A]): CNF[A] = headTailOption(dnf.s) match {
  case None => falseCNF[A]      // Base case: The empty set Set() means ‘false‘ in DNF.
  case Some((head, tail)) => CNF(for {
    x  <- head                  // The first group of conjunctions.
    ys <- toCNF(DNF(tail)).s    // The remaining groups of conjunctions, recursively converted to CNF.
  } yield ys + x)               // For sets, ‘ys + x‘ is the same as ‘Set(x) ++ ys‘.
}                               // Some tests for this code:
val dnf1 = DNF(Set(Set("a", "b"), Set("c", "d", "e")))
val cnf1 = CNF(Set(Set("a", "c"), Set("a", "d"), Set("a", "e"), Set("b", "c"), Set("b", "d"),
     Set("b", "e")))

scala> toCNF(dnf1) == cnf1
res0: Boolean = true

scala> (toCNF(trueDNF) == trueCNF, toCNF(falseDNF) == falseCNF)
res1: (Boolean, Boolean) = (true,true)
```

Because the rules of Boolean logic are completely symmetric with respect to swapping the operations $\vee$ and $\wedge$, the function `cnf2dnf` can be implemented via `dnf2cnf` like this:

```
def toDNF[A](cnf: CNF[A]): DNF[A] = DNF(toCNF(DNF(cnf.s)).s)

scala> (toDNF(trueCNF) == trueDNF, toDNF(falseCNF) == falseDNF)
res2: (Boolean, Boolean) = (true,true)
```

If we test some more, we will find that the functions `toCNF` and `toDNF` are *not* inverses:

```
scala> toDNF(cnf1)     // Expected this to equal DNF(Set(Set("a", "b"), Set("c", "d", "e"))) == dnf1.
res3: DNF[String] = DNF(Set(Set(b, a), Set(b, a, c, e), Set(d, a, b), Set(b, e, a), Set(b, a, c),
```

```
    Set(d, e, c, b), Set(d, e, c), Set(d, a, c, b), Set(e, a, b, c, d), Set(d, a, e, c), Set(d, a,
    e, b)))
```

However, we already verified that `cnf1 == toCNF(dnf1)`. We expect the formula `dnf1` to remain the same when we convert it to CNF and back to DNF. Why the discrepancy? The reason is that the transformations `toDNF` and `toCNF` fail to perform certain simplifications allowed in Boolean logic. For example, both `(a) || (a && b)` and `(a) && (a || b)` are equivalent to just `(a)`. In the set-based representations, we must be able to simplify `Set(Set("a"), Set("a", "b"))` to just `Set(Set("a"))`. Generally, we may discard any conjunction group that happens to be a superset of another conjunction group, and similarly for disjunction groups. After these simplifications, the formula `toDNF(cnf1)` becomes equal to `dnf1` because, as we can see, all inner sets in `toDNF(cnf1)` are supersets of either `Set("a", "b")` or `Set("c", "d", "e")))` or both. We can implement the simplifications as a function `simplifyCNF`:

```
def simplifyCNF[A](cnf: CNF[A]): CNF[A] = CNF(
    cnf.s.toIndexedSeq.sortBy(_.size).foldLeft(Set[Set[A]]()) { case (prevGroups, group) =>
        if (prevGroups.exists(_ subsetOf group)) prevGroups else prevGroups + group
      }    // Omit ‘group‘ if it happens to be a superset of some previous group within ‘prevGroups‘.
)    // Since ‘prevGroups‘ is sorted by group size, we only need to check later groups for supersets.

scala> simplifyCNF(CNF(Set(Set("a", "b"), Set("a", "b", "c", "d"))))
res3: CNF[String] = CNF(Set(Set(a, b)))
```

With this function, we may define more practically useful transformations `dnf2cnf` and `cnf2dnf` as:

```
def dnf2cnf[A](dnf: DNF[A]): CNF[A] = simplifyCNF(toCNF(dnf))
def cnf2dnf[A](cnf: CNF[A]): DNF[A] = DNF(dnf2cnf(DNF(cnf.s)).s)
                                                    // Verify that dnf2cnf and cnf2dnf are inverses:
scala> (dnf2cnf(cnf2dnf(cnf1)) == cnf1, cnf2dnf(dnf2cnf(dnf1)) == dnf1)
res4: (Boolean, Boolean) = (true,true)
```

The new conversion functions `dnf2cnf` and `cnf2dnf` are inverses of each other.

Example 10.1.2.8* (matrix operations) If matrices are represented by nested sequences of numbers, matrix products can be calculated via nested functor blocks. The present task is to implement: **(a)** matrix transposition, **(b)** the vector-matrix dot product, and **(c)** the matrix-matrix dot product.

Solution **(a)** If a matrix has type `Seq[Seq[A]]`, a transposed matrix has the same type. An example:

```
val matrix = Seq(                    val matrix_T = Seq(
    //IGNORETHIS
              Seq(1, 2, 3),                          Seq(1, 10),
              Seq(10, 20, 30)                        Seq(2, 20),
            )                                        Seq(3, 30)
/* Both matrices have type Seq[Seq[Int]]. */        )
```

To compute the transposed matrix, we have to iterate over the initial matrix. A functor block of type `Seq` will return a value of type `Seq[Seq[A]]` only if the `yield` expression itself returns a value of type `Seq[A]`. The example above shows that the first iteration should return `Seq(1, 10)`, which contains the first elements of all inner sequences from `matrix`. The second iteration should return all second elements, and so on. We see that we need to iterate over the indices of the matrix columns. Those indices are returned by the library method `indices`:

```
scala> Seq("Wilhelm Roentgen", "Henri Beckerel", "Marie Curie").indices
res0: scala.collection.immutable.Range = Range(0, 1, 2)
```

The value `Seq(1, 10)` can be computed by the functor block

```
for { row <- matrix } yield row(0)
```

This functor block needs to be written *inside* the `yield` value of the iterations of the indices:

```
scala> val matrix_T = for { index <- matrix.head.indices }    // [0, 1, 2] are the column indices.
                      yield { for { row <- matrix }        // Iterate over the rows of the matrix.
                              yield row(index)
                            }
matrix_T: IndexedSeq[Seq[Int]] = Vector(List(1, 10), List(2, 20), List(3, 30))
```

The two nested `for`/`yield` blocks represent two nested loops whose result value is again a nested

sequence. We have implemented nested loops over column and row indices by nested functor blocks that use the column indices but directly iterate over rows.

(b) To see how a vector-matrix dot product works, consider this example:

$$\begin{vmatrix} a_0 & a_1 & a_2 \end{vmatrix} \cdot \begin{vmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \\ b_{20} & b_{21} \end{vmatrix} = \begin{vmatrix} a_0 \cdot b_{00} + a_1 \cdot b_{10} + a_2 \cdot b_{20} & a_0 \cdot b_{01} + a_1 \cdot b_{11} + a_2 \cdot b_{21} \end{vmatrix} \quad .$$

The 2×3 matrix is represented as a sequence containing three nested sequences. We need to iterate over the first elements of the nested sequences (b_{00}, b_{10}, b_{20}) , multiply them with the corresponding elements (a_0, a_1, a_2) of the vector, and compute the sum of all products. The code is:

```
Seq(b00, b10, b20).zip(Seq(a0, a1, a2)).map { case (x, y) => x * y }.sum
```

Then we need to do the same with the second elements (b_{01}, b_{11}, b_{21}). To obtain the full code, we iterate over the indices of the nested sequences, as we did in part **(a)** for the transposition:

```
import scala.math.Numeric.Implicits.infixNumericOps
def vectorMatrixProduct[N: Numeric](vector: Seq[N], matrix: Seq[Seq[N]]): Seq[N] =
  for { index <- matrix.head.indices } yield {
    val b = for { row <- matrix } yield row(index)
    val pairs = for { (x, y) <- b.zip(vector) } yield x * y
    pairs.sum
  }

scala> vectorMatrixProduct(Seq(3,4,5), matrix_T)
res1: Seq[Int] = Vector(26, 260)
```

(c) The matrix-matrix dot product is defined as a matrix containing the results of the vector-matrix dot products, for all row vectors of the first matrix. For example:

$$\begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{vmatrix} \cdot \begin{vmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \\ b_{20} & b_{21} \end{vmatrix} = \begin{vmatrix} a_{00} \cdot b_{00} + a_{01} \cdot b_{10} + a_{02} \cdot b_{20} & a_{00} \cdot b_{01} + a_{01} \cdot b_{11} + a_{02} \cdot b_{21} \\ a_{10} \cdot b_{00} + a_{11} \cdot b_{10} + a_{12} \cdot b_{20} & a_{10} \cdot b_{01} + a_{11} \cdot b_{11} + a_{12} \cdot b_{21} \end{vmatrix} \quad .$$

We reuse the solution of part **(b)** and write the code as:

```
def matrixProduct[N: Numeric](matrix1: Seq[Seq[N]], matrix2: Seq[Seq[N]]): Seq[Seq[N]] =
  for { row1 <- matrix1 } yield vectorMatrixProduct(row1, matrix2)
```

The code shown in this example is for illustration only; for higher performance, matrix operations must be implemented through flat arrays rather than nested sequences.

Exercise 10.1.2.9 Implement the matrix-vector dot product as a function:

```
def matrixVectorProduct[N: Numeric](matrix: Seq[Seq[N]], vector: Seq[N]): Seq[N] = ???
```

The result of computing the matrix-vector product is a column vector, for example:

$$\begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{vmatrix} \cdot \begin{vmatrix} b_{00} \\ b_{10} \\ b_{20} \end{vmatrix} = \begin{vmatrix} a_{00} \cdot b_{00} + a_{01} \cdot b_{10} + a_{02} \cdot b_{20} \\ a_{10} \cdot b_{00} + a_{11} \cdot b_{10} + a_{12} \cdot b_{20} \end{vmatrix} \quad .$$

Exercise 10.1.2.10 The **trace** of a square matrix is the sum of its diagonal elements, e.g.:

$$\text{Tr} \begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix} \stackrel{\text{def}}{=} a_{00} + a_{11} + a_{22} \quad .$$

Implement the trace of a matrix (assuming a square matrix) as a function:

```
def trace[N: Numeric](matrix: Seq[Seq[N]]): N = ???
```

10.1.3 Pass/fail monads

The type `Option[A]` can be viewed as a collection that can either empty or hold a single value of type `A`. An "iteration" over such a collection will perform a computation at most once:

```
scala> for { x <- Some(123) } yield x * 2     // The computation is performed once.
res0: Option[Int] = Some(246)
```

When an `Option` value is empty, the computation is not performed at all:

```
scala> for { x <- None: Option[Int] } yield x * 2     // The computation is not performed at all.
res1: Option[Int] = None
```

What would a *nested* "iteration" over several `Option` values do? When all of the `Option` values are non-empty, the "iteration" will perform some computations using the wrapped values. However, if even one of the `Option` values happens to be empty, the computed result will be an empty value:

```
scala> for {
         x <- Some(123)
         y <- None
         z <- Some(-1)
       } yield x + y + z
res2: Option[String] = None
```

Computations with `Either` and `Try` values follow the same logic: nested "iteration" will perform no computations unless all values are non-empty. This logic is useful for implementing a series of computations that could produce failures, where any failure should stop all further processing. For this reason (and since they all support the `pure` method and are lawful monads, as this chapter will show), we call the type constructors `Option`, `Either`, and `Try` the **pass/fail monads**.

The following schematic example illustrates this logic:

```
val result: Try[A] = for { // Computations in the ‘Try‘ monad.
  x <- Try(k())            // First computation ‘k()‘, may fail.
  y = f(x)                 // No possibility of failure in this line.
  if p(y)                  // The entire expression will fail if ‘p(y) == false‘.
  z <- Try(g(x, y))        // The computation may also fail here.
  r <- Try(h(x, y, z))     // May fail here as well.
} yield r                  // If ‘r‘ has type ‘A‘ then ‘result‘ has type ‘Try[A]‘.
```

The function `Try()` catches exceptions thrown by its argument. If one of `k()`, `g(x, y)`, or `h(x, y, z)` throws an exception, the corresponding `Try(...)` call will evaluate to a `Failure(...)` case class, and further computations will not be performed. The value `result` will indicate the *first* encountered failure. Only if all `Try(...)` calls evaluate to a `Success(...)` case class, the entire expression evaluates to a `result` of type `Success` that wraps a value of type `A`.

Whenever this pattern of computation is found, a functor block gives concise and readable code that replaces a series of nested `if`/`else` or `match`/`case` expressions. Such a situation was shown in Example 3.2.2.4 (Chapter 3), where a "safe integer" computation continues only as long as every result is a success; the chain of operations stops at the first failure. The code of Example 3.2.2.4 introduced custom data type with hand-coded methods such as `add`, `mul`, and `div`. We can now implement equivalent functionality using functor blocks and a standard type `Either[String, Int]`:

```
type Result = Either[String, Int]
def div(x: Int, y: Int): Result = if (y == 0) Left(s"error: $x / $y") else Right(x / y)
def sqrt(x: Int): Result = if (x < 0) Left(s"error: sqrt($x)") else Right(math.sqrt(x).toInt)
val initial: Result = Right(20)       // Start with a given ‘initial‘ value of type ‘Result‘.

scala> val result: Result = for {     // Safe computation: ‘sqrt(1000 / initial - 100) + 20‘.
  x <- initial
```

```
  y <- div(1000, x)
  z <- sqrt(y - 100)
} yield z + 1
result: Result = Left("error: sqrt(-50)")
```

The concise and readable code of `val result` replaces more verbose implementations such as:

```
val result: Result = previous match {
  case Left(error)   => Left(error)
  case Right(x)      => div(1000, x) match {
    case Left(error)   => Left(error)
    case Right(y)      => sqrt(y - 100) match {
      case Left(error)   => Left(error)
      case Right(z)      => ...  // Keep writing repetitive, deeply nested code of this kind.
```

The following examples illustrate the typical tasks where pass/fail monads are used. These tasks perform a linear sequence of computations that may fail; the first failure is then returned as a value.

Example 10.1.3.1: chaining computations with `Option` Some clients have placed some orders with some companies. The information is made available via Java system properties, for example:

```
System.setProperty("client 0", "company 2")
System.setProperty("client 1", "company 3")
System.setProperty("company 2", "order 4")
System.setProperty("company 3", "order 5")
System.setProperty("order 4", "123")
System.setProperty("order 5", "456")
```

Given a client's name, obtain the corresponding order quantity if it exists.

Solution The Java method `System.getProperty` returns the property value as `String` if the property is present, and otherwise returns `null`. Wrapping that return value into an `Option()` call, we replace null values by empty `Option` values (i.e., `None`). This makes the result values safe: using a `null` value may throw an exception, which will not happen when using `map` and `flatMap` methods on `Option` values. It remains to chain the computations:

```
def getOrderAmount(client: String): Option[Int] =   for {
      company     <- Option(System.getProperty(client))
      order       <- Option(System.getProperty(company))
      stringValue <- Option(System.getProperty(orders))
      intValue    <- Try(stringValue.toInt).toOption   // Non-integer string values are invalid.
    } yield intValue

scala> getOrderAmount("client 1")
res0: Option[Int] = Some(123)

scala> getOrderAmount("client 2")
res1: Option[Int] = Some(456)

scala> getOrderAmount("client 3")
res2: Option[Int] = None
```

In the example just shown, `Option` values are sufficient since the absence of a property is not an error situation. Now we consider a task where we need to keep track of error information.

Example 10.1.3.2: chaining computations with `Try` Three given functions f, g, h all have Scala type `Int => Int` but may throw exceptions. Given an integer x, compute $f(g(h(x)))$ safely, reporting the first encountered error.

Solution Wrap each function into a `Try()` and chain the resulting computations:

```
def f(x: Int): Int = 1 / x
def g(x: Int): Int = 2 - x
def h(x: Int): Int = 2 / x

import scala.util.Try
def result(x: Int): Try[Int] = for {
```

```
      p <- Try(h(x))
      q <- Try(g(p))
      r <- Try(f(q))
    } yield r

scala> result(1)
res0: Try[Int] = Failure(java.lang.ArithmeticException: / by zero)
```

The result value shows information about the failure generated in this computation.

Example 10.1.3.3: chaining with `Future` Scala library's `Future` class can be seen as a pass/fail monad because `Future(x)` will encapsulate any exception thrown while computing `x`. However, in addition to the pass/fail features, a `Future` value has a concurrency effect: the encapsulated computation `x` is scheduled to be run in parallel on another CPU thread. For this reason, `Future`'s methods (such as `map` and `flatMap`) require an implicit `ExecutionContext` argument, which provides access to a JVM thread pool where computations will be scheduled.

As soon as a `Future` value is created, its computation is scheduled immediately. So, several `Future` values may run their computations in parallel. Nevertheless, computations chained via `flatMap` (or in a functor block) will run sequentially if new values need to wait for previous values:

```
import scala.concurrent.ExecutionContext.Implicits.global
def longComputation(x: Double): Future[Double] = Future { ... } // A long computation.

val result1: Future[Double] = for {
  p <- longComputation(10.0)
  q <- longComputation(p + 20.0)
  r <- longComputation(q - 20.0)
} yield p + q + r  // Three ‘longComputation‘ calls are running sequentially.
```

This code waits for the first `Future` that computes `p`, then creates a `Future` value that will eventually compute `q`, and finally creates a `Future` that will eventually compute `r`; only then the sum `p + q + r` may be obtained (wrapped in a `Future` constructor). This computation cannot run the three `longComputation(...)` calls in parallel, since each call depends on the result of the previous one.

Another possibility is that each `longComputation(...)` is independent of the results of the other computations. Then the three `Future` values may be created up front, and the functor block code represents three "long computations" running in parallel:

```
val long1 = longComputation(10.0)
val long2 = longComputation(50.0)
val long3 = longComputation(100.0)

val result2: Future[Double] = for {
  p <- long1
  q <- long2
  r <- long3
} yield p + q + r  // Three ‘longComputation()‘ calls are running in parallel.
```

10.1.4 Tree-like semimonads and monads

Tree-like type constructors are recursive types such as those described in Section 3.3. A typical example of a tree-like type constructor is the binary tree defined by the type equation:

$$\text{BT}^A \stackrel{\text{def}}{=} A + \text{BT}^A \times \text{BT}^A .$$

Statement 6.2.3.6 shows that BT^A is a functor by replacing the right-hand side $A + \text{BT}^A \times \text{BT}^A$ by an arbitrary "recursion scheme" S^{A,BT^A} and then showing that the recursive type constructor L^A defined by $L^A \stackrel{\text{def}}{=} S^{A,L^A}$ is a functor. As we will see, the type constructor L^A will be a semimonad or a monad with certain choices of $S^{\bullet,\bullet}$.

For lists, nested iteration goes over inner lists contained in an outer list. How does nested iteration work for a tree-shaped collection? An iteration over a tree enumerates the values at the *leaves* of a

tree. So, a tree analog of nested iteration implies that each leaf of an outer tree contains an inner tree. A `flatMap` function must concatenate all nested trees into a single "flattened" tree.

Let us implement the `flatMap` method for the binary tree $BT^\bullet$ in that way. It is convenient to define an equivalent curried function (denoted by "flm") with type signature:

$$\text{flm}^{A,B} : (A \to \text{BT}^B) \to \text{BT}^A \to \text{BT}^B \quad .$$

```
sealed trait BTree[A]
final case class Leaf[A](x: A) extends BTree[A]
final case class Branch[A](b1: BTree[A], b2: BTree[A]]) extends BTree[A]

def flm[A, B](f: A => BTree[B]): BTree[A] => BTree[B] = {              // t.flatMap(f) == flm(f)(t)
  case Leaf(x)          => f(x) // Here f(x) has type BTree[B], which could be a Leaf or a Branch.
  case Branch(b1, b2)   => Branch(flm(f)(b1), flm(f)(b2))              // Recursive calls of ‘flm‘.
}
```

The same implementation is written in the code notation as:

$$\text{flm}^{A,B}(f^{:A\to\text{BT}^B}) \stackrel{\text{def}}{=} \left| \begin{array}{c||c} & B + \text{BT}^B \times \text{BT}^B \\ \hline A & f \\ \text{BT}^A \times \text{BT}^A & b_1 \times b_2 \to \mathbb{0}^{:B} + \overline{\text{flm}}(f)(b_1) \times \overline{\text{flm}}(f)(b_2) \end{array} \right| \quad .$$

To visualize how the `flatMap` method operates on binary trees, let us compute `tree1.flatMap(f)`, where we take `tree1 =` and a function $f : A \to \text{BT}^B$ that has $f(a_1) =$, $f(a_2) = b_2$, and $f(a_3) =$. (Here a_i for $i = 1, 2, 3$ are some values of type A and b_i for $i = 0, ..., 4$ are some values of type B.) Evaluating the code of `flatMap`, we find `tree1.flatMap(f) =` .

So, we see that `flatMap` works by grafting a subtree into every `Leaf` of a given tree: A leaf is replaced by a new tree in line 6 in the code of `flatMap`. That code can be generalized to the recursive type PT^A (representing a "tree with P-shaped branches") defined by:

$$\text{PT}^A \stackrel{\text{def}}{=} A + P^{\text{PT}^A} \quad ,$$

for any given functor P. The disjunctive part $A + \mathbb{0}$ is replaced by a new tree:

```
sealed abstract class PT[P[_] : Functor, A]    // Need an ‘abstract class‘ due to implicits.
final case class Leaf[P[_] : Functor, A](x: A)              extends PT[P, A]
final case class Branch[P[_] : Functor, A](p: P[PT[P, A]]) extends PT[P, A]

def flm[P[_]: Functor, A, B](f: A => PT[P, B]): PT[P, A] => PT[P, B] = {
  case Leaf(x)      => f(x)     // Here f(x) has type PT, which could be a Leaf or a Branch.
  case Branch(p)    => Branch(p.map(t => flm(f)(t))  // Conceptually, Branch(p.map(flm(f))).
}
```

The same function is written in the code notation as:

$$\text{flm}^{A,B}\left(f^{:A\to P^{\text{PT}^B}}\right) \stackrel{\text{def}}{=} \left| \begin{array}{c||c} & B + P^{\text{PT}^B} \\ \hline A & f \\ P^{\text{PT}^A} & \left(\overline{\text{flm}}\,(f)\right)^{\uparrow P} \end{array} \right| \quad .$$

We can also implement `flatMap` for more general type constructors L defined by $L^A \stackrel{\text{def}}{=} P^A + P^{L^A}$ for some functor P. Such L^A can be visualized as trees with P-shaped branches and P-shaped leaves.

```
sealed abstract class PLPT[P[_] : Functor, A]
final case class Leaf[P[_] : Functor, A](px: P[A]) extends PLPT[P, A]
final case class Branch[P[_] : Functor, A](pb: P[PLPT[P, A]]) extends PLPT[P, A]

def flm[P[_]: Functor, A, B](f: A => PLPT[P, B]): PLPT[P, A] => PLPT[P, B] = {
  case Leaf(px)      => Branch(px.map(f))   // Here px.map(f) has type P[PLPT[P, B]].
  case Branch(pb)    => Branch(pb.map(t => flm(f)(t))
}
```

$$\text{flm}^{A,B}\left(f^{:A\to P^{\text{PLPT}^B}}\right) \stackrel{\text{def}}{=} \left|\begin{array}{c||cc} & P^B & P^{\text{PLPT}^B} \\ \hline P^A & \mathbb{0} & f^{\uparrow P} \\ P^{\text{PLPT}^A} & \mathbb{0} & \left(\overline{\text{flm}}\,(f)\right)^{\uparrow P} \end{array}\right| .$$

The code of `flatMap` for the `PLPT` tree never creates any leaves, only branches. We will see later how this prevents `PLPT` from being a full monad (it is only a semimonad; no `pure` method can satisfy the required laws). Nevertheless, having a lawful `flatMap` is sufficient for using `PLPT` in functor blocks.

The following examples show some use cases for tree-like monads.

Example 10.1.4.1 Implement the `flatMap` operation for a tree of configuration properties of the form:

```
url: http://server:8000
users:
  user:
    name: abcde
    pass: 12345
  user2:
    name: fghij
    pass: 67890
```

Write a function to convert trees into strings in this format.

Solution The code for this data structure must support any number of simple properties or branches with `String` labels. A suitable structure is a tree with P-shaped leaves and P-shaped branches, where the functor P is defined as $P^A \stackrel{\text{def}}{=} \text{List}^{\text{String}\times A}$. Implement the tree type:

```
sealed trait PropTree[A]  // Introduce the type parameter A for the values of properties.
final case class Simple[A](props: List[(String, A)])                  extends PropTree[A]
final case class Branches[A](branches: List[(String, PropTree[A])])   extends PropTree[A]
```

To pretty-print trees of this type, we need to keep track of the current level of indentation:

```
def prettyPrint[A](pt: PropTree[A], indent: String = "")(toString: A => String): String = (pt match {
  case Simple(props) => props.map { case (name, a) => indent + name + ": " + toString(a) }
  case Branches(brs) => brs.map { case (name, ps) => indent + name + ":\n" +
    prettyPrint(ps, indent + "  ")(toString) }
}).mkString("\n") + "\n"
```

For convenience, the methods `flatMap` and `toString` can be defined directly on the trait:

```
sealed trait PropTree[A] {
  def flatMap[B](f: A => PropTree[B]): PropTree[B] = this match {
    case Simple(props) => Branches(props.map { case (name, a) => (name, f(a)) })
    case Branches(brs) => Branches(brs.map { case (name, ps) => (name, ps.flatMap(f)) })
  }
  override def toString: String = prettyPrint(this)(_.toString)
}
```

The following code illustrates the `flatMap` operation by replacing all integer leaf values below `100` by a simple property and other values by a branch of properties:

```
val pt1: PropTree[Int] = Simple(List("quantity1" -> 50, "quantity2" -> 250))

scala> println(pt1.toString)
quantity1: 50
quantity2: 250
```

```
val pt2 = pt1.flatMap { x => if (x < 100) Simple(List("value" -> x)) else Branches(List(
               "large" -> Simple(List("value" -> x / 100, "factor" -> 100)),
               "small" -> Simple(List("value" -> x % 100, "factor" -> 1))
          )) }

scala> println(pt2.toString)
quantity1:
  value: 50
quantity2:
  large:
    value: 2
    factor: 100
  small:
    value: 50
    factor: 1
```

Example 10.1.4.2 Implement variable substitution for a simple arithmetic language. Example use:

```
val expr1: Term[String] = Const(123) * Var("a") + Const(456) * Var("b")
val expr2: Term[String] = Const(20) * Var("c")

scala> expr1.flatMap { x => if (x == "a") expr2 else Var(x) } // Substitute "a" = expr2 in expr1.
res0: Term[String] = Plus(Mult(Const(123), Mult(Const(20), Var("c"))), Mult(Const(456), Var("b")))
```

Solution Begin by implementing the basic functionality of the language: constants, variables, addition, and multiplication. The type parameter `A` in `Term[A]` is the type of labels for variables.

```
sealed trait Term[A] {
  def +(other: Term[A]): Term[A] = Plus(this, other)
  def *(other: Term[A]): Term[A] = Mult(this, other)
  def map[B](f: A => B): Term[B] = ???
  def flatMap[B](f: A => Term[B]): Term[B] = ???
}
final case class Const[A](value: Int)                extends Term[A]
final case class Var[A](name: A)                     extends Term[A]
final case class Plus[A](t1: Term[A], t2: Term[A])   extends Term[A]
final case class Mult[A](t1: Term[A], t2: Term[A])   extends Term[A]
```

The type constructor `Term` is an example of an abstract syntax tree and can be equivalently defined as:

$$\text{Term}^A \stackrel{\text{def}}{=} L^A + S^{\text{Term}^A} \quad , \qquad L^A \stackrel{\text{def}}{=} \text{Int} + A \quad , \qquad S^B \stackrel{\text{def}}{=} B \times B + B \times B \quad ,$$

where the functors L and S describe the structure of the leaves and the branches of the tree.

The code of the `map` method is:

```
def map[B](f: A => B): Term[B] = this match {  // This code must be within ‘trait Term[A]‘.
  case Const(value)   => Const(value)
  case Var(name)      => Var(f(name))
  case Plus(t1, t2)   => Plus(t1 map f, t2 map f)
  case Mult(t1, t2)   => Mult(t1 map f, t2 map f)
}
```

The code of `flatMap` replaces variables by new trees, leaving everything else unchanged:

```
def flatMap[B](f: A => Term[B]): Term[B] = this match {  // This code must be within ‘trait Term[A]‘.
  case Const(value)   => Const(value)
  case Var(name)      => f(name)
  case Plus(t1, t2)   => Plus(t1 flatMap f, t2 flatMap f)
  case Mult(t1, t2)   => Mult(t1 flatMap f, t2 flatMap f)
}
```

Note that the `flatMap` functions for lists, pass/fail monads, and tree-like monads are information-preserving: no data is discarded from the original tree or from any computed results.

Example 10.1.4.3 Show that a perfect-shaped tree *cannot* have an information-preserving `flatMap`.

Solution The type constructor P^A for a perfect-shaped binary tree is defined by $P^A \stackrel{\text{def}}{=} A + P^{A\times A}$. We can (non-rigorously) view the type P^A as a disjunction with "infinitely many" parts:

$$P^A \cong A + A\times A + A\times A\times A\times A + \ldots \quad ,$$

where the nth part ($n = 1, 2, \ldots$) is a product of 2^n copies of A. The `flatMap` function must have the type:

$$\text{flm} : (A \to P^B) \to P^A \to P^B \quad .$$

How could `flatMap` work? For any function $f^{:A\to P^B}$, we must compute a result value $\text{flm}\,(f)(p)$ of type P^B. A possible value p of type P^A is, say, a product of 4 copies of A. The only way of obtaining a value of type P^B is to apply f to some value of type A. But p contains 4 such values. If we apply f to each one of them, we will obtain 4 different values of type P^B. Some of these values may correspond to the part $B\times B$ in the disjunction, others to $B\times B\times B\times B$, etc. The total number of values of type B depends on the result of that computation and is not necessarily equal to a power of 2. So, it will be impossible to accommodate all those values of type B within a *single* perfect-shaped tree of type P^B. If we discard some values of type B to fit the rest into a perfect-shaped tree, we will lose information. So, it is not possible to implement an information-preserving `flatMap` for a perfect-shaped binary tree. (One can prove that P^A does not have a monad implementation.)

10.1.5 The `Reader` monad

This chapter started with the list-like monads whose `flatMap` method is motivated by the requirements of nested iteration. We then looked at tree-like monads, which generalize nested list iterations to tree grafting. It turns out that the `flatMap` method can be generalized to many other type constructors that are useful for various programming tasks not limited to nested iteration.

A general (semi)monad type constructor L^A no longer represents a collection of data items of type A. Instead, we regard L^A informally as a value of type A wrapped in a special "computational effect". We view "computations with an L-effect" as functions of type $A \to L^B$ (as in `flatMap`'s argument type). In this view, different monads — such as list-like, pass/fail, or tree-like — implement different kinds of effects. An ordinary function of type $A \to B$ is a computation with a "trivial effect".

In this sense, monadic effects are *not* side effects. Functions of type $A \to L^B$ can be referentially transparent and behave as values. Informally, an "L-effect" describes the information computed by a function of type $A \to L^B$ in addition to a value of type B. To make the vague idea of "effect" concrete, we write the required type of the computation in the form $A \to L^B$ with a specific type constructor L. In this and the next subsections, we will look at some monads that can be derived in that approach: the `Reader`, `Writer`, `Eval`, `State`, and `Cont` ("continuation") monads.

The first of those, the `Reader`, corresponds to a function that computes a result while using some additional data of a fixed type Z. A function consuming ("reading") that additional data will have type $A\times Z \to B$ instead of $A \to B$. It remains to rewrite the type $A\times Z \to B$ in the form $A \to L^B$ with a suitable choice of a type constructor $L^\bullet$. By currying, we obtain an equivalent type:

$$(A\times Z \to B) \cong (A \to Z \to B) \quad ,$$

which has the form $A \to L^B$ if we define $L^A \stackrel{\text{def}}{=} Z \to A$. This type constructor is called the `Reader` monad and is denoted by $\text{Read}^{Z,A} \stackrel{\text{def}}{=} Z \to A$. The Scala definition is `type Reader[Z, A] = Z => A`.

Fully parametric implementations of `map` and `flatMap` directly follow from their type signatures:

```
def map[A, B](r: Z => A)(f: A => B): (Z => B) = r andThen f  // Example 5.4.1.7.
def flatMap[A, B](r: Z => A)(f: A => Z => B): (Z => B) = { z => f(r(z))(z) } // Exercise 5.4.2.11(c).
```

What are the use cases for `Reader`? If we write a functor block with a value of type `Reader[Z, A]` as a "source" value, it follows that other source values must have type `Reader[Z, B]`, `Reader[Z, C]`, and so on. In other words, the type Z must be fixed for the entire functor block. So, data of type Z plays the role of a *common dependency* that may be used by all computations within the functor block.

As an example, imagine a program built up by composing several "procedures". Each "procedure" takes an argument and returns a value, like an ordinary function, but also may need to run a Unix shell command. For simplicity, assume that the interface to shell commands is a function that runs a new command with given input, capturing the command's entire output. The type of that function may be $\text{String} \times \text{String} \to \text{Int} \times \text{String}$, taking the command string and the input for the command, and returning the command's exit code and the output.

```
type RunSh = (String, String) => (Int, String)
```

A simple implementation (that does not handle any run-time exceptions) is:

```
import sys.process._
val runSh: RunSh = { (command, input) =>
  var result: Array[Char] = Array()
  val p: Process = command.run(new ProcessIO(
    { os => os.write(input.getBytes); os.close() },
    { is => result = scala.io.Source.fromInputStream(is).toArray; is.close() },
    _.close())
  )
  val exitCode = p.exitValue()
  (exitCode, new String(result))
}
```

We can now use this "shell runner" to execute some standard Unix commands:

```
scala> runSh("echo -n abcd", "") // Use ‘-n‘ to avoid trailing newlines.
res0: (Int, String) = (0, "abcd")

scala> runSh("cat", "xyz")._2   // Equivalent to ‘cat < $(echo -n xyz)‘.
res1: String = "xyz"
```

Consider a program that determines the total line count of all files under a given directory, but only including files whose names match given patterns. We may split the computation into three "procedures": 1) list all files under the given directory, 2) filter out files whose names do not match the given patterns, 3) determine the line count for each of the remaining files. For the purposes of this example, we will implement each stage as a function that uses `runSh` to run a shell command.

```
def listFiles(runSh: RunSh, dir: String): String = runSh(s"find $dir -type f", "")._2
def filterFiles(runSh: RunSh, files: String, patterns: String): String =
  runSh(s"grep -f $patterns", files)._2
def lineCounts(runSh: RunSh, files: String): Array[Int] = files.split("\n")   // Array of file names.
  .map { file => runSh("wc -l $file", "")._2.replaceAll("^ +", "").split(" ")(0).toInt }
```

This code assumes that file names do not contain the newline character `"\n"`. Use this code only as an illustration of a use case for the `Reader` monad.

We can now write the program like this:

```
def getLineCount(runSh: RunSh, dir: String, patterns: String): Int = {
  val fileList = listFiles(runSh, dir)
  val filtered = filterFiles(runSh, fileList, patterns)
  val counts = lineCounts(runSh, filtered)
  counts.sum
}
```

The value `runSh` is a common dependency of all the "procedures" and is repeated throughout the code. This repetition becomes a problem if we have many "procedures" within different code modules; or when different "runners" of type `RunSh` are used within the program. For instance, one "runner" executes commands on a remote machine, while another "runner" is used for testing and returns fixed results without running any shell commands. How can we avoid writing repetitive code and at the same time assure that the correct "runners" are passed to all the "procedures"?

The `Reader` monad offers a solution: it allows us to combine smaller "procedures" into larger ones while passing the "runner" values automatically. We first need to convert all "procedures" into functions of type `A => Reader[RunSh, B]` with suitable choices of type parameters `A`, `B`:

```
type Reader[Z, A] = Z => A
def listFilesR(dir: String): Reader[RunSh, String] = runSh => runSh(s"find $dir -type f", "")._2
def filterFilesR(patterns: String): String => Reader[RunSh, String] = files => runSh =>
  runSh(s"grep -f $patterns", files)._2
def lineCountsR(files: String): Reader[RunSh, Array[Int]] = runSh => files.split("\n")
  .map { file => runSh("wc -l $file", "")._2.replaceAll("^ +", "").split(" ")(0).toInt }
```

This allows us to express `getLineCount` as a combination of the three "procedures" by using the `Reader` monad's `flatMap` function (for convenience, assume that we defined an extension method `flatMap`):

```
def getLineCount(dir: String, patterns: String): Reader[RunSh, Int] = listFilesR(dir)
  .flatMap(files => filterFilesR(patterns)(files))
  .flatMap(lineCountsR).map(_.sum) // Assuming an extension method ‘map‘ is defined for ‘Reader‘.
```

For better readability, rewrite this code equivalently in the functor block syntax:

```
def getLineCountR(dir: String, patterns: String): Reader[RunSh, Int] = for {
  files       <-  listFilesR(dir)
  filtered    <-  filterFilesR(patterns)(files)
  lineCounts  <-  lineCountsR(filtered)
} yield lineCounts.sum

val program: Reader[RunSh, Int] = getLineCountR(".", "patterns.txt")
```

We obtained a value `program` of type `Reader[RunSh, Int]`, which is a function type. It is important to note that at this point no shell commands have been run yet. We merely packaged all the necessary actions into a function value (a **monadic program**). We now need to apply that function to a "runner" value of type `RunSh`. Only then we will obtain the actual line count:

```
val count: Int = program(runSh)
```

The `Reader` monad allows us to split the code into two stages: first, we build up a monadic program (a value of type `Reader`) from its parts ("procedures"). Monadic programs are ordinary values that can be passed as arguments to functions, stored in arrays, etc. We may compose smaller monadic programs into larger ones using `map`, `flatMap`, or `for`/`yield` blocks. We have the full flexibility of manipulating those monadic values and combining them into larger monadic programs in any order, since no shell commands are being run while we perform these manipulations. When we are done building the full monadic program, we can "run" it using a chosen runner.

Since the runner is the common dependency of all `Reader`-monadic programs, running a monadic program means performing **dependency injection**. At the point of running a `Reader` program, we have the full flexibility of choosing the value of the dependency. The code guarantees that the dependency will be passed correctly to each individual part of the monadic program.

Implicit values for dependency injection Scala's implicit argument feature allows us to solve the problem of dependency injection in a different way. Instead of converting all code to use the `Reader` monad, we convert all code to use an implicit argument for the common dependency:

```
def listFilesIm(dir: String)(implicit runSh: RunSh): String = ...
def filterFilesIm(patterns: String)(files: String)(implicit runSh: RunSh): String = ...
def lineCountsIm(files: String)(implicit runSh: RunSh): Array[Int] = ...

def getLineCountIm(dir: String, patterns: String)(implicit runSh: RunSh): Int = {
  val fileList = listFilesIm(dir)
  val filtered = filterFilesIm(fileList, patterns)
  val counts = lineCountsIm(filtered)
  counts.sum
}
```

Compare this code and the code that uses the `Reader` monad: the type signatures of functions are the same up to the `implicit` keyword in the last argument. Scala's implicit arguments reproduce the code style of the `Reader` monad (especially with Scala 3's implicit function types[5]).

[5] See https://www.scala-lang.org/blog/2016/12/07/implicit-function-types.html

10.1.6 The `Writer` monad

The `Writer` monad represents a function of type $A \to B$ that returns its result (of type B) and additionally outputs some information (say, logging data) about the computation just performed. Let W be the type of the logging data. To model this situation, we then need a function $f : A \to B$ and additionally a function $g : A \to W$ that computes the logging output. So, we can define the type of a "computation with the `Writer` effect" via the product of the two function types:

$$(A \to B) \times (A \to W) \cong A \to B \times W \quad .$$

Since this type has be of the form $A \to L^B$, we must define the functor L as $L^A \stackrel{\text{def}}{=} A \times W$. This is the type constructor of the `Writer` monad, denoted by $\text{Writer}^{A,W} \stackrel{\text{def}}{=} A \times W$.

If several computations are performed one after another in the `Writer` monad, the logging information should be "accumulated" in some way. In the logging example, additional lines are simply appended to the log file. It means that we must be able somehow to combine several values of type W into one. A general way of doing that is to require W to be a semigroup (see Example 8.2.5.4) with a binary operation $\oplus$. We can then implement the `flatMap` method for `Writer` like this:

```
final case class Writer[A, W: Semigroup](a: A, log: W) {
  def flatMap[B](f: A => Writer[B, W]): Writer[B, W] = {
   val Writer(b, newLog) = f(a) // Pattern-match to destructure the value f(a).
   Writer(b, log |+| newLog)    // Use the semigroup operation |+|.
  }
}
```

The logging type W is often a monoid (a semigroup with an "empty" value). If so, `Writer[A, W]` will be a full monad whose `pure` method is implemented as:

```
def pure[A, W: Monoid]: A => (A, W) = a => (a, Monoid[W].empty)
```

When W is a semigroup but not a monoid, `Writer[A, W]` will be a semimonad but not a monad.

An example of using a `Writer` semimonad is logging with timestamps where we need to keep track of the earliest and the latest timestamp. Define the type $W \stackrel{\text{def}}{=}$ `Logs` and a semigroup operation `|+|` by:

```
final case class Logs(begin: LocalDateTime, end: LocalDateTime, message: String) {
  def |+|(other: Logs): Logs = Logs(begin, other.end, message + "\n" + other.message)
}      // For simplicity, we assume that timestamps will be monotonically increasing.
```

The type `Logs` is not a monoid because its binary operation discards some of the input data, so we cannot define an "empty" value satisfying the identity laws (see Eq. (8.2) in Example 8.2.5.6).

We can now use the semimonad `Writer[A, Logs]`. Here are some example computations:

```
type Logged[A] = Writer[A, Logs]
def log[A](message: String)(x: A): Logged[A] = {        // Define this function for convenience.
  val timestamp = LocalDateTime.now
  new Logged(x, Logs(timestamp, timestamp, message))
}
def compute[A](x: => A): A = { Thread.sleep(100L); x }           // Simulate a long computation.

scala> val result: Logged[Double] = for {
  x <- log("begin with 3")(compute(3))              // The initial source type is ‘Logged[Int]‘.
  y <- log("add 1")(compute(x + 1))
  z <- log("multiply by 2.0")(compute(y * 2.0))  // The type of result becomes ‘Logged[Double]‘.
} yield z                                        // The computation should take between 300 and 400 ms.
res0: Logged[Double] = Writer(8.0,Logs(2020-02-15T22:02:42.313,2020-02-15T22:02:42.484,begin with 3
add 1
multiply by 2.0))
```

Unlike the `Reader` monad, which delays all computations until a runner is called, a monadic value of type `Writer[A, W]` already contains the final computed values of types `A` and `W`.

10.1.7 The `State` monad

Heuristically, the `Reader` monad $\text{Read}^{S,A}$ is able to "read" values of type S, while the `Writer` monad $\text{Writer}^{A,S}$ may "write" values of type S, in addition to computing the result of type A. The `State` monad, denoted by $\text{State}^{S,A}$, combines the functionality of `Reader` and `Writer` in a special way: an extra value of type S is updated and automatically passed from one computation to the next.

To derive the required type constructor, consider a computation of type $A \to B$ that additionally needs to read and to write a value of type S. Since the total input is a pair of A and S, and the total output is a pair of B and S, this kind of computation is represented by a function of type $A \times S \to B \times S$. We now try to rewrite this type in the form $A \to L^B$ with a suitable type constructor L. It is clear that we need to curry the argument A. The result is:

$$(A \times S \to B \times S) \cong (A \to S \to B \times S) = A \to L^B \quad , \quad \text{where} \quad L^B \stackrel{\text{def}}{=} S \to B \times S \quad .$$

So, the `State` monad must be defined by the type constructor $\text{State}^{S,A} \stackrel{\text{def}}{=} S \to A \times S$. This is a function that computes a value of type A while using and possibly updating the "state value" of type S.

To code of the `flatMap` method for this type constructor was derived in Example 5.4.1.11(c). It does indeed pass the updated state value to the next computation:

```
type State[S, A] = S => (A, S)
def flatMap[S, A, B](prev: State[S, A])(f: A => State[S, B]): State[S, B] = { s =>
  val (a, newState) = prev(s)   // Compute result of type ‘A‘, updating the state.
  f(a)(newState)                // Pass the updated state to the next computation.
}
```

An example of using the `State` monad is the task of implementing a random number generator. A simple generator is the **Lehmer's algorithm**,[6] which generates integer sequences x_n defined by:

$$x_{n+1} \stackrel{\text{def}}{=} (48271 * x_n) \,\%\, (2^{31} - 1) \quad , \qquad 1 \le x_n \le 2^{31} - 2 \quad , \qquad n = 0, 1, 2, ...$$

The "updating" function for this sequence, $x_{n+1} = \text{lehmer}\,(x_n)$, can be implemented as:

```
def lehmer(x: Long): Long = x * 48271L % ((1L << 31) - 1)
```

In many applications, one needs uniformly distributed floating-point numbers in the interval $[0, 1]$. To produce such numbers, let us define a helper function:

```
def uniform(x: Long): Double = (x - 1).toDouble / ((1L << 31) - 3)   // Enforce the interval [0, 1].
```

To use the uniform generator, we need to provide an initial value x_0 (the "seed") and then call the function `lehmer` repeatedly on successive values. The code would look like this:

```
val s0 = 123456789L  // A "seed" value.
val s1 = lehmer(s0)
val r1 = uniform(s1)
... // Use pseudo-random value r1.
val s2 = lehmer(s1)
val r2 = uniform(s2)
... // Use pseudo-random value r2.
val s3 = lehmer(s2)       // And so on.
```

We need to keep track of the generator's state values `s1`, `s2`, ..., that are not directly needed for other computations. This "bookkeeping" is error-prone since we might reuse a previous generator state by mistake. The `State` monad keeps track of the updated state values automatically and correctly. This comes at a cost: we need to convert all computations into `State`-typed monadic programs.

As a simple example, consider the task of generating uniformly distributed floating-point numbers in the interval $[0, 1]$. We need to maintain the generator state while computing the result. The floating-point generator is implemented as a monadic value of type `State[Long, Double]`:

```
val rngUniform: State[Long, Double] = { oldState =>
  val result = uniform(oldState)   // Enforce the interval [0, 1].
  val newState = lehmer(oldState)
  (result, newState)
}
```

[6] See https://en.wikipedia.org/wiki/Lehmer_random_number_generator

Code using `rngUniform` will be of the monadic type `State[Long, A]` for some `A`:

```
val program: State[Long, String] = for {  // Assume flatMap and map methods are defined for State.
  r1 <- rngUniform
  ... // Use pseudo-random value r1. The internal state of rngUniform is maintained automatically.
  r2 <- rngUniform
  ... // Use pseudo-random value r2.
} yield s"Pair is $r1, $r2" // Compute result of type String.
```

Monadic programs of this type can be composed in arbitrary ways. The "bookkeeping" of the state values is safely handled by the `State` monad and hidden from the programmer's view. When the entire monadic program has been composed, it needs to be "run" to extract its result value. Since the type `State[S, A]` is a function with argument of type S, the runner is just an application of that function to an initial value of type S, specifying the initial state:

```
val seed = 123456789L     // Initial state of the generator.

scala> program(seed)      // Run this monadic program.
res0: (String, Long) = ("Pair is 0.028744523433557146, 0.5269012540999576", 216621204L)
```

10.1.8 The eager/lazy evaluation monad

The monads `Reader`, `Writer`, and `State` manage extra information about computations. Those monads' effects can be viewed as working with an extra value of a certain fixed type. We now turn to monads whose effects are not values but special strategies of evaluation.

The first of these monads is called `Eval`, and its task is to encapsulate lazy and eager evaluations into a single type. A value of type `Eval[A]` can be eager (available now) or lazy (available later). Values of these sub-types can be combined with correct logic: for instance, a combination of eager and lazy values automatically becomes lazy.

To derive the type constructor, note that lazy values of type A are equivalent to functions of type $\mathbb{1} \to A$. So, the disjunctive type $\text{Eval}^A \stackrel{\text{def}}{=} A + (\mathbb{1} \to A)$ represents a value that is either lazy or eager:

```
sealed trait Eval[A]
final case class Eager[A](x: A)              extends Eval[A]
final case class Lazy[A](lazyX: Unit => A)   extends Eval[A]
```

It is useful to have functions converting between eager and lazy values whenever needed:

```
def get: Eval[A] => A = {
  case Eager(x)    => x
  case Lazy(lazyX) => lazyX(())
}

def now(x: A): Eval[A] = Eager(x)
def later(e: => A): Eval[A] = Lazy(_ => e)
```

Now we can implement a `flatMap` method that correctly keeps track of the evaluation strategy:

```
def flatMap[A, B](f: A => Eval[B]): Eval[A] => Eval[B] = {
  case Eager(x)    => f(x)         // This value can be eager or lazy, according to f(x).
  case Lazy(lazyX) => Lazy(_ => get(f(lazyX()))))  // Call ‘get‘ to avoid nested Lazy().
}
```

Assuming that `map` and `flatMap` are defined as extension methods for `Eval`, we may use functor blocks to combine eager and lazy computations freely:

```
val result: Eval[Int] = for {
  x <- later(longComputation1()) // Delay the long computation.
  y <- now(x + 2)       // Short computation, no need to delay.
  z <- later(longComputation2(y * 100))
} yield z
```

The value of `result` is a lazy computation because it involves lazy steps. It is quick to compute `result` because the long computations are not yet started. To extract the final value of type `Int` out of `result`, we need to evaluate `get(result)`, which will take a longer time.

10.1.9 The continuation monad

The continuation monad is another monadic design pattern that involves a special evaluation strategy called the **continuation-passing** programming style. In that style, functions do not return their results directly but instead call an auxiliary function that consumes the result. The auxiliary function is called a "continuation" or a "callback".

To compare the direct style with the continuation-passing style, consider this calculation:

```
def add3(x: Int): Int = x + 3
def mult4(x: Int): Int = x * 4
val result = add3(mul4(10))     // Will have result == 43 after this.
```

Now we add a callback argument to each function and rewrite the code as:

```
def add3(x: Int)(callback: Int => Unit): Unit = callback(x + 3)
def mult4(x: Int)(callback: Int => Unit): Unit = callback(x * 4)
def result(callback: Int => Unit): Unit = mult4(10)(r => add3(r)(callback))
```

To make the pattern more clear, replace the constant `10` by a function `pure` with a callback argument:

```
def pure(x: Int)(callback: Int => Unit): Unit =
  callback(x)

def result(callback: Int => Unit): Unit =
  pure(10) { x =>
    mult4(x) { y =>
      add3(y) { z =>
        callback(z)
      }
    }
  }
```

This code is a typical pattern of continuation-passing style. The final result of the calculation is only available as the bound variable `z` in a deeply nested function scope. This makes working with this code style more difficult: We can continue the program only by writing code either directly within that deeply nested scope or within the given `callback`. This is the heuristic reason why callback programming is known as "continuation-passing" style.

Another feature of the continuation-passing style is that callbacks could be called at a later time, concurrently with the main thread of computation. To show an example, redefine `add3` and `mult4` as:

```
def add3(x: Int)(callback: Int => Unit): Unit = { Future(callback(x + 3)); () }
def mult4(x: Int)(callback: Int => Unit): Unit = { Future(callback(x * 4)); () }
```

The new code schedules the calls to `callback` on separate threads. But the type signatures of `add3` and `mult4` hide this fact: they just return `Unit`. So, the code of `def result(...)` remains unchanged.

The type signature of `result` is $(\text{Int} \to \mathbb{1}) \to \mathbb{1}$, which shows that it consumes a callback (of type $\text{Int} \to \mathbb{1}$). A function that consumes a callback is at liberty to call the callback later, and to call it several times or not at all. The type $(\text{Int} \to \mathbb{1}) \to \mathbb{1}$ does not show whether the callback will be called; it merely "registers" the callback for possible later use. This gives us flexibility in the execution strategy, at the cost of making the code more complicated to write and to understand.

One complication is that it is difficult to maintain code that contains deeply nested function scopes. The continuation monad solves this problem by converting nested function scopes into more easily composable functor blocks. The code also becomes more readable.

To derive the required type constructor, consider a computation of type $A \to B$ that needs to use the continuation-passing style. Instead of returning a value of type B, it registers a callback function for possible later use and returns a `Unit` value. If the callback has type $B \to \mathbb{1}$, the total input of our computation is $A \times (B \to \mathbb{1})$ while the output is simply $\mathbb{1}$. So, the type of a continuation-passing computation is $A \times (B \to \mathbb{1}) \to \mathbb{1}$. Rewrite this type in the form $A \to L^B$ with a suitable functor L:

$$(A \times (B \to \mathbb{1}) \to \mathbb{1}) \cong (A \to (B \to \mathbb{1}) \to \mathbb{1}) = A \to L^B \quad , \quad \text{where} \quad L^A \stackrel{\text{def}}{=} (A \to \mathbb{1}) \to \mathbb{1} \quad .$$

It is sometimes helpful if the callback returns a more informative value than `Unit`. For instance, that value could show error information or give access to processes that were scheduled concurrently.

So, we generalize the type constructor L to $(A \to R) \to R$, where R is a fixed "result" type. This type constructor is called the **continuation monad** and is denoted by $\text{Cont}^{R,A} \stackrel{\text{def}}{=} (A \to R) \to R$.

How does the continuation monad make callback-based code composable? The answer is in the code of the `flatMap` method. Its implementation is (see Exercise 5.4.2.12):

```
type Cont[R, A] = (A => R) => R
def flatMap[R, A, B](ca: Cont[R, A])(f: A => Cont[R, B]): Cont[R, B] = { br => ca(a => f(a)(br)) }
```

The code of `flatMap` substitutes a new callback, `br: B => R`, into the innermost scope of the computation `f`. In this way, we obtain easy access to the innermost callback scope. This trick makes code with deeply nested callbacks composable.

After defining `map` and `flatMap` as extension methods on `Cont`, we can rewrite the code above as:

```
def pure[A]: A => Cont[Unit, A] = a => ar => ar(a)
def add3(x: Int): Cont[Unit, Int] = callback => callback(x + 3)
def mult4(x: Int): Cont[Unit, Int] = callback => callback(x * 4)

val result: Cont[Unit, Int] = for {
  x <- pure(10)
  y <- mult4(x)
  z <- add3(y)
} yield z
```

This style of code is more readable and easier to modify.

The result of the computation has type $\text{Cont}^{\mathbb{1},\text{Int}}$ (so, it is a function). Like the `State` and the `Reader` monads, the continuation monad delays all computations until we apply a runner. One way of extracting values from `Cont`-monadic programs is to use a runner producing a `Future` value that resolves when the callback is called:

```
def runner[A](c: Cont[Unit, A]): Future[A] = {
  val pr = Promise[A]() // scala.concurrent.Promise
  c { a => pr.success(a) }  // Resolve the promise.
  pr.future        // Create a Future from Promise.
}
// Wait for the Future value.
Await.result(runner(result), Duration.Inf)
```

This runner uses special low-level features of the `Future` class, such as a mutable value of type `Promise`. If these features are used in many places in the code, the programmer risks creating concurrency bugs, such as race conditions or deadlocks, that are difficult to fix. When using the continuation monad, typically there will be only one place in the code where a monadic program is being "run". In this way, we isolate the high-level business logic in the `Cont` monad from the low-level code of the runners.

We conclude this subsection with some more examples of using the continuation monad.

Example 10.1.9.1 Each arithmetic computation, such as `add3` or `mult4`, now has a certain cost, which is a value of a monoid type W. Use the monad `Cont[W, A]` to implement computations with a specified cost. The total cost must add up automatically when computations are chained using `flatMap`.

Solution The functions `pure`, `add3`, and `mult4` need to be redefined with new types:

```
implicit val monoidW: Monoid[W] = ???  // Implement the "cost" monoid here.
def pure[A](a: A): Cont[W, A] = { ar => ar(a) }
def add3(x: Int, cost: W): Cont[W, Int] = { callback => callback(x + 3) |+| cost }
def mult4(x: Int, cost: W): Cont[W, Int] = { callback => callback(x * 4) |+| cost }
```

The computation of `result` can be now written as:

```
val result: Cont[W, Int] = for {
  x <- pure(10)
  y <- mult4(x, cost1) // Here, cost1 and cost2 are some values of type W.
  z <- add3(y, cost2)
} yield z
```

The implementation of `runner` now needs to take an initial cost value. We can generalize the previous code of the runner to an arbitrary result type R:

```
def runCont[R, A](c: Cont[R, A], init: R): (R, Future[A]) = {
  val promise = Promise[A]()
  val res = c {a => promise.success(a); init}    // Resolve the promise and return init.
  (res, promise.future)      // Return the new result value r together with a Future[A].
```

```
}
val (totalCost, futureResult) = runCont(result, Monoid[W].empty)
val resultInt = Await.result(futureResult, Duration.Inf)   // Wait for the Future value.
```

If the code contains many other monadic operations such as `add3` and `mult4`, it is inconvenient to hard-code the cost each time. Instead, we can easily implement a function that adds a given cost to any given monadic operation:

```
def addCost[A](c: Cont[W, A], cost: W): Cont[W, A] = { callback => c(callback) |+| cost }
```

Example 10.1.9.2 Convert the callback-based API of `java.nio` to the continuation monad. Read a file into a string, write that string to another file, and finally read the new file again to verify that the string was written correctly. Do not implement any error handling.

Solution The `java.nio` package provides APIs for asynchronous input/output using buffers. For instance, the code for reading a file into a string works by creating a `fileChannel` and then calling `fileChannel.read` with a callback (encapsulated by the `CompletionHandler` class). The result of reading the file is available only when the callback is called, and only within the callback's scope:

```
import java.nio.ByteBuffer
import java.nio.channels.{AsynchronousFileChannel, CompletionHandler}
import java.nio.file.{Paths, StandardOpenOption => SOO}

val fileChannel = AsynchronousFileChannel.open(Paths.get("sample.txt"), SOO.READ)
val buffer = ByteBuffer.allocate(256)// In our simple example, the file is shorter than 256 bytes.

fileChannel.read(buffer, 0, null, new CompletionHandler[Integer, Object] {
  override def failed(e: Throwable, attachment: Object): Unit = println(s"Error reading file: $e")
  override def completed(byteCount: Integer, attachment: Object): Unit = {
    println(s"Read $byteCount bytes")
    fileChannel.close()
    buffer.rewind()
    buffer.limit(byteCount)
    val data = new String(buffer.array()) // Within this scope, we can work with the obtained data.
  }
}
```

Writing data to file is implemented similarly:

```
val outputFileChannel = AsynchronousFileChannel.open(Paths.get("sample2.txt"), SOO.CREATE, SOO.WRITE)
outputFileChannel.write(buffer, 0, null, new CompletionHandler[Integer, Object] {
  override def failed(e: Throwable, attachment: Object): Unit = println(s"Error writing file: $e")
  override def completed(byteCount: Integer, attachment: Object): Unit = {
    println(s"Wrote $byteCount bytes")
    outputFileChannel.close()
    ... // Continue the program within the scope of this callback.
  }
}
```

This API forces us to write the business logic of the program in deeply nested callbacks, since the results of input/output operations are only available within the callback scopes. The continuation monad solves this problem by converting all values into the function type `Cont[Unit, A]`. Begin by defining monadic-valued functions that encapsulate the `java.nio` APIs for reading and writing files:

```
type NioMonad[A] = Cont[Unit, A]
def nioRead(filename: String): NioMonad[ByteBuffer] = { callback =>
  val buffer = ByteBuffer.allocate(256)
  val channel = AsynchronousFileChannel.open(Paths.get(filename), SOO.READ)
  channel.read(buffer, 0, null, new CompletionHandler[Integer, Object] {
    override def failed(e: Throwable, attachment: Object): Unit = println(s"Error reading file: $e")
    override def completed(result: Integer, attachment: Object): Unit = {
      buffer.rewind()
      buffer.limit(result)
      channel.close()
      callback(buffer))
```

```
    }
  })
}
def nioWrite(buffer: ByteBuffer, filename: String): NioMonad[Int] = { callback =>
  val channel = AsynchronousFileChannel.open(Paths.get(filename), SOO.CREATE, SOO.WRITE)
  channel.write(buffer, 0, null, new CompletionHandler[Integer, Object] {
    override def failed(e: Throwable, attachment: Object): Unit = println(s"Error writing file: $e")
    override def completed(result: Integer, attachment: Object): Unit = {
      channel.close()
      callback(result.intValue)
    }
  })
}
```

Using these functions, we can implement the required code using a functor block in the `Cont` monad:

```
val filesHaveEqualContent: NioMonad[Boolean] = for {
  buffer1 <- nioRead("sample.txt")
  _       <- nioWrite("sample2.txt")
  buffer2 <- nioRead("sample2.txt")
} yield { new String(buffer1.array()) == new String(buffer2.array()) }
```

The code has become significantly easier to work with, as its high-level logic is clearly displayed.

Since the input/output operations are run concurrently, the value of type `NioMonad[Boolean]` is a function that will compute its `Boolean` result at some time in the future. We can use the runner shown above to wait for that value to become available:

```
val result: Future[Boolean] = runner(filesHaveEqualContent)

scala> Await.result(result, Duration.Inf)
res0: Boolean = true
```

10.1.10 Exercises

Exercise 10.1.10.1 For a given set of type `Set[Int]`, compute all subsets (w, x, y, z) of size 4 such that $w < x < y < z$ and $w + z = x + y$. (The values w, x, y, z must be all different.)

Exercise 10.1.10.2 Given 3 sequences xs, ys, zs of type `Seq[Int]`, compute all tuples (x, y, z) such that $x \in xs$, $y \in ys$, $z \in zs$ and $x < y < z$ and $x + y + z < 10$.

Exercise 10.1.10.3* Solve the n-queens problem on an $3 \times 3 \times 3$ cube.

Exercise 10.1.10.4 Read a file into a string and write it to another file using Java `Files` and `Paths` API. Use `Try` and `for`/`yield` to make that API composable and safe with respect to exceptions.

Exercise 10.1.10.5 Write a tiny library for arithmetic using `Future`s, implementing the functions:

```
def const(implicit ec: ExecutionContext): Int => Future[Int] = ???
def add(x: Int)(implicit ec: ExecutionContext): Int => Future[Int] = ???
def isEqual(x: Int)(implicit ec: ExecutionContext): Int => Future[Boolean] = ???
```

Use these functions to write a functor block (`for`/`yield`) program that computes $1 + 2 + ... + 100$ via a parallel computation and verifies that the result is correct.

Exercise 10.1.10.6 Given a semigroup W, make a semimonad out of the functor $F^A \stackrel{\text{def}}{=} E \to A \times W$.

Exercise 10.1.10.7 Implement `map` and `flatMap` for the tree-like functor $F^A \stackrel{\text{def}}{=} A + A \times A + F^A + F^A \times F^A$.

Exercise 10.1.10.8* Find the largest prime number below 1000 via a simple sieve of Eratosthenes.[7] Use the `State[S, Int]` monad with `S = Array[Boolean]`.

[7] See https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

10.2 Laws of semimonads and monads

10.2.1 Motivation for the semimonad laws

This chapter introduced semimonads to encode nested iteration as functor block programs with multiple source lines. What properties do we intuitively expect such programs to have?

When functor blocks describe iterations over data collections, a source line `x <- c` means that the value of `x` iterates over items in the collection `c`. An assignment line `y = f(x)` means that we define a local variable `y` to equal the expression `f(x)`. We expect to get the same result by iterating over a collection `c` whose values `x` were replaced by `f(x)`, i.e., by iterating over `c.map(f)`. It means that the following two code fragments should always give the same results:

```
val result1 = for {
  x <- c
  y = f(x)
  z <- g(y) // Same as z <- g(f(x)).
} yield z
 // Translating the functor block into methods:
val result1 = c.flatMap(x => g(f(x)))
```

```
val result2 = for {
  y <- c.map(x =>
                f(x))
  z <- g(y) // This code is unchanged.
 } yield z
 // Translating the functor block into methods:
val result2 = c.map(f).flatMap(y => g(y))
```

In the code just shown, an assignment line `y = f(x)` occurs before the last source line `z <- p(y)`. The other possibility is when the assignment line occurs *after* the last source line. A similar reasoning gives the requirement of equality between these two code fragments:

```
val result1 = for {
  x <- c
  z <- f(x)
  y = g(z)
} yield y
 // Translating the functor block into methods:
val result1 = c.flatMap(f).map(g)
```

```
val result2 = for {
  x <- c
  y <- f(x).map(z =>
                   g(z))
 } yield y
 // Translating the functor block into methods:
val result2 = c.flatMap { x => f(x).map(g) }
```

Now consider the case when there are more than two levels of nested iterations. Such programs are usually written as functor blocks with three or more source lines. However, any subgroup of source lines can be refactored into a separate functor block, and the result remains the same:

```
val result1 = for {
  x <- c
  y <- f(x)
  z <- g(y)
} yield z
 // Translating the functor block into methods:
val result1 = c.flatMap{ x => f(x).flatMap(g) }
```

```
val result2 = for {
  yy <- for { x <- c
              y <- f(x) } yield y
  z  <- g(yy)
 } yield z
 // Translating the functor block into methods:
val result2 = c.flatMap(f).flatMap(yy => g(yy))
```

We obtained three general requirements for the `flatMap` method. Let us now formulate these requirements as equations (or "laws").

For brevity, we will denote the `flatMap` method for a semimonad `S[_]` by "flm". We will write flm_S when we want to indicate explicitly the type constructor (S) being used. The type signature is:

```
def flm[A, B](f: A => S[B]): S[A] => S[B]
```

$$\text{flm}^{A,B} : (A \to S^B) \to S^A \to S^B \quad .$$

The first law is written in Scala code as:

```
c.flatMap(x => g(f(x))) ==
                c.map(f).flatMap(g)
```

In the code notation, we write this law in the point-free style by omitting the argument `c`:

$$\text{flm}\,(f^{:A\to B} \fatsemi g^{:B\to S^C}) = f^{\uparrow S} \fatsemi \text{flm}\,(g) \quad . \tag{10.2}$$

$$\begin{tikzcd} & S^B \arrow[dr, "\text{flm}\,(g^{:B\to S^C})"] & \\ S^A \arrow[ur, "f^{\uparrow S}"] \arrow[rr, "\text{flm}\,(f^{:A\to B} \fatsemi g^{:B\to S^C})"'] & & S^C \end{tikzcd}$$

This equation holds for arbitrary $f^{:A\to B}$ and $g^{:B\to S^C}$. This is a "**left naturality** law" of `flatMap` since it exchanges the order of lifted functions to the *left* of `flatMap`. More precisely, we may call this equation the naturality law of `flatMap[A, B]` "with respect to `A`" since $f^{\uparrow S}$ acts on the type parameter `A`.

The second law holds for arbitrary $f^{:A\to S^B}$ and $g^{:B\to C}$:

```
c.flatMap(f).map(g) ==
        c.flatMap { x => f(x).map(g) }
```

$$\begin{tikzcd} & S^B \arrow[dr, "(g^{:B\to C})^{\uparrow S}"] & \\ S^A \arrow[ur, "\text{flm}\,(f^{:A\to S^B})"] \arrow[rr, "\text{flm}\,(f^{:A\to S^B} \,⨾\, g^{\uparrow S})"'] & & S^C \end{tikzcd}$$

$$\text{flm}\,(f^{:A\to S^B} \,⨾\, g^{\uparrow S}) = \text{flm}\,(f) \,⨾\, g^{\uparrow S} \quad . \tag{10.3}$$

This is a **right naturality** law or "naturality with respect to `B`" of `flatMap[A, B]`. It manipulates a lifted function $g^{\uparrow S}$ to the right of `flatMap`, acting on the type parameter `B`.

The third law relates nested `flatMap` operations:

```
c.flatMap { x => f(x).flatMap(g) } ==
                c.flatMap(f).flatMap(g)
```

$$\begin{tikzcd} & S^B \arrow[dr, "\text{flm}\,(g^{:B\to S^C})"] & \\ S^A \arrow[ur, "\text{flm}\,(f^{:A\to S^B})"] \arrow[rr, "\text{flm}\,(f^{:A\to S^B} \,⨾\, \text{flm}\,(g))"'] & & S^C \end{tikzcd}$$

$$\text{flm}\left(f^{:A\to S^B} \,⨾\, \text{flm}\,(g^{:B\to S^C})\right) = \text{flm}\,(f) \,⨾\, \text{flm}\,(g) \quad . \tag{10.4}$$

This equation is called the **associativity law** of `flatMap`, for reasons we will explain later.

The three laws of semimonads express a programmer's intuitive reasoning about code. If a monad's implementation does not obey one of these laws, programs written with that monad may give wrong results even though the code looks correct. To avoid those hard-to-debug errors, we must verify that the code for every monad's `flatMap` method obeys the laws.

At this point, the three laws of semimonads may appear complicated and hard to understand and to verify. In the next subsections, we will derive a shorter and clearer formulation of those laws. For now, let us define a `Semimonad` typeclass and test the laws using the `scalacheck` library:

```
abstract class Semimonad[F[_]: Functor] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
implicit class SemimonadOps[F[_]: Semimonad, A](fa: F[A]) { // Define flatMap as an extension method.
  def flatMap[B](f: A => F[B]): F[B] = implicitly[Semimonad[F]].flatMap(fa)(f)
}
def checkSemimonadLaws[F[_], A, B, C]()(implicit ff: Semimonad[F],   // Use the ‘Arbitrary‘ typeclass
   fa: Arbitrary[F[A]], ab: Arbitrary[A => F[B]], bc: Arbitrary[B => F[C]]) = { // from ‘scalacheck‘.
    forAll { (f: A => F[B], g: B => F[C], fa: F[A]) =>                // Associativity law of flatMap.
      fa.flatMap(x => f(x).flatMap(g)) shouldEqual fa.flatMap(f).flatMap(g)
    }
} // Assuming that a Semimonad instance was defined for Seq[_], check the laws with specific A, B, C.
checkSemimonadLaws[Seq, Int, String, Double]()
```

10.2.2 The laws of `flatten`

In Section 9.2.1 we simplified the laws of the `filter` operation by passing to a simpler `deflate` function. We then showed that these two functions are equivalent if certain laws are assumed to hold for `filter`. We will now derive a similar relationship between the methods `flatMap` and `flatten`. We will see that `flatten` has fewer laws, and that its laws are simpler to verify.

Statement 10.2.2.1 The type of the method `flatten` (denoted by $\text{ftn}^A : S^{S^A} \to S^A$) is equivalent to the type of `flatMap` as long as `flatMap` satisfies its left naturality law (10.2).

Proof By definition, `flatMap` is expressed as a composition of `map` and `flatten`:

```
c.flatMap(f) == c.map(f).flatten
```

$$\text{flm}_S(f) = f^{\uparrow S} \,⨾\, \text{ftn}_S \quad .$$

$$\begin{tikzcd} & S^{S^B} \arrow[dr, "\text{ftn}"] & \\ S^A \arrow[ur, "(f^{:A\to S^B})^{\uparrow S}"] \arrow[rr, "\text{flm}\,(f^{:A\to S^B})"'] & & S^B \end{tikzcd}$$

Substituting $f \stackrel{\text{def}}{=} \text{id}^{:S^A\to S^A}$ into this equation, we get:

$$\text{flm}\,(\text{id}) = \underline{\text{id}^{\uparrow S}} \,⨾\, \text{ftn} = \text{ftn} \quad .$$

This expresses `flatten` through `flatMap`. It remains to show that the relationship between `flatten` and `flatMap` is an isomorphism. For that, we need to prove two properties:

(1) Starting with a given function $\text{ftn} : S^{S^A} \to S^A$, define $\text{flm}\,(f) \stackrel{\text{def}}{=} f^{\uparrow S} \,⨾\, \text{ftn}$ and then define a new function $\text{ftn}' \stackrel{\text{def}}{=} \text{flm}\,(\text{id})$. Prove that $\text{ftn}' = \text{ftn}$:

$$\text{ftn}' = \text{flm}\,(\text{id}) = \underline{\text{id}^{\uparrow S}} \,⨾\, \text{ftn} = \text{id} \,⨾\, \text{ftn} = \text{ftn} \quad .$$

If `flatMap` is defined via `flatten`, the left naturality law of `flatMap` is automatically satisfied:

$$\begin{aligned}\text{expect to equal flm}\,(f\,⨾\,g):\quad & f^{\uparrow S}\,⨾\,\text{flm}\,(g) = \underline{f^{\uparrow S}\,⨾\,g^{\uparrow S}}\,⨾\,\text{ftn}\\ \text{composition law of } S:\quad & = (f\,⨾\,g)^{\uparrow S}\,⨾\,\text{ftn} = \text{flm}\,(f\,⨾\,g)\quad .\end{aligned}$$

(2) Starting with a given function flm that satisfies the left naturality law, define $\text{ftn} \stackrel{\text{def}}{=} \text{flm}\,(\text{id})$ and then define a new function $\text{flm}'\,(f) \stackrel{\text{def}}{=} f^{\uparrow S}\,⨾\,\text{ftn}$. Prove that $\text{flm}' = \text{flm}$:

$$\begin{aligned}\text{expect to equal flm}\,(f):\quad & \text{flm}'\,(f) = f^{\uparrow S}\,⨾\,\text{ftn} = f^{\uparrow S}\,⨾\,\text{flm}\,(\text{id})\\ \text{left naturality law of flm}:\quad & = \text{flm}\,(f\,⨾\,\text{id}) = \text{flm}\,(f)\quad .\end{aligned}$$

Statement 10.2.2.2 If a `flatMap` function satisfies the three laws (10.2)–(10.4), the corresponding `flatten` function defined as $\text{ftn} \stackrel{\text{def}}{=} \text{flm}\,(\text{id})$ satisfies its *two* laws, with an arbitrary $f^{:A\to B}$:

$$\text{naturality law of ftn}:\quad f^{\uparrow S\uparrow S}\,⨾\,\text{ftn} = \text{ftn}\,⨾\,f^{\uparrow S}\quad , \tag{10.5}$$

$$\text{associativity law of ftn}:\quad \text{ftn}^{\uparrow S}\,⨾\,\text{ftn} = \text{ftn}\,⨾\,\text{ftn}\quad . \tag{10.6}$$

The following type diagrams illustrate these laws:

$$\begin{tikzcd}[column sep=small] & S^{S^B} \arrow[dr, "\text{ftn}^B"] & \\ S^{S^A} \arrow[ur, "(f^{:A\to B})^{\uparrow S\uparrow S}"] \arrow[dr, "\text{ftn}^A"'] \arrow[rr, phantom, "\text{(naturality)}"] & & S^B \\ & S^A \arrow[ur, "(f^{:A\to B})^{\uparrow S}"'] & \end{tikzcd}$$

$$\begin{tikzcd}[column sep=small] & S^{S^A} \arrow[dr, "\text{ftn}^A"] & \\ S^{S^{S^A}} \arrow[ur, "(\text{ftn}^A)^{\uparrow S}"] \arrow[dr, "\text{ftn}^{S^A}"'] \arrow[rr, phantom, "\text{(associativity)}"] & & S^A \\ & S^{S^A} \arrow[ur, "\text{ftn}^A"'] & \end{tikzcd}$$

Note that the associativity law involves two intermediate values of type S^{S^A} that are *not* necessarily equal. The associativity law requires only the final results (of type S^A) to be equal.

Proof By Statement 10.2.2.1, we have $\text{flm}\,(f) = f^{\uparrow S}\,⨾\,\text{ftn}$ since it is given that the left naturality law of `flatMap` holds. Substituting that into the other two laws of `flatMap`, we get:

$$\begin{aligned}\text{right naturality law of flm}:\quad & \text{flm}\left(f\,⨾\,g^{\uparrow S}\right) = \text{flm}\,(f)\,⨾\,g^{\uparrow S}\quad ,\\ \text{substitute flm via ftn}:\quad & \left(f\,⨾\,g^{\uparrow S}\right)^{\uparrow S}\,⨾\,\text{ftn} = f^{\uparrow S}\,⨾\,\text{ftn}\,⨾\,g^{\uparrow S}\quad ,\\ \text{set } f \stackrel{\text{def}}{=} \text{id}:\quad & g^{\uparrow S\uparrow S}\,⨾\,\text{ftn} = \text{ftn}\,⨾\,g^{\uparrow S}\quad ,\end{aligned}$$

which shows that Eq. (10.5) holds; and finally:

$$\begin{aligned}\text{associativity law of flm}:\quad & \text{flm}\left(f\,⨾\,\text{flm}\,(g)\right) = \text{flm}\,(f)\,⨾\,\text{flm}\,(g)\quad ,\\ \text{substitute flm via ftn}:\quad & \left(f\,⨾\,g^{\uparrow S}\,⨾\,\text{flm}\,(g)\right)^{\uparrow S}\,⨾\,\text{ftn} = \left(\underline{f\,⨾\,g^{\uparrow S}}\,⨾\,\text{ftn}\right)^{\uparrow S}\,⨾\,\text{ftn} = \underline{f^{\uparrow S}}\,⨾\,\text{ftn}\,⨾\,\underline{g^{\uparrow S}}\,⨾\,\text{ftn}\quad ,\\ \text{set } f \stackrel{\text{def}}{=} \text{id and } g \stackrel{\text{def}}{=} \text{id}:\quad & \text{ftn}^{\uparrow S}\,⨾\,\text{ftn} = \text{ftn}\,⨾\,\text{ftn}\quad .\end{aligned}$$

This verifies Eq. (10.6).

Statement 10.2.2.3 If a `flatten` function satisfies the laws (10.5)–(10.6) then the corresponding `flatMap` function defined by $\text{flm}\,(f) \stackrel{\text{def}}{=} f^{\uparrow S}\,⨾\,\text{ftn}$ will satisfy its three laws (10.2)–(10.4).

Proof By Statement 10.2.2.1, the left naturality law holds. To check the right naturality law:

$$\begin{aligned}\text{expect to equal flm}\,(f)\,⨾\,g^{\uparrow S}:\quad & \text{flm}\left(f\,⨾\,g^{\uparrow S}\right) = \underline{\left(f\,⨾\,g^{\uparrow S}\right)^{\uparrow S}}\,⨾\,\text{ftn} = f^{\uparrow S}\,⨾\,\underline{g^{\uparrow S\uparrow S}\,⨾\,\text{ftn}}\\ \text{naturality law of ftn}:\quad & = \underline{f^{\uparrow S}\,⨾\,\text{ftn}}\,⨾\,g^{\uparrow S} = \text{flm}\,(f)\,⨾\,g^{\uparrow S}\quad .\end{aligned}$$

To check the associativity law:

$$\begin{aligned}\text{expect to equal flm}\,(f)\,⨾\,\text{flm}\,(g):\quad & \text{flm}\left(f\,⨾\,\text{flm}\,(g)\right) = \left(f\,⨾\,\text{flm}\,(g)\right)^{\uparrow S}\,⨾\,\text{ftn} = \left(f\,⨾\,g^{\uparrow S}\,⨾\,\underline{\text{ftn}}\right)^{\uparrow S}\,⨾\,\text{ftn}\\ \text{associativity law of ftn}:\quad & = f^{\uparrow S}\,⨾\,\underline{g^{\uparrow S\uparrow S}\,⨾\,\text{ftn}}\,⨾\,\text{ftn}\\ \text{naturality law of ftn}:\quad & = \underline{f^{\uparrow S}\,⨾\,\text{ftn}}\,⨾\,\underline{g^{\uparrow S}\,⨾\,\text{ftn}} = \text{flm}\,(f)\,⨾\,\text{flm}\,(g)\quad .\end{aligned}$$

We have proved that `flatten` is equivalent to `flatMap` but has fewer laws.

By the parametricity theorem (see Appendix D), any purely functional code will obey naturality laws. All semimonads and monads in this chapter (except `Future`) are purely functional, so we will not need to verify their naturality laws. However, verifying the associativity law involves complicated derivations. Using `flatten` instead of `flatMap` will often make those derivations shorter.

10.2.3 Verifying the associativity law via `flatten`

The following examples will verify the associativity law of `flatten` for some standard monads.

Example 10.2.3.1 The standard Scala types `Either` and `Try` are examples of the monad $F^A \stackrel{\text{def}}{=} Z + A$, where Z is a fixed type. Show that this monad satisfies the associativity law.

Solution The type signature of `flatten` is $\text{ftn} : Z + (Z + A) \to Z + A$, and its code is:

```
def flatten[A]: Either[Z, Either[Z, A]] => Either[Z, A] = {
  case Left(z)            => Left(z)
  case Right(Left(z))     => Left(z)
  case Right(Left(z))     => Left(z)
  case Right(Right(a))    => Right(a)
}
```

$$\text{ftn}^{:Z+Z+A\to Z+A} \stackrel{\text{def}}{=} \begin{array}{|c||cc|} & Z & A \\ \hline Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} .$$

Since `flatten` is fully parametric, both sides of the law are fully parametric functions with the type signature $Z + Z + Z + A \to Z + A$. This type signature has *only one* fully parametric implementation: since it is not possible to produce values of unknown types A and Z from scratch, an implementation of $Z + Z + Z + A \to Z + A$ must return $Z + \mathbb{0}$ when the input contains a value of type Z; otherwise it must return $\mathbb{0} + A$. So, both sides of the law must have the same code.

To prove rigorously that only one implementation exists, we must use the Curry-Howard correspondence and a decision algorithm for constructive logic. Instead, let us verify the associativity law by an explicit derivation. First, we need to lift `flatten` to the functor F. The lifting code is:

$$(f^{:A\to B})^{\uparrow F} \stackrel{\text{def}}{=} \begin{array}{|c||cc|} & Z & B \\ \hline Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & f \end{array} , \qquad \text{ftn}^{\uparrow F} = \begin{array}{|c||ccc|} & Z & Z & A \\ \hline Z & \text{id} & \mathbb{0} & \mathbb{0} \\ Z & \mathbb{0} & \text{id} & \mathbb{0} \\ Z & \mathbb{0} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} .$$

For comparison, the Scala code for $\text{ftn}^{\uparrow F}$ (had we needed to write it) would look like this:

```
def fmapFlatten[A]: Either[Z, Either[Z, Either[Z, A]]] => Either[Z, Either[Z, A]] = {
  case Left(z)                    => Left(z)
  case Right(Left(z))             => Right(Left(z))
  case Right(Right(Left(z)))      => Right(Left(z))
  case Right(Right(Right(a)))     => Right(Right(a))
}
```

Now we can compute the two sides of the associativity law (10.6) via matrix composition:

$$\text{left-hand side}: \quad \text{ftn}^{\uparrow F} \fatsemi \text{ftn} = \begin{array}{|c||ccc|} & Z & Z & A \\ \hline Z & \text{id} & \mathbb{0} & \mathbb{0} \\ Z & \mathbb{0} & \text{id} & \mathbb{0} \\ Z & \mathbb{0} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} \fatsemi \begin{array}{|c||cc|} & Z & A \\ \hline Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} = \begin{array}{|c||cc|} & Z & A \\ \hline Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} ,$$

$$\text{right-hand side}: \quad \text{ftn}^{:Z+Z+(Z+A)\to Z+(Z+A)} \fatsemi \text{ftn} = \begin{array}{|c||cc|} & Z & \underline{Z+A} \\ \hline Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ \underline{Z+A} & \mathbb{0} & \underline{\text{id}} \end{array} \fatsemi \begin{array}{|c||cc|} & Z & A \\ \hline Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array}$$

$$\text{expand } \text{id}^{Z+A}: \quad = \begin{array}{|c||ccc|} & Z & Z & A \\ \hline Z & \text{id} & \mathbb{0} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} & \mathbb{0} \\ Z & \mathbb{0} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} \fatsemi \begin{array}{|c||cc|} & Z & A \\ \hline Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} = \begin{array}{|c||cc|} & Z & A \\ \hline Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} \quad .$$

The two sides of the associativity law are equal.

When it works, the technique of Curry-Howard code inference gives much shorter proofs than explicit derivations:

Example 10.2.3.2 Verify that the `Reader` monad, $F^A \stackrel{\text{def}}{=} Z \to A$, satisfies the associativity law.

Solution The type signature of `flatten` is $(Z \to Z \to A) \to Z \to A$. Both sides of the law (10.6) are functions with the type signature $(Z \to Z \to Z \to A) \to Z \to A$. By code inference with typed holes, we find that there is only one fully parametric implementation of this type signature, namely:

$$p^{:Z\to Z\to Z\to A} \to z^{:Z} \to p(z)(z)(z) \quad .$$

So, both sides of the law must have the same code, and the law holds.

Example 10.2.3.3 Show that the `List` monad ($F^A \stackrel{\text{def}}{=} \text{List}^A$) satisfies the associativity law.

Solution The `flatten[A]` method has the type signature $\text{ftn}^A : \text{List}^{\text{List}^A} \to \text{List}^A$ and concatenates the nested lists in their order. Let us first show a more visual (but less formal) proof of the associativity law. Both sides of the law are functions of type $\text{List}^{\text{List}^{\text{List}^A}} \to \text{List}^A$. We can visualize how both sides of the law are applied to a triple-nested list value p defined by:

$$p \stackrel{\text{def}}{=} [[[x_{11}, x_{12}, ...], [x_{21}, x_{22}, ...], ...], [[y_{11}, y_{12}, ...], [y_{21}, y_{22}, ...], ...], ...] \quad ,$$

where all x_{ij}, y_{ij}, ... have type A. Applying $\text{ftn}^{\uparrow\text{List}}$ flattens the inner lists and produces:

$$p \triangleright \text{ftn}^{\uparrow\text{List}} = [[x_{11}, x_{12}, ..., x_{21}, x_{22}, ...], [y_{11}, y_{12}, ..., y_{21}, y_{22}, ...], ...] \quad .$$

Flattening that result gives a list of all values x_{ij}, y_{ij}, ..., in the order they appear in p:

$$p \triangleright \text{ftn}^{\uparrow\text{List}} \triangleright \text{ftn} = [x_{11}, x_{12}, ..., x_{21}, x_{22}, ..., y_{11}, y_{12}, ..., y_{21}, y_{22}, ..., ...] \quad .$$

Applying $\text{ftn}^{\text{List}^A}$ to p will flatten the outer lists:

$$p \triangleright \text{ftn}^{\text{List}^A} = [[x_{11}, x_{12}, ...], [x_{21}, x_{22}, ...], ..., [y_{11}, y_{12}, ...], [y_{21}, y_{22}, ...], ...] \quad .$$

Flattening that value results in $p \triangleright \text{ftn}^{\text{List}^A} \triangleright \text{ftn} = [x_{11}, x_{12}, ..., x_{21}, x_{22}, ..., y_{11}, y_{12}, ..., y_{21}, y_{22}, ..., ...]$. This is exactly the same as $p \triangleright \text{ftn}^{\uparrow\text{List}} \triangleright \text{ftn}$: namely, the list of all values in the order they appear in p.

Let us now prove the associativity law by an explicit derivation. Using the recursive type definition $\text{List}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times \text{List}^A$, we can define `flatten` as a recursive function:

$$\text{ftn}^A \stackrel{\text{def}}{=} \begin{array}{|c||c|} & \mathbb{1} + \text{List}^A \times \text{List}^{\text{List}^A} \\ \hline \mathbb{1} & 1 \to 1 + \mathbb{0} \\ \text{List}^A \times \text{List}^{\text{List}^A} & h^{:\text{List}^A} \times t^{:\text{List}^{\text{List}^A}} \to h{+\!\!+}(t \triangleright \overline{\text{ftn}}) \end{array} \quad ,$$

where we have used the function `concat` (denoted $+\!\!+$) whose associativity property was derived in Statement 8.5.2.1. The operation of lifting to the `List` functor is defined for arbitrary functions f by:

$$(f^{:A\to B})^{\uparrow\text{List}} \stackrel{\text{def}}{=} \left|\begin{array}{c||cc} & \mathbb{1} & B\times\text{List}^B \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A\times\text{List}^A & \mathbb{0} & h^{:A}\times t^{:\text{List}^A}\to f(h)\times\left(t\triangleright\overline{f^{\uparrow\text{List}}}\right) \end{array}\right| .$$

The proof is by induction; the overline denotes recursive function calls, for which all laws hold by inductive assumption. We write the code for $\text{ftn}^{\uparrow\text{List}}$ and $\text{ftn}^{\text{List}^A}$, both of type $\text{List}^{\text{List}^{\text{List}^A}}\to\text{List}^{\text{List}^A}$:

$$\text{ftn}^{\uparrow\text{List}} = \left|\begin{array}{c||cc} & \mathbb{1} & \text{List}^A\times\text{List}^{\text{List}^A} \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ \text{List}^{\text{List}^A}\times\text{List}^{\text{List}^{\text{List}^A}} & \mathbb{0} & h^{:\text{List}^{\text{List}^A}}\times t^{:\text{List}^{\text{List}^{\text{List}^A}}}\to\text{ftn}(h)\times\left(t\triangleright\overline{\text{ftn}^{\uparrow\text{List}}}\right) \end{array}\right| ,$$

$$\text{ftn}^{\text{List}^A} = \left|\begin{array}{c||c} & \mathbb{1}+\text{List}^A\times\text{List}^{\text{List}^A} \\ \hline \mathbb{1} & 1\to 1+\mathbb{0} \\ \text{List}^{\text{List}^A}\times\text{List}^{\text{List}^{\text{List}^A}} & h^{:\text{List}^{\text{List}^A}}\times t^{:\text{List}^{\text{List}^{\text{List}^A}}}\to h+\!\!+\left(t\triangleright\overline{\text{ftn}}^{\text{List}^A}\right) \end{array}\right| .$$

It remains to compute $\text{ftn}^{\uparrow\text{List}}\,{\scriptstyle\fatsemi}\,\text{ftn}^A$ and $\text{ftn}^{\text{List}^A}\,{\scriptstyle\fatsemi}\,\text{ftn}^A$ via matrix composition. The first one is quick:

$$\text{ftn}^{\uparrow\text{List}}\,{\scriptstyle\fatsemi}\,\text{ftn}^A = \left\|\begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h\times t\to\text{ftn}(h)\times\left(t\triangleright\text{ftn}^{\uparrow\text{List}}\right) \end{array}\right\| \,{\scriptstyle\fatsemi}\, \left\|\begin{array}{c} 1\to 1+\mathbb{0} \\ h\times t\to h+\!\!+(t\triangleright\overline{\text{ftn}}) \end{array}\right\|$$

$$= \left|\begin{array}{c||c} & \mathbb{1}+A\times\text{List}^A \\ \hline \mathbb{1} & 1\to 1+\mathbb{0} \\ \text{List}^{\text{List}^A}\times\text{List}^{\text{List}^{\text{List}^A}} & h\times t\to\text{ftn}(h)+\!\!+\left(t\triangleright\text{ftn}^{\uparrow\text{List}}\triangleright\overline{\text{ftn}}\right) \end{array}\right| .$$

The second calculation gets stuck because the code matrix for $\text{ftn}^{\text{List}^A}$ has an unsplit column:

$$\text{ftn}^{\text{List}^A}\,{\scriptstyle\fatsemi}\,\text{ftn}^A = \left|\begin{array}{c||c} & \mathbb{1}+\text{List}^A\times\text{List}^{\text{List}^A} \\ \hline \mathbb{1} & 1\to 1+\mathbb{0} \\ \text{List}^{\text{List}^A}\times\text{List}^{\text{List}^{\text{List}^A}} & h\times t\to h+\!\!+\left(t\triangleright\overline{\text{ftn}}^{\text{List}^A}\right) \end{array}\right| \,{\scriptstyle\fatsemi}\, \left\|\begin{array}{c} 1\to 1+\mathbb{0}^{:A\times\text{List}^A} \\ h\times t\to h+\!\!+(t\triangleright\overline{\text{ftn}}) \end{array}\right\| .$$

We cannot split the column because the expression $h+\!\!+\left(t\triangleright\overline{\text{ftn}}^{\text{List}^A}\right)$ may evaluate to any part of the disjunction $\mathbb{1}+\text{List}^A\times\text{List}^{\text{List}^A}$ depending on the values of h and t. We are unable to compose the result with the second matrix unless we identify that part of the disjunction and then substitute into the first or the second column of that matrix. A general recipe in such situations is to perform additional pattern matching on the arguments of the function and to substitute the results into the matrix. Consider the value of $h : \text{List}^{\text{List}^A}$, which can be an empty list, $h = 1+\mathbb{0}$, or a product, $h = \mathbb{0}+g^{:\text{List}^A}\times k^{:\text{List}^{\text{List}^A}}$. Substitute these possibilities into the matrix expression for $\text{ftn}^{\text{List}^A}\,{\scriptstyle\fatsemi}\,\text{ftn}^A$:

$$\begin{aligned} &\text{with } h = 1+\mathbb{0}: && (\mathbb{0}+(1+\mathbb{0})\times t)\triangleright\text{ftn}^{\text{List}^A}\,{\scriptstyle\fatsemi}\,\text{ftn}^A = \underline{(1+\mathbb{0})+\!\!+}\left(t\triangleright\overline{\text{ftn}^{\text{List}^A}}\,{\scriptstyle\fatsemi}\,\text{ftn}^A\right) \\ &\text{concatenate with empty list}: && = t\triangleright\overline{\text{ftn}^{\text{List}^A}}\triangleright\text{ftn}^A \quad . \end{aligned}$$

Substituting this value h into $\text{ftn}^{\uparrow\text{List}} ⨾ \text{ftn}^A$, we get:

$$(\mathbb{0} + (1 + \mathbb{0}) \times t) \triangleright \text{ftn}^{\uparrow\text{List}} ⨾ \text{ftn}^A = t \triangleright \overline{\text{ftn}^{\overline{\uparrow\text{List}}}} \triangleright \overline{\text{ftn}} \quad .$$

Now we just need to show that:

$$t \triangleright \overline{\text{ftn}^{\text{List}^A}} \triangleright \text{ftn}^A \stackrel{?}{=} t \triangleright \overline{\text{ftn}^{\overline{\uparrow\text{List}}}} \triangleright \overline{\text{ftn}} \quad .$$

This holds by the inductive assumption.

It remains to examine the second possibility, $h = \mathbb{0} + g \times k$:

$$\begin{aligned}
\text{with } h = \mathbb{0} + g \times k : \quad & (\mathbb{0} + (\mathbb{0} + g \times k) \times t) \triangleright \text{ftn}^{\text{List}^A} ⨾ \text{ftn}^A = \big((\mathbb{0} + g \times k) +\!\!+ \big(t \triangleright \overline{\text{ftn}^{\text{List}^A}}\big)\big) \triangleright \text{ftn}^A \\
\text{code of } +\!\!+ : \quad & = \big(\mathbb{0} + g \times \big(k +\!\!+ \big(t \triangleright \overline{\text{ftn}^{\text{List}^A}}\big)\big)\big) \triangleright \text{ftn}^A \\
\text{code of } \text{ftn}^A : \quad & = g +\!\!+ \big(k +\!\!+ \big(t \triangleright \overline{\text{ftn}^{\text{List}^A}}\big)\big) \triangleright \overline{\text{ftn}} \\
\text{Exercise 10.2.9.11} : \quad & = \underline{g +\!\!+ \big(k \triangleright \overline{\text{ftn}}\big)} +\!\!+ \big(t \triangleright \overline{\text{ftn}^{\text{List}^A}} \triangleright \overline{\text{ftn}}\big) = (\mathbb{0} + g \times k) \triangleright \overline{\text{ftn}} +\!\!+ \big(\underline{t \triangleright \overline{\text{ftn}^{\text{List}^A}} \triangleright \overline{\text{ftn}}}\big) \\
\text{inductive assumption} : \quad & = \text{ftn}\,(h) +\!\!+ \big(t \triangleright \overline{\text{ftn}^{\uparrow\text{List}}} \triangleright \overline{\text{ftn}}\big) \quad .
\end{aligned}$$

This is the same as the result of substituting $\mathbb{0} + h \times t$ into $\text{ftn}^{\uparrow\text{List}} ⨾ \text{ftn}^A$.

Example 10.2.3.4 Consider the `List` type constructor with a `flatten` method that concatenates the nested lists in *reverse* order. Show that this implementation violates the associativity law of `flatten`.

Solution Apply both sides of the law to the nested list $p \stackrel{\text{def}}{=} [[[a,b],[c,d]],[[e,f],[g,h]]]$:

$$\begin{aligned}
& p \triangleright \text{ftn}^{\uparrow\text{List}} = [[c,d,a,b],[g,h,e,f]] \quad , \\
\text{left-hand side} : \quad & p \triangleright \text{ftn}^{\uparrow\text{List}} \triangleright \text{ftn} = [g,h,e,f,c,d,a,b] \quad , \\
& p \triangleright \text{ftn}^{\text{List}^A} = [[e,f],[g,h],[a,b],[c,d]] \quad , \\
\text{right-hand side} : \quad & p \triangleright \text{ftn}^{\text{List}^A} \triangleright \text{ftn} = [c,d,a,b,g,h,e,f] \quad .
\end{aligned}$$

The results are not the same. So, p is a counterexample that refutes the associativity law.

Example 10.2.3.5 Show that the `Writer` semimonad, $F^A \stackrel{\text{def}}{=} A \times W$, is lawful if W is a semigroup.

Solution The type signature of `flatten` is $\text{ftn} : (A \times W) \times W \to A \times W$, and the code is:

$$\text{ftn} : (a \times w_1) \times w_2 \to a \times (w_1 \oplus w_2) \quad ,$$

where $\oplus$ is the binary operation of the semigroup W. The lifting to F is $f^{\uparrow F} = a \times w \to (a \triangleright f) \times w$, so:

$$\text{ftn}^{\uparrow F} = ((a \times w_1) \times w_2) \times w_3 \to (((a \times w_1) \times w_2) \triangleright \text{ftn}) \times w_3 = (a \times (w_1 \oplus w_2)) \times w_3 \quad .$$

To verify the associativity law, it is convenient to substitute a value $((a \times w_1) \times w_2) \times w_3$ of type $F^{F^{F^A}}$ into both sides of the law:

$$\begin{aligned}
& (((a \times w_1) \times w_2) \times w_3) \triangleright \text{ftn}^{\uparrow F} \triangleright \text{ftn} = ((a \times (w_1 \oplus w_2)) \times w_3) \triangleright \text{ftn} = a \times ((w_1 \oplus w_2) \oplus w_3) \quad , \\
& (((a \times w_1) \times w_2) \times w_3) \triangleright \text{ftn}^{F^A} \triangleright \text{ftn} = ((a \times w_1) \times (w_2 \oplus w_3)) \triangleright \text{ftn} = a \times (w_1 \oplus (w_2 \oplus w_3)) \quad .
\end{aligned}$$

The operation $\oplus$ is associative since W is a semigroup. So, both sides of the law are equal.

Example 10.2.3.6 Consider the functor $F^A \stackrel{\text{def}}{=} A \times V \times V$ with the `flatten` method defined by:

$$\text{ftn} \stackrel{\text{def}}{=} \big(a^{:A} \times u_1^{:V} \times u_2^{:V}\big) \times v_1^{:V} \times v_2^{:V} \to a \times v_2 \times v_1 \quad .$$

Show that this definition violates the associativity law of `flatten`.

Solution Substitute a value of type $F^{F^{F^A}}$ into both sides of the law and get unequal results:

$$(((a \times u_1 \times u_2) \times v_1 \times v_2) \times w_1 \times w_2) \triangleright \text{ftn}^{\uparrow F} \triangleright \text{ftn} = ((a \times v_2 \times v_1) \times w_1 \times w_2) \triangleright \text{ftn} = a \times w_2 \times w_1 \quad ,$$
$$(((a \times u_1 \times u_2) \times v_1 \times v_2) \times w_1 \times w_2) \triangleright \text{ftn}^{F^A} \triangleright \text{ftn} = ((a \times u_1 \times u_2) \times w_2 \times w_1) \triangleright \text{ftn} = a \times w_1 \times w_2 \quad .$$

Had the code not exchanged the order of w_1 and w_2, the law would have held.

10.2.4 From semimonads to monads: Motivation for the identity laws

Semimonads are heuristically viewed as values with a special "computational effect". Semimonad-valued computations can be composed using the `flatMap` method, which will "merge" the effects associatively. It is generally useful to be able to create values with an "empty effect", such that merging the empty effect leaves other effects unchanged. A monad M is a semimonad that has a method for creating values with "empty effect". That method is called `pure` (notation pu_M):

```
def pure[A](a: A): M[A]
```

$$\text{pu}_M^A : A \to M^A \quad .$$

To get intuition about the properties of a vaguely defined "empty effect", again consider nested iteration over arrays. The "empty effect" is an array containing *one* element, because an iteration of such an array goes over a single value, which is equivalent to no iteration. In a functor block, this intuition says that a source line with an "empty effect", `y <- pure(x)`, should be equivalent to just `y = x`. This line must occur either before or after another source line, for instance:

```
result1 = for {
    ... // Some code, then:
    y <- pure(x)   // "Empty effect" with x: A.
    z <- someArray(y)   // someArray: A => M[B]
     // Same as z <- pure(x).flatMap(someArray)
```

```
result2 = for {
    ... // Some code, then:
    y = x                // x: A
    z <- someArray(y)   // someArray: A => M[B]
    // Same as z <- someArray(x)
```

The equality of `result1` and `result2` gives the following law: for all $g^{:A \to M^B}$,

```
pure(x).flatMap(g) == g(x)
```

$$\text{left identity law of } M: \quad \text{pu}_M \,{}^{\circ}_{\circ}\, \text{flm}_M(g^{:A \to M^B}) = g \quad . \tag{10.7}$$

The second possibility is that an empty effect comes *after* a source line:

```
result1 = for {
    x <- someArray   // someArray: M[A]
    y <- pure(x)     // Empty effect with x: A.
// Same as y <- someArray.flatMap(x => pure(x))
```

```
result2 = for {
    x <- someArray   // someArray: M[A]
    y = x
    // Same as y <- someArray
```

Then the equality of `result1` and `result2` gives the law:

```
g.flatMap(pure) == g
```

$$\text{right identity law of } M: \quad \text{flm}_M(\text{pu}_M) = \text{id}^{:M^A \to M^A} \quad . \tag{10.8}$$

A typeclass for monads and a law-checking test function can be defined by:

```
abstract class Monad[F[_]: Functor : Semimonad] {
  def pure[A](a: A): F[A]
}
def checkMonadIdentityLaws[F[_], A, B]()(implicit mf: Monad[F], sf: Semimonad[F],
      aa: Arbitrary[A], af: Arbitrary[F[A]], ab: Arbitrary[A => F[B]]) = {
    forAll { (x: A, g: A => F[B]) =>
      mf.pure(x).flatMap(g) shouldEqual g(x)   // Left identity law.
    }
    forAll { (fa: F[A]) =>
      fa.flatMap(mf.pure[A]) shouldEqual fa   // Right identity law.
    }
}
```

Note that `pure` is the same method as in the `Pointed` typeclass (Sections 8.3.5–8.3.6). So, we could say that a monad is a pointed semimonad whose `pure` method obeys the two identity laws (10.7)–(10.8). Although the `pure` method can be replaced by a simpler "wrapped unit" value (wu_M), derivations turn out to be easier when using `pure`.

The `Pointed` typeclass requires the `pure` method to satisfy the naturality law (8.8). A full monad's `pure` method must also satisfy that naturality law, in addition to the two identity laws.

Just as some useful semigroups are not monoids, there exist some useful semimonads that are not full monads. A simple example is the `Writer` semimonad $F^A \stackrel{\text{def}}{=} A \times W$ whose type W is a semigroup but not a monoid (see Exercise 10.2.9.1).

10.2.5 The monad identity laws in terms of `pure` and `flatten`

Since the laws of semimonads are simpler when formulated via the `flatten` method, let us convert the identity laws to that form. We use the code for `flatMap` in terms of `flatten`:

$$\text{flm}_M(f^{:A\to M^B}) = f^{\uparrow M} \fatsemi \text{ftn}_M \quad .$$

Begin with the left identity law of `flatMap`, written as:

$$\text{pu}_M \fatsemi \text{flm}_M(f) = f \quad .$$

$$\begin{tikzcd} & M^{M^A} \arrow[dr, "\text{ftn}^A"] & \\ M^A \arrow[ur, "\text{pu}^{M^A}"] \arrow[rr, "\text{id}"'] & & M^A \end{tikzcd}$$

Since this law holds for arbitrary f, we can set $f \stackrel{\text{def}}{=} \text{id}$ and get:

$$\text{pu}_M \fatsemi \text{ftn}_M = \text{id}^{:M^A\to M^A} \quad . \tag{10.9}$$

This is the **left identity law** of `flatten`. Conversely, if Eq. (10.9) holds, we can compose both sides with an arbitrary function $f^{:A\to M^B}$ and recover the left identity law of `flatMap` (Exercise 10.2.9.2).

The **right identity law** of `flatten` is written as:

$$\begin{tikzcd} & M^{M^A} \arrow[dr, "\text{ftn}^A"] & \\ M^A \arrow[ur, "(\text{pu}^A)^{\uparrow M}"] \arrow[rr, "\text{id}"'] & & M^A \end{tikzcd}$$

$$\text{flm}_M(\text{pu}_M) = \text{pu}_M^{\uparrow M} \fatsemi \text{ftn}_M \stackrel{!}{=} \text{id} \quad . \tag{10.10}$$

In the next section, we will see a reason why these laws have their names.

10.2.6 Monad laws in terms of Kleisli functions

A **Kleisli function** is a function with type signature $A \to M^B$ where M is a monad. We first encountered Kleisli functions in Section 9.2.3 when deriving the laws of filterable functors using the `liftOpt` method. At that point, M was the simple `Option` monad. We found that functions of type $A \to \mathbb{1} + B$ can be composed using the Kleisli composition denoted by $\diamond_{\text{Opt}}$ (see page 311). Later, Section 9.4.2 stated the general properties of Kleisli composition. We will now show that the Kleisli composition gives a useful way of formulating the laws of a monad.

The Kleisli composition operation for a monad M, denoted $\diamond_M$, is a function with type signature:

$$\diamond_M : (A \to M^B) \to (B \to M^C) \to A \to M^C \quad .$$

This resembles the forward composition of ordinary functions, $(\fatsemi) : (A \to B) \to (B \to C) \to A \to C$, except for different types of functions. If M is a monad, the implementation of $\diamond_M$ is:

```
def <>[M[_]: Monad, A,B,C](f: A => M[B], g: B => M[C]): A => M[C] =
  { x => f(x).flatMap(g) }
```

$$f \diamond_M g \stackrel{\text{def}}{=} f \fatsemi \text{flm}_M(g) \quad . \tag{10.11}$$

The Kleisli composition can be equivalently expressed by a functor block code as:

```
def <>[M[_]: Monad, A,B,C](f: A => M[B], g: B => M[C]): A => M[C] = { x =>
  for {
    y <- f(x)
    z <- g(y)
  } yield z
}
```

This example shows that Kleisli composition is a basic part of functor block code: it expresses the chaining of two consecutive "source" lines.

Let us now derive the laws of Kleisli composition $\diamond_M$, assuming that the monad laws hold for M.

Statement 10.2.6.1 For a lawful monad M, the Kleisli composition $\diamond_M$ satisfies the identity laws:

$$\text{left identity law of } \diamond_M : \quad \text{pu}_M \diamond_M f = f \quad , \quad \forall f^{:A\to M^B} \quad , \tag{10.12}$$

$$\text{right identity law of } \diamond_M : \quad f \diamond_M \text{pu}_M = f \quad , \quad \forall f^{:A\to M^B} \quad . \tag{10.13}$$

Proof We may assume that Eqs. (10.7)–(10.8) hold. Using the definition (10.11), we find:

$$\begin{aligned}
\text{left identity law of } \diamond_M\text{, should equal } f : \quad & \text{pu}_M \diamond_M f = \underline{\text{pu}_M ⨾ \text{flm}_M(f)} \\
\text{use Eq. (10.7)} : \quad & = f \quad , \\
\text{right identity law of } \diamond_M\text{, should equal } f : \quad & f \diamond_M \text{pu}_M = f ⨾ \underline{\text{flm}_M(\text{pu}_M)} \\
\text{use Eq. (10.8)} : \quad & = f ⨾ \text{id} = f \quad .
\end{aligned}$$

The following statement and the identity law (10.8) show that `flatMap` can be viewed as a "lifting",

$$\text{flm}_M : (A \to M^B) \to (M^A \to M^B) \quad ,$$

from Kleisli functions $A \to M^B$ to M-lifted functions $M^A \to M^B$, except that Kleisli functions must be composed using $\diamond_M$, while pu_M plays the role of the Kleisli-identity function.

Statement 10.2.6.2 For a lawful monad M, the `flatMap` method satisfies the composition law:

$$\begin{tikzcd}
 & M^B \arrow[dr, "\text{flm}_M(g)"] & \\
M^A \arrow[ur, "\text{flm}_M(f)"] \arrow[rr, "\text{flm}_M(f \diamond_M g)"'] & & M^C
\end{tikzcd}$$

$$\text{flm}_M(f \diamond_M g) = \text{flm}_M(f) ⨾ \text{flm}_M(g) \quad .$$

Proof We may use Eq. (10.4) since M is a lawful monad. A direct calculation verifies the law:

$$\begin{aligned}
\text{expect to equal } \text{flm}_M(f) ⨾ \text{flm}_M(g) : \quad & \text{flm}_M(f \diamond_M g) = \text{flm}_M(f ⨾ \text{flm}_M(g)) \\
\text{use Eq. (10.4)} : \quad & = \text{flm}_M(f) ⨾ \text{flm}_M(g) \quad .
\end{aligned}$$

The following statement motivates calling Eq. (10.4) an "associativity" law.

Statement 10.2.6.3 For a lawful monad M, the Kleisli composition $\diamond_M$ satisfies the **associativity law**

$$(f \diamond_M g) \diamond_M h = f \diamond_M (g \diamond_M h) \quad , \quad \forall f^{:A\to M^B}, g^{:B\to M^C}, h^{:C\to M^D} \quad . \tag{10.14}$$

So, we may write $f \diamond_M g \diamond_M h$ unambiguously with no parentheses.

Proof Substitute Eq. (10.11) into both sides of the law:

$$\begin{aligned}
\text{left-hand side} : \quad & \underline{(f \diamond_M g)} \diamond_M h = (f ⨾ \text{flm}_M(g)) \underline{\diamond_M h} = f ⨾ \text{flm}_M(g) ⨾ \text{flm}_M(h) \quad , \\
\text{right-hand side} : \quad & \underline{f \diamond_M} (g \diamond_M h) = f ⨾ \underline{\text{flm}_M(g \diamond_M h)} \\
\text{use Statement 10.2.6.2} : \quad & = f ⨾ \text{flm}_M(g) ⨾ \text{flm}_M(h) \quad .
\end{aligned}$$

Both sides of the law are now equal.

We find that the properties of the operation $\diamond_M$ are similar to the identity and associativity properties of the function composition $f ⨾ g$ except for using pu_M instead of the identity function.[8]

Since the Kleisli composition describes the chaining of consecutive lines in functor blocks, its associativity means that multiple lines are chained unambiguously. For example, this code:

[8] It means that Kleisli functions satisfy the properties of morphisms of a category; see Section 9.4.3.

```
1 x => for {
2   y <- f(x)
3   z <- g(y)
4   t <- h(z)
5 } yield t
```

corresponds to the Kleisli composition:

$$(x \to f(x)) \diamond_M (y \to g(y)) \diamond_M (z \to h(z))$$

and does not need to specify whether lines 2 and 3 are chained before appending line 4, or lines 3 and 4 are chained before prepending line 2.

We will now prove that the Kleisli composition with its laws is equivalent to `flatMap` with *its* laws. In other words, we may equally well use the Kleisli composition when formulating the requirements for a functor M to be a monad.

Statement 10.2.6.4 The types of Kleisli composition $\diamond_M$ and of M's `flatMap` are equivalent:

$$f^{:A\to M^B} \diamond_M g^{:B\to M^C} = f \fatsemi \text{flm}_M(g) \quad , \qquad \text{flm}_M(f^{:A\to M^B}) = \text{id}^{:M^A\to M^A} \diamond_M f \quad , \tag{10.15}$$

provided that Eqs. (10.12)–(10.14) and the following additional law hold for $\diamond_M$:

$$\text{left naturality of } \diamond_M : \quad (f^{:A\to B} \fatsemi g^{:B\to M^C}) \diamond_M h^{:C\to M^D} = f \fatsemi (g \diamond_M h) \quad . \tag{10.16}$$

Note that this law makes parentheses unnecessary in the expression $f \fatsemi g \diamond_M h$.

Proof Equations (10.15) map $\diamond_M$ to flm_M and back. We have to show that these mappings are isomorphisms when the given laws hold. We proceed in two steps:

(1) Given an operation $\diamond_M$, we define flm_M and then a new operation $\diamond'_M$ using Eq. (10.15). We then need to prove that $\diamond'_M = \diamond_M$. Calculate using arbitrary functions $f^{:A\to M^B}$ and $g^{:B\to M^C}$:

$$\begin{aligned} \text{use Eq. (10.15)} : \quad & f \diamond'_M g = f \fatsemi \text{flm}_M(g) = f \fatsemi (\text{id}^{M^B} \diamond_M g) \\ \text{left naturality (10.16) of } \diamond_M : \quad & = (\underline{f \fatsemi \text{id}}) \diamond_M g = f \diamond_M g \quad . \end{aligned}$$

When $\diamond_M$ is defined via flm_M, the left naturality law (10.16) will hold because "$\fatsemi$" is associative,

$$(f \fatsemi g) \diamond_M h = (f \fatsemi g) \fatsemi \text{flm}_M(h) = f \fatsemi (g \fatsemi \text{flm}_M(h)) = f \fatsemi (g \diamond_M h) \quad .$$

(2) Given a function flm_M, we define $\diamond_M$ and then a new function flm'_M using Eq. (10.15). We then need to prove that $\text{flm}'_M = \text{flm}_M$. Calculate using an arbitrary function $f^{:A\to M^B}$:

$$\text{use Eq. (10.15)} : \quad \text{flm}'_M(f) = \text{id}^{M^A} \diamond_M f = \underline{\text{id}} \fatsemi \text{flm}_M(f) = \text{flm}_M(f) \quad .$$

We have already derived the laws of Kleisli composition from the laws of `flatMap`. We will now derive the converse statement. In this way, we will show that the Kleisli composition laws and the `flatMap` laws are fully equivalent.

Statement 10.2.6.5 If the Kleisli composition $\diamond_M$ obeys the laws (10.12)–(10.14), the corresponding `flatMap` method defined by Eq. (10.15) will satisfy the laws (10.2)–(10.4).

Proof To derive the identity laws of `flatMap`:

$$\begin{aligned} \text{left identity law} : \quad & \text{pu}_M \fatsemi \underline{\text{flm}_M(f)} = \underline{\text{pu}_M \fatsemi \text{id}} \diamond_M f = \underline{\text{pu}_M \diamond_M} f = f \quad , \\ \text{right identity law} : \quad & \text{flm}_M(\text{pu}_M) = \text{id} \underline{\diamond_M \text{pu}_M} = \text{id} \quad . \end{aligned}$$

To derive the associativity law (10.4) of `flatMap`, substitute the definition of flm_M into both sides:

$$\begin{aligned} \text{left-hand side} : \quad & \text{flm}_M(f \fatsemi \text{flm}_M(g)) = \text{id} \diamond_M (\underline{f \fatsemi \text{id}} \diamond_M g) = \text{id} \diamond_M (f \diamond_M g) \\ \text{associativity law (10.14)} : \quad & = (\text{id} \diamond_M f) \diamond_M g \quad , \\ \text{right-hand side} : \quad & \text{flm}_M(f) \fatsemi \text{flm}_M(g) = (\text{id} \diamond_M f) \underline{\fatsemi (\text{id}} \diamond_M g) \\ \text{left naturality (10.16) of } \diamond_M : \quad & = (\text{id} \diamond_M f) \underline{\fatsemi \text{id}} \diamond_M g = (\text{id} \diamond_M f) \diamond_M g \quad . \end{aligned}$$

Both sides of the law are now equal.

The two naturality laws of `flatMap` are equivalent to the three naturality laws of $\diamond_M$, but we omit those derivations.

10.2.7 Verifying the monad laws using Kleisli functions

When the monad laws are formulated via Kleisli composition, the intuition behind the laws becomes clearer: they are analogous to the identity and associativity laws of the function composition ($\fatsemi$). The price is that the type signatures become complicated. For instance, the associativity law (10.14) has four type parameters, while the corresponding law (10.6) for `flatten` has only one. For certain monads, however, a trick called **flipped Kleisli** makes direct proofs of laws much shorter. That trick applies to monads of a function type, such as the continuation and the state monads.

Statement 10.2.7.1 The continuation monad, $\text{Cont}^{R,A} \stackrel{\text{def}}{=} (A \to R) \to R$, satisfies all monad laws.

Proof Begin by writing the type of a Kleisli function corresponding to this monad:

$$A \to \text{Cont}^{R,B} = A \to (B \to R) \to R \quad .$$

This function type has two curried arguments. The first step of the flipped Kleisli technique is to change the types of the Kleisli functions by flipping their two curried arguments. We obtain:

$$(B \to R) \to A \to R \quad .$$

This type looks like a function of the form $K^B \to K^A$, where we temporarily defined $K^A \stackrel{\text{def}}{=} A \to R$. The remaining steps are to flip the arguments of pu_{Cont} and get a modified $\tilde{\text{pu}}_{\text{Cont}}$. Then, we modify the Kleisli composition $\diamond_{\text{Cont}}$ into $\tilde{\diamond}_{\text{Cont}}$. The flipped Kleisli functions are composed using $\tilde{\diamond}_{\text{Cont}}$. We will then prove the laws:

$$\tilde{\text{pu}}_{\text{Cont}} \tilde{\diamond}_{\text{Cont}} f = f \quad , \qquad f \,\tilde{\diamond}_{\text{Cont}} \tilde{\text{pu}}_{\text{Cont}} = f \quad , \qquad (f \,\tilde{\diamond}_{\text{Cont}} g) \,\tilde{\diamond}_{\text{Cont}} h = f \,\tilde{\diamond}_{\text{Cont}} (g \,\tilde{\diamond}_{\text{Cont}} h) \quad .$$

Flipping the arguments is an operation that maps functions to equivalent functions. So, if we prove the laws of identity and composition for flipped Kleisli functions, it will follow that the same laws hold for the original functions.

The original `pure` method is:

$$\text{pu}_{\text{Cont}} \stackrel{\text{def}}{=} a^{:A} \to f^{:A \to R} \to f(a) \quad .$$

We find that the flipped `pure` method is just an identity function:

$$\tilde{\text{pu}}_{\text{Cont}} \stackrel{\text{def}}{=} f^{:A \to R} \to a^{:A} \to f(a) = f^{:A \to R} \to f = \text{id}^{:(A \to R) \to A \to R} = \text{id}^{:K^A \to K^A} \quad .$$

This is a significant simplification. The flipped Kleisli composition $\tilde{\diamond}_{\text{Cont}}$ has the type signature:

$$f^{:K^B \to K^A} \,\tilde{\diamond}_{\text{Cont}}\, g^{:K^C \to K^B} = ???^{:K^C \to K^A} \quad .$$

There is only one implementation of that type signature, namely the backward composition:

$$f^{:K^B \to K^A} \,\tilde{\diamond}_{\text{Cont}}\, g^{:K^C \to K^B} \stackrel{\text{def}}{=} g \fatsemi f = f \circ g \quad .$$

So, this must be the code of the flipped Kleisli composition. It is now quick to verify the laws:

$$\begin{aligned}
&\text{left identity law :} && \tilde{\text{pu}}_{\text{Cont}} \tilde{\diamond}_{\text{Cont}} f = \text{id} \circ f = f \quad , \\
&\text{right identity law :} && f \,\tilde{\diamond}_{\text{Cont}} \tilde{\text{pu}}_{\text{Cont}} = f \circ \text{id} = f \quad , \\
&\text{associativity law :} && \left(f \,\tilde{\diamond}_{\text{Cont}} g\right) \tilde{\diamond}_{\text{Cont}} h = (f \circ g) \circ h = f \circ (g \circ h) = f \,\tilde{\diamond}_{\text{Cont}} \left(g \,\tilde{\diamond}_{\text{Cont}} h\right) \quad .
\end{aligned}$$

We could have avoided writing the last three lines by noticing that functions of types $K^B \to K^A$ automatically satisfy the laws of (backward) function composition (Section 4.2.2).

Statement 10.2.7.2 The state monad, $\text{State}^{S,A} \stackrel{\text{def}}{=} S \to A \times S$, satisfies all monad laws.

Proof Begin by writing the type of a Kleisli function corresponding to this monad:

$$A \to \text{State}^{S,B} = A \to S \to B \times S \quad .$$

An equivalent type is obtained by uncurrying the two arguments:

$$(A \to S \to B \times S) \cong (A \times S \to B \times S) = K^A \to K^B \quad ,$$

where we temporarily defined $K^A \stackrel{\text{def}}{=} A \times S$. This type looks like an ordinary function, which promises to simplify the proof. Uncurrying is an equivalence transformation. So, let us uncurry the arguments in all Kleisli functions and prove the laws in the "uncurried Kleisli" formulation. We need to uncurry the arguments in pu_{State} and $\diamond_{\text{State}}$ as well. Denote the resulting functions $\tilde{\text{pu}}_{\text{State}}$ and $\tilde{\diamond}_{\text{State}}$:

$$\tilde{\text{pu}}_{\text{State}} \stackrel{\text{def}}{=} a^{:A} \times s^{:S} \to a \times s = \text{id}^{:K^A \to K^A} \quad , \qquad f^{:K^A \to K^B} \tilde{\diamond}_{\text{State}} g^{:K^B \to K^C} \stackrel{\text{def}}{=} f \fatsemi g \quad .$$

We can see that the composition $f \fatsemi g$ implements the correct logic of the state monad: an initial state value $s^{:S}$ is updated by f and then passed to g.

We found that the uncurried Kleisli functions have simple types $K^A \to K^B$, and the operation $\tilde{\diamond}_{\text{State}}$ is just the ordinary composition of those functions. Since we already know that the laws of identity and associativity hold for ordinary functions (Section 4.2.2), the proof is finished.

For comparison, look at the type signatures of `flatten` for the state and continuation monads:

$$\text{ftn}_{\text{State}^{S,\bullet}} : (S \to (S \to A \times S) \times S) \to S \to A \times S \quad ,$$
$$\text{ftn}_{\text{Cont}^{R,\bullet}} : ((((A \to R) \to R) \to R) \to R) \to (A \to R) \to R \quad .$$

These type signatures are complicated and confusing to read. Direct proofs of the monad laws for these functions are much longer than the proofs of Statements 10.2.7.2–10.2.7.1. When a monad M has a function type, the Kleisli function $A \to M^B$ has two curried arguments. Flipping or uncurrying these arguments often produces an equivalent function that is easier to work with.

10.2.8 Constructions of semimonads and monads

We have seen different examples of well-known monads that were discovered by programmers working on specific tasks. Hoping to find systematically all possible monads, we will now apply structural analysis to semimonads and monads. For each type construction, we will prove rigorously that the monad laws hold.

Type parameters Three type constructions are based on using just type parameters: the constant functor ($\text{Const}^{Z,A} \stackrel{\text{def}}{=} Z$), the identity functor ($\text{Id}^A \stackrel{\text{def}}{=} A$), and the functor composition ($L^A \stackrel{\text{def}}{=} F^{G^A}$).

A constant functor $F^A \stackrel{\text{def}}{=} Z$ is a lawful semimonad because we can implement:

$$\text{ftn}_F = \text{id}^{:Z \to Z} \quad .$$

An identity function will always satisfy the laws. To obtain a full monad, we need to implement:

$$\text{pu}_F : A \to Z \quad .$$

This is possible only if we have a default value z_0 of type Z. Assuming that, we set $\text{pu}_F \stackrel{\text{def}}{=} _ \to z_0$ and check the identity laws:

$$\text{left identity law of } F: \quad \text{pu}_F^{:F^A \to F^{F^A}} \fatsemi \text{ftn}_F = \text{pu}_F \fatsemi \text{id} = \text{pu}_F \stackrel{?}{=} \text{id} \quad .$$

The function pu_F is a constant function that always returns z_0; and yet it must be equal to the identity function. This law can be satisfied only if the identity function of type $Z \to Z$ always returns the same value z_0. It follows that z_0 is the only available value of the type Z, which means $Z \cong \mathbb{1}$. In that

case, all functions become constants returning 1, and the laws are trivially satisfied. We conclude that the only case when a constant functor is a lawful monad is when $L^A \stackrel{\text{def}}{=} \mathbb{1}$ (the constant `Unit` type).

The identity functor $\text{Id}^A \stackrel{\text{def}}{=} A$ is a monad: its `pure` and `flatten` methods are identity functions.

Functor composition $F \circ G$ is not guaranteed to produce monads even if F and G are both monads. A counterexample can be found by taking $F^A \stackrel{\text{def}}{=} Z + A$ and $G^A \stackrel{\text{def}}{=} R \to A$; the composition $L^A \stackrel{\text{def}}{=} Z + (R \to A)$ is not even a semimonad.

Statement 10.2.8.1 The functor $L^A \stackrel{\text{def}}{=} Z + (R \to A)$, where R and Z are fixed but arbitrary types, cannot have a `flatten` method.

Proof The type signature of `flatten` is:

$$\text{ftn}_L : Z + (R \to Z + (R \to A)) \to Z + (R \to A) \quad .$$

A fully parametric implementation is impossible: we would need to compute either a value of type Z or of type $R \to A$ from an argument that may have either type Z or type $R \to Z + (R \to A)$. When the argument has the latter type, it is impossible to compute either a value of type Z or a value of type $R \to A$, because different values may be returned for different values of R, and we do not have any known values of type R available.

Products The product construction works for semimonads as well as for monads.

Statement 10.2.8.2 Given two semimonads F^A and G^A, the functor $L^A \stackrel{\text{def}}{=} F^A \times G^A$ is a semimonad. If both F^A and G^A are monads then L^A is also a monad.

Proof Begin by defining the `flatten` method for the semimonad L via the `flatten` methods of F and G. The `flatten` method needs to transform a value of type $L^{L^A} = F^{F^A \times G^A} \times G^{F^A \times G^A}$ into a value of type $F^A \times G^A$. Since F and G are functors, we can extract F^{F^A} out of $F^{F^A \times G^A}$ by lifting the standard projection function π_1:

$$\pi_1^{\uparrow F} : F^{F^A \times G^A} \to F^{F^A} \quad .$$

Then we use F's `flatten` method to obtain a value of type F^A. In a similar way, we transform a value of type $G^{F^A \times G^A}$ into G^A. The resulting code is:

```
def flatten_L[A]: (F[(F[A], G[A])], G[(F[A], G[A])]) => (F[A], G[A]) = { case (fla, gla) =>
  val ffa: F[F[A]] = fla.map(_._1)
  val gga: G[G[A]] = gla.map(_._2)
  (flatten_F(ffa), flatten_G(gga))
}
```

$$\text{ftn}_L \stackrel{\text{def}}{=} f^{:F^{F^A \times G^A}} \times g^{:G^{F^A \times G^A}} \to \left(f \triangleright \pi_1^{\uparrow F} \triangleright \text{ftn}_F\right) \times \left(g \triangleright \pi_2^{\uparrow G} \triangleright \text{ftn}_G\right) = \left(\pi_1^{\uparrow F} \fatsemi \text{ftn}_F\right) \boxtimes \left(\pi_2^{\uparrow G} \fatsemi \text{ftn}_G\right) \quad . \tag{10.17}$$

To verify the associativity law, we need to write the code of $\text{ftn}_L^{\uparrow L}$:

$$(h^{:A \to B})^{\uparrow L} = h^{\uparrow F} \boxtimes h^{\uparrow G} \quad , \qquad \text{ftn}_L^{\uparrow L} = \text{ftn}_L^{\uparrow F} \boxtimes \text{ftn}_L^{\uparrow G} \quad .$$

Substitute these definitions into the associativity law:

$$\begin{aligned}
\text{left-hand side}: \quad & \text{ftn}_L^{\uparrow L} \fatsemi \text{ftn}_L = \left(\text{ftn}_L^{\uparrow F} \boxtimes \text{ftn}_L^{\uparrow G}\right) \fatsemi \left(\left(\pi_1^{\uparrow F} \fatsemi \text{ftn}_F\right) \boxtimes \left(\pi_2^{\uparrow G} \fatsemi \text{ftn}_G\right)\right) \\
\text{use Eq. (7.2)}: \quad & = \left(\text{ftn}_L^{\uparrow F} \fatsemi \pi_1^{\uparrow F} \fatsemi \text{ftn}_F\right) \boxtimes \left(\text{ftn}_L^{\uparrow G} \fatsemi \pi_2^{\uparrow G} \fatsemi \text{ftn}_G\right) \quad , \\
\text{right-hand side}: \quad & \text{ftn}_L \fatsemi \text{ftn}_L = \left(\left(\pi_1^{\uparrow F} \fatsemi \text{ftn}_F\right) \boxtimes \left(\pi_2^{\uparrow G} \fatsemi \text{ftn}_G\right)\right) \fatsemi \left(\left(\pi_1^{\uparrow F} \fatsemi \text{ftn}_F\right) \boxtimes \left(\pi_2^{\uparrow G} \fatsemi \text{ftn}_G\right)\right) \\
\text{use Eq. (7.2)}: \quad & = \left(\pi_1^{\uparrow F} \fatsemi \text{ftn}_F \fatsemi \pi_1^{\uparrow F} \fatsemi \text{ftn}_F\right) \boxtimes \left(\pi_2^{\uparrow G} \fatsemi \text{ftn}_G \fatsemi \pi_2^{\uparrow G} \fatsemi \text{ftn}_G\right) \quad .
\end{aligned}$$

The only given information about ftn_F and ftn_G is that they obey their associativity laws; e.g.,

$$\text{ftn}_F^{\uparrow F} \fatsemi \text{ftn}_F = \text{ftn}_F \fatsemi \text{ftn}_F \quad .$$

In order to use this law, we need to move the two functions ftn_F next to each other in the expressions

$$(\text{ftn}_L^{\uparrow F} ⨾ \pi_1^{\uparrow F} ⨾ \text{ftn}_F) \quad \text{and} \quad (\pi_1^{\uparrow F} ⨾ \text{ftn}_F ⨾ \pi_1^{\uparrow F} ⨾ \text{ftn}_F) \quad ,$$

hoping to show that these expressions are equal. We begin with the first of those expressions:

$$\begin{aligned}
& \underline{\text{ftn}_L^{\uparrow F} ⨾ \pi_1^{\uparrow F}} ⨾ \text{ftn}_F = (\underline{\text{ftn}_L ⨾ \pi_1})^{\uparrow F} ⨾ \text{ftn}_F \\
\text{left projection law (7.3)}: \quad & = (\pi_1 ⨾ \pi_1^{\uparrow F} ⨾ \text{ftn}_F)^{\uparrow F} ⨾ \text{ftn}_F = \pi_1^{\uparrow F} ⨾ \pi_1^{\uparrow F \uparrow F} ⨾ \underline{\text{ftn}_F^{\uparrow F} ⨾ \text{ftn}_F} \\
\text{associativity law of } F: \quad & = \pi_1^{\uparrow F} ⨾ \underline{\pi_1^{\uparrow F \uparrow F} ⨾ \text{ftn}_F} ⨾ \text{ftn}_F \\
\text{naturality law of } \text{ftn}_F: \quad & = \pi_1^{\uparrow F} ⨾ \underline{\text{ftn}_F ⨾ \pi_1^{\uparrow F}} ⨾ \text{ftn}_F \quad .
\end{aligned}$$

Both expressions are now the same. An analogous derivation shows that:

$$\text{ftn}_L^{\uparrow G} ⨾ \pi_2^{\uparrow G} ⨾ \text{ftn}_G = \pi_2^{\uparrow G} ⨾ \text{ftn}_G ⨾ \pi_2^{\uparrow G} ⨾ \text{ftn}_G \quad .$$

So, both sides of the associativity law are equal.

Now we assume that F and G are monads with given `pure` methods pu_F and pu_G. We define:

$$\text{pu}_L \stackrel{\text{def}}{=} a^{:A} \to \text{pu}_F(a) \times \text{pu}_G(a) = \Delta ⨾ (\text{pu}_F \boxtimes \text{pu}_G) \quad .$$

Assuming that identity laws hold for F and G, we can now verify the identity laws for L:

$$\begin{aligned}
\text{left identity law of } L: \quad & \text{pu}_L ⨾ \text{ftn}_L = \Delta ⨾ (\text{pu}_F \boxtimes \text{pu}_G) ⨾ ((\pi_1^{\uparrow F} ⨾ \text{ftn}_F) \boxtimes (\pi_2^{\uparrow G} ⨾ \text{ftn}_G)) \\
\text{use Eq. (7.2)}: \quad & = \Delta ⨾ ((\underline{\text{pu}_F ⨾ \pi_1^{\uparrow F}} ⨾ \text{ftn}_F) \boxtimes (\underline{\text{pu}_G ⨾ \pi_2^{\uparrow G}} ⨾ \text{ftn}_G)) \\
\text{naturality of } F, G: \quad & = \Delta ⨾ ((\pi_1 ⨾ \underline{\text{pu}_F ⨾ \text{ftn}_F}) \boxtimes (\pi_2 ⨾ \underline{\text{pu}_G ⨾ \text{ftn}_G})) \\
\text{identity laws of } F, G: \quad & = \Delta ⨾ (\pi_1 \times \pi_2) = \text{id} \quad , \\
\text{right identity law of } L: \quad & \text{pu}_L^{\uparrow L} ⨾ \text{ftn}_L = ((\Delta ⨾ (\text{pu}_F \boxtimes \text{pu}_G))^{\uparrow F} \boxtimes (\Delta ⨾ (\text{pu}_F \boxtimes \text{pu}_G))^{\uparrow G}) ⨾ \text{ftn}_L \\
\text{use Eq. (7.2)}: \quad & = ((\underline{\Delta ⨾ (\text{pu}_F \boxtimes \text{pu}_G))^{\uparrow F} ⨾ \pi_1^{\uparrow F}} ⨾ \text{ftn}_F) \boxtimes ((\underline{\Delta ⨾ (\text{pu}_F \boxtimes \text{pu}_G))^{\uparrow G} ⨾ \pi_2^{\uparrow G}} ⨾ \text{ftn}_G) \\
\text{projection laws (7.3)}: \quad & = ((\underline{\Delta ⨾ \pi_1} ⨾ \text{pu}_F)^{\uparrow F} ⨾ \text{ftn}_F) \boxtimes ((\underline{\Delta ⨾ \pi_2} ⨾ \text{pu}_G)^{\uparrow G} ⨾ \text{ftn}_G) \\
\text{identity laws (7.1)}: \quad & = (\underline{\text{pu}_F^{\uparrow F} ⨾ \text{ftn}_F}) \boxtimes (\underline{\text{pu}_G^{\uparrow G} ⨾ \text{ftn}_G}) = \text{id} \boxtimes \text{id} = \text{id} \quad .
\end{aligned}$$

Let us build some intuition about how the product of two monads works in practice. A simple example is the product of two identity monads, $L^A \stackrel{\text{def}}{=} A \times A$. This type constructor is a monad whose `flatten` function is defined by

```
type Pair[A] = (A, A)
def flatten[A]: Pair[Pair[A]] => Pair[A] = { case ((a, b), (c, d)) => (a, d) }
```

A sample calculation shows that "nested iterations" apply functions element by element:

```
final case class P[A](x: A, y: A) {
  def map[B](f: A => B): P[B] = P(f(x), f(y))
  def flatMap[B](f: A => P[B]): P[B] = P(f(x).x, f(y).y)
}

scala> for {
  x <- P(1, 10)
  y <- P(2, 20)
  z <- P(3, 30)
} yield x + y + z       // The result is P(1 + 2 + 3, 10 + 20 + 30).
res0: P[Int] = P(6, 60)
```

Note that the pair type $A \times A$ is equivalent to the function type $\mathbb{2} \to A$ (in Scala, `Boolean => A`). We know that $\mathbb{2} \to A$ is a `Reader` monad. One can check that the implementation of `flatten` for the `Reader` monad $\mathbb{2} \to A$ is equivalent to the code of `flatten` for $L^A \stackrel{\text{def}}{=} A \times A$ as shown above.

Now consider the product $F^A \times G^A$ with arbitrary monads F and G that represent some effects. Then a Kleisli function of type $A \to F^B \times G^B$ contains both effects. How does the monad L combine two effects when we compose two such Kleisli functions? We see from the code of L's `flatten` that the first effect in F is combined with the second effect in F, and the first effect in G is combined with the second effect in G. The part F^{G^A} from $F^{F^A \times G^A}$ is discarded by `flatten`, leaving only F^{F^A}, i.e., a combination of two F-effects. Also, G^{F^A} is discarded from $G^{F^A \times G^A}$, leaving only G^{G^A}. As an example, consider this code:

```
val result: (F[C], G[C]) = for {
  x <- (fa, ga)          // Assume fa: F[A], ga: G[A]
  y <- (fb, gb)          // Assume fb: F[B], gb: G[B]
} yield h(x, y)          // Assume h: (A, B) => C
```

The expression `result` is equivalent to the following code that works with F and G separately:

```
val result1: F[C] = for {
  x <- fa
  y <- fb
} yield h(x, y)
```

```
val result2: G[C] = for {
  x <- ga
  y <- gb
} yield h(x, y)
```

```
val result: (F[C], G[C]) =
  (result1, result2)
```

A composition of two L-effects is a pair consisting of F-effects and G-effects composed separately. Because of that, the composition of L-effects is associative (and so L is a lawful semimonad) as long as F- and G-effects are themselves composed associatively.

Statement 10.2.8.3 Given any functor F, the functor $L^A \stackrel{\text{def}}{=} A \times F^A$ is a semimonad.

Proof We begin by implementing the `flatten` method for L, with the type signature $A \times F^A \times F^{A \times F^A} \to A \times F^A$. Since we know nothing about the functor F, we cannot extract values of type A from F^A. We also do not have a `flatten` method for F. How can we get values of type A and F^A out of $A \times F^A \times F^{A \times F^A}$? One possibility is simply to discard the part of type $F^{A \times F^A}$:

```
def flatten1_L[A]: ((A, F[A]), F[(A, F[A])]) => (A, F[A]) = _._1
```

The other possibility is to transform $F^{A \times F^A}$ to F^A within the functor F:

```
def flatten2_L[A]: ((A, F[A]), F[(A, F[A])]) => (A, F[A]) = { case (afa, fafa) =>
  (afa._1, fafa.map(_._1))
}
```

In the code notation, these alternative implementations may be written as:

$$\text{ftn}_{1\,L} \stackrel{\text{def}}{=} \pi_1 = p^{:A \times F^A} \times q^{:F^{A \times F^A}} \to p \quad , \quad \text{ftn}_{2\,L} \stackrel{\text{def}}{=} p^{:A \times F^A} \times q^{:F^{A \times F^A}} \to (p \triangleright \pi_1) \times (q \triangleright \pi_1^{\uparrow F}) = \pi_1 \boxtimes \pi_1^{\uparrow F} \quad .$$

To check the associativity laws, we need to prepare the code for lifting to the functor L:

$$f^{\uparrow L} = f \boxtimes f^{\uparrow F} \quad , \quad \text{ftn}_{1\,L}^{\uparrow L} = \pi_1 \boxtimes \pi_1^{\uparrow F} \quad , \quad \text{ftn}_{2\,L}^{\uparrow L} = (\pi_1 \boxtimes \pi_1^{\uparrow F}) \boxtimes (\pi_1 \boxtimes \pi_1^{\uparrow F})^{\uparrow F} \quad .$$

Then we verify the associativity law for $\text{ftn}_{1\,L}$ using the projection law (7.3):

$$\text{left-hand side}: \quad \text{ftn}_{1\,L}^{\uparrow L} \fatsemi \text{ftn}_{1\,L} = (\pi_1 \boxtimes \pi_1^{\uparrow F}) \fatsemi \pi_1 = \pi_1 \fatsemi \pi_1 \quad ,$$

$$\text{right-hand side}: \quad \text{ftn}_{1\,L} \fatsemi \text{ftn}_{1\,L} = \pi_1 \fatsemi \pi_1 \quad .$$

The law holds. For $\text{ftn}_{2\,L}$, we also find that the two sides of the law are equal:

$$\text{ftn}_{2\,L}^{\uparrow L} \fatsemi \text{ftn}_{2\,L} = \left((\pi_1 \boxtimes \pi_1^{\uparrow F}) \boxtimes (\pi_1 \boxtimes \pi_1^{\uparrow F})^{\uparrow F}\right) \fatsemi (\pi_1 \boxtimes \pi_1^{\uparrow F})$$

$$\text{use Eq. (7.2)}: \quad = \left((\pi_1 \boxtimes \pi_1^{\uparrow F}) \fatsemi \pi_1\right) \boxtimes \left(\underline{(\pi_1 \boxtimes \pi_1^{\uparrow F})^{\uparrow F} \fatsemi \pi_1^{\uparrow F}}\right) = \left(\underline{(\pi_1 \boxtimes \pi_1^{\uparrow F}) \fatsemi \pi_1}\right) \boxtimes \left(\underline{(\pi_1 \boxtimes \pi_1^{\uparrow F}) \fatsemi \pi_1}\right)^{\uparrow F}$$

$$\text{use Eq. (7.3)}: \quad = (\pi_1 \fatsemi \pi_1) \boxtimes (\pi_1 \fatsemi \pi_1)^{\uparrow F} = (\pi_1 \fatsemi \pi_1) \boxtimes (\pi_1^{\uparrow F} \fatsemi \pi_1^{\uparrow F}) \quad ,$$

$$\text{ftn}_{2\,L} \fatsemi \text{ftn}_{2\,L} = (\pi_1 \boxtimes \pi_1^{\uparrow F}) \fatsemi (\pi_1 \boxtimes \pi_1^{\uparrow F}) = (\pi_1 \fatsemi \pi_1) \boxtimes (\pi_1^{\uparrow F} \fatsemi \pi_1^{\uparrow F}) \quad .$$

So, both implementations (ftn_{2L} and ftn_{2L}) are associative. Those implementations of `flatten` discard either the first F-effect or the second one. Since all implementations of `flatten` discard effects, identity laws cannot hold, and so the semimonads of the form $L^A \stackrel{\text{def}}{=} A \times F^A$ are not monads. A semimonad L combines F-effects similarly to a "trivial" semigroup (see Section 8.3.3) whose binary operation simply discards the left or the right argument. Such semigroups cannot be made into monoids since the identity laws will not hold. This construction illustrates the connections between monoids, semigroups, monads, and semimonads.

Co-products As a rule, the co-product of two monads ($F^A + G^A$) is not a monad. For simple examples, see Exercise 10.2.9.8 for $\mathbb{1} + F^A$ (where $F^A \stackrel{\text{def}}{=} A \times A$) and Exercise 10.2.9.5 for $M^A + M^A$ with an arbitrary monad M. An exception to that rule is a co-product with the *identity* monad:

Statement 10.2.8.4 If F is any monad, the functor $L^A \stackrel{\text{def}}{=} A + F^A$ is a monad. (The functor L is called the **free pointed functor on** F, for reasons explained in Chapter 13.)

Proof We need to define the monad methods for L, for which we may use the `pure` and `flatten` methods of F. Begin with the `flatten` method, which needs to have the type signature:

$$\text{ftn}_L : L^{L^A} \to L^A = A + F^A + F^{A+F^A} \to A + F^A \quad .$$

Since we know nothing about the specific monad F, we cannot extract a value of type A out of F^A. However, we can use F's `pure` method to create a value of type F^A out of A. This allows us to convert $A + F^A$ into F^A using the function we will denote γ:

```
type L[A] = Either[A, F[A]]
def gamma[A]: L[A] => F[A] = {
  case Left(a)   => F.pure(a)
  case Right(fa) => fa
}
```

$$\gamma^A \stackrel{\text{def}}{=} \begin{array}{|c||c|} & F^A \\ \hline A & \text{pu}_F \\ F^A & \text{id} \end{array} \quad .$$

Lifting this function to F, we can convert F^{A+F^A} into F^{F^A} and finally into F^A via F's `flatten` method:

```
def flatten_L[A]: L[L[A]] => L[A] = {
  case Left(Left(a))   => Left(a)
  case Left(Right(fa)) => Right(fa)
  case Right(g)        => Right(g.map(gamma).flatten)
}  // The last line equals ‘Right(g.flatMap(gamma))‘.
```

$$\text{ftn}_L \stackrel{\text{def}}{=} \begin{array}{|c||cc|} & A & F^A \\ \hline A & \text{id} & \mathbb{0} \\ F^A & \mathbb{0} & \text{id} \\ F^{L^A} & \mathbb{0} & \gamma^{\uparrow F} ⨾ \text{ftn}_F \end{array} \quad .$$

Is there another implementation for ftn_L? We could have replaced A by F^A using pu_F. However, that code would never return a result of type $A + \mathbb{0}$, which makes it impossible to satisfy identity laws such as $\text{pu}_F ⨾ \text{ftn}_F = \text{id}$.

The `pure` method for L could be defined in two ways: $\text{pu}_L \stackrel{\text{def}}{=} a^{:A} \to a + \mathbb{0}$ or $\text{pu}_L \stackrel{\text{def}}{=} a \to \mathbb{0} + \text{pu}_F(a)$. It turns out that only the first definition satisfies the monad L's identity laws (Exercise 10.2.9.16).

To verify the identity laws for the definition $\text{pu}_L \stackrel{\text{def}}{=} a^{:A} \to a + \mathbb{0}$, begin by writing the lifting code for the functor L and a fully split matrix for $\text{pu}_L^{\uparrow L}$:

$$(f^{:A\to B})^{\uparrow L} = \begin{array}{|c||cc|} & B & F^B \\ \hline A & f & \mathbb{0} \\ F^A & \mathbb{0} & f^{\uparrow F} \end{array} \quad , \quad \text{pu}_L^{\uparrow L} = \begin{array}{|c||cc|} & L^A & F^{L^A} \\ \hline A & \text{pu}_L & \mathbb{0} \\ F^A & \mathbb{0} & \text{pu}_L^{\uparrow F} \end{array} = \begin{array}{|c||ccc|} & A & F^A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} & \mathbb{0} \\ F^A & \mathbb{0} & \mathbb{0} & \text{pu}_L^{\uparrow F} \end{array} \quad .$$

Then write the two identity laws and simplify using matrix compositions:

$$\text{pu}_L^{L^A} ⨾ \text{ftn}_L = \begin{array}{|c||ccc|} & A & F^A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} & \mathbb{0} \\ F^A & \mathbb{0} & \text{id} & \mathbb{0} \end{array} ⨾ \begin{array}{|c||cc|} & A & F^A \\ \hline A & \text{id} & \mathbb{0} \\ F^A & \mathbb{0} & \text{id} \\ F^{L^A} & \mathbb{0} & \gamma^{\uparrow F} ⨾ \text{ftn}_F \end{array} = \begin{array}{|c||cc|} & A & F^A \\ \hline A & \text{id} & \mathbb{0} \\ F^A & \mathbb{0} & \text{id} \end{array} = \text{id} \quad ,$$

$$\mathrm{pu}_L^{\uparrow L} \fatsemi \mathrm{ftn}_L = \begin{array}{c||ccc} & A & F^A & F^{L^A} \\ \hline A & \mathrm{id} & \mathbb{0} & \mathbb{0} \\ F^A & \mathbb{0} & \mathbb{0} & \mathrm{pu}_L^{\uparrow F} \end{array} \fatsemi \begin{array}{c||cc} & A & F^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ F^A & \mathbb{0} & \mathrm{id} \\ F^{L^A} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} = \begin{array}{c||cc} & A & F^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ F^A & \mathbb{0} & \mathrm{pu}_L^{\uparrow F} \fatsemi \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} .$$

We will show that the last matrix equals identity by proving that:

$$\mathrm{id} \stackrel{?}{=} \mathrm{pu}_L^{\uparrow F} \fatsemi \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F = (\mathrm{pu}_L \fatsemi \gamma)^{\uparrow F} \fatsemi \mathrm{ftn}_F \quad .$$

Let us simplify the function $\mathrm{pu}_L \fatsemi \gamma$ in a separate calculation:

$$\mathrm{pu}_L \fatsemi \gamma = \begin{array}{c||cc} & A & F^A \\ \hline A & \mathrm{id} & \mathbb{0} \end{array} \fatsemi \begin{array}{c||c} & F^A \\ \hline A & \mathrm{pu}_F \\ F^A & \mathrm{id} \end{array} = \mathrm{pu}_F \quad .$$

It remains to show that:

$$\mathrm{id} \stackrel{?}{=} (\mathrm{pu}_L \fatsemi \gamma)^{\uparrow F} \fatsemi \mathrm{ftn}_F = \mathrm{pu}_F^{\uparrow F} \fatsemi \mathrm{ftn}_F \quad .$$

But this is the right identity law of the monad F, which (by assumption) is a lawful monad.

The next step is to verify the associativity law for L. Write the code for $\mathrm{ftn}_L^{\uparrow L}$ and $\mathrm{ftn}_L^{L^A}$:

$$\mathrm{ftn}_L^{\uparrow L} = \begin{array}{c||cc} & L^A & F^{L^A} \\ \hline L^{L^A} & \mathrm{ftn}_L & \mathbb{0} \\ F^{L^{L^A}} & \mathbb{0} & \mathrm{ftn}_L^{\uparrow F} \end{array} = \begin{array}{c||ccc} & A & F^A & F^{L^A} \\ \hline A & \mathrm{id} & \mathbb{0} & \mathbb{0} \\ F^A & \mathbb{0} & \mathrm{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F & \mathbb{0} \\ F^{L^{L^A}} & \mathbb{0} & \mathbb{0} & \mathrm{ftn}_L^{\uparrow F} \end{array} ,$$

$$\mathrm{ftn}_L^{L^A} = \begin{array}{c||cc} & L^A & F^{L^A} \\ \hline L^A & \mathrm{id} & \mathbb{0} \\ F^{L^a} & \mathbb{0} & \mathrm{id} \\ F^{L^{L^A}} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} = \begin{array}{c||ccc} & A & F^A & F^{L^A} \\ \hline A & \mathrm{id} & \mathbb{0} & \mathbb{0} \\ F^A & \mathbb{0} & \mathrm{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \mathbb{0} & \mathrm{id} \\ F^{L^{L^A}} & \mathbb{0} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} .$$

We are ready to verify the associativity law. Simplify both sides using matrix compositions:

$$\mathrm{ftn}_L^{\uparrow L} \fatsemi \mathrm{ftn}_L = \begin{array}{c||ccc} & A & F^A & F^{L^A} \\ \hline A & \mathrm{id} & \mathbb{0} & \mathbb{0} \\ F^A & \mathbb{0} & \mathrm{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F & \mathbb{0} \\ F^{L^{L^A}} & \mathbb{0} & \mathbb{0} & \mathrm{ftn}_L^{\uparrow F} \end{array} \fatsemi \begin{array}{c||cc} & A & F^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ F^A & \mathbb{0} & \mathrm{id} \\ F^{L^A} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} = \begin{array}{c||cc} & A & F^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ F^A & \mathbb{0} & \mathrm{id} \\ F^{L^A} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \\ F^{L^{L^A}} & \mathbb{0} & \mathrm{ftn}_L^{\uparrow F} \fatsemi \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} ,$$

$$\mathrm{ftn}_L^{L^A} \fatsemi \mathrm{ftn}_L = \begin{array}{c||ccc} & A & F^A & F^{L^A} \\ \hline A & \mathrm{id} & \mathbb{0} & \mathbb{0} \\ F^A & \mathbb{0} & \mathrm{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \mathbb{0} & \mathrm{id} \\ F^{L^{L^A}} & \mathbb{0} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} \fatsemi \begin{array}{c||cc} & A & F^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ F^A & \mathbb{0} & \mathrm{id} \\ F^{L^A} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} = \begin{array}{c||cc} & A & F^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ F^A & \mathbb{0} & \mathrm{id} \\ F^{L^A} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \\ F^{L^{L^A}} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \fatsemi \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} .$$

We will show that the last matrices are equal if we prove the equality of the expressions:

$$\text{ftn}_L^{\uparrow F} \fatsemi \gamma^{\uparrow F} \fatsemi \text{ftn}_F \stackrel{?}{=} \gamma^{\uparrow F} \fatsemi \text{ftn}_F \fatsemi \gamma^{\uparrow F} \fatsemi \text{ftn}_F \quad .$$

These expressions are functions of type $F^{L^A} \to F^A$. Simplify them separately:

$$\begin{aligned}
&\text{left-hand side}: && \text{ftn}_L^{\uparrow F} \fatsemi \gamma^{\uparrow F} \fatsemi \text{ftn}_F = \left(\text{ftn}_L \fatsemi \gamma\right)^{\uparrow F} \fatsemi \text{ftn}_F \quad , \\
&\text{right-hand side}: && \gamma^{\uparrow F} \fatsemi \underline{\text{ftn}_F \fatsemi \gamma^{\uparrow F}} \fatsemi \text{ftn}_F \\
&\text{naturality of } \text{ftn}_F: && = \gamma^{\uparrow F} \fatsemi \gamma^{\uparrow F \uparrow F} \fatsemi \underline{\text{ftn}_F \fatsemi \text{ftn}_F} \\
&\text{associativity law of } F: && = \underline{\gamma^{\uparrow F} \fatsemi \gamma^{\uparrow F \uparrow F} \fatsemi \text{ftn}_F^{\uparrow F}} \fatsemi \text{ftn}_F = \left(\gamma \fatsemi \gamma^{\uparrow F} \fatsemi \text{ftn}_F\right)^{\uparrow F} \fatsemi \text{ftn}_F \quad .
\end{aligned}$$

It remains to show the equality of the functions under $(...)^{\uparrow F}$:

$$\text{ftn}_L \fatsemi \gamma \stackrel{?}{=} \underline{\gamma \fatsemi \gamma^{\uparrow F}} \fatsemi \text{ftn}_F = \gamma^{\uparrow L} \fatsemi \gamma \fatsemi \text{ftn}_F \quad , \tag{10.18}$$

where in the last step we used the naturality law of $\gamma^{:L^A \to F^A}$, which is a natural transformation:

$$\gamma \fatsemi f^{\uparrow F} = f^{\uparrow L} \fatsemi \gamma \quad , \quad \text{for all } f^{:A \to B} \quad .$$

Finally, the functions in Eq. (10.18) are equal due to F's identity law ($\text{pu}_F \fatsemi \text{ftn}_F = \text{id}$):

$$\text{ftn}_L \fatsemi \gamma = \begin{array}{c||cc|} & A & F^A \\ \hline A & \text{id} & \mathbb{0} \\ F^A & \mathbb{0} & \text{id} \\ F^{L^A} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \text{ftn}_F \end{array} \fatsemi \begin{array}{c||c|} & F^A \\ \hline A & \text{pu}_F \\ F^A & \text{id} \end{array} = \begin{array}{c||c|} & F^A \\ \hline A & \text{pu}_F \\ F^A & \text{id} \\ F^{L^A} & \gamma^{\uparrow F} \fatsemi \text{ftn}_F \end{array} = \begin{array}{c||c|} & F^A \\ \hline L^A & \gamma \\ F^{L^A} & \gamma^{\uparrow F} \fatsemi \text{ftn}_F \end{array} \quad ,$$

$$\gamma^{\uparrow L} \fatsemi \gamma \fatsemi \text{ftn}_F = \begin{array}{c||cc|} & F^A & F^{F^A} \\ \hline L^A & \gamma & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \gamma^{\uparrow F} \end{array} \fatsemi \begin{array}{c||c|} & F^{F^A} \\ \hline F^A & \text{pu}_F \\ F^{F^A} & \text{id} \end{array} \fatsemi \text{ftn}_F = \begin{array}{c||c|} & F^A \\ \hline L^A & \underline{\gamma \fatsemi \text{pu}_F \fatsemi \text{ftn}_F} \\ F^{L^A} & \gamma^{\uparrow F} \fatsemi \text{id} \fatsemi \text{ftn}_F \end{array} = \begin{array}{c||c|} & F^A \\ \hline L^A & \gamma \\ F^{L^A} & \gamma^{\uparrow F} \fatsemi \text{ftn}_F \end{array} \quad .$$

This concludes the proof.

Function types If F is a monad, can we find a type constructor H such that $L^A \stackrel{\text{def}}{=} H^A \to F^A$ is a monad? In order for L to be a functor, H must be a contrafunctor. We have already seen that $Z \to A$ is a monad (a `Reader`) when Z is a fixed type. The type expression $Z \to A$ is indeed of the form $H^A \to F^A$ if the type constructors are set to $H^A \stackrel{\text{def}}{=} Z$ and $F^A \stackrel{\text{def}}{=} A$. The following statements show two generalizations of the `Reader` monad to other H and F.

Statement 10.2.8.5 For a fixed type Z, if a functor F is a (semi)monad then so is $L^A \stackrel{\text{def}}{=} Z \to F^A$.

Proof As with other monads of function type, derivations are made shorter by using the flipped Kleisli trick. We assume that F's Kleisli composition $\diamond_F$ is known and obeys the associativity law. When F is a monad, we also assume that its pure method pu_F is known and obeys the identity laws. For the monad L, we will now define the flipped Kleisli composition $\tilde{\diamond}_L$ and prove its laws.

The ordinary Kleisli functions have types $A \to Z \to F^B$, so the flipped Kleisli functions have types $Z \to A \to F^B$. We begin by defining the flipped Kleisli composition $\tilde{\diamond}_L$ and the flipped method $\tilde{\text{pu}}_L$ that has type $Z \to A \to F^A$ (note that we use the *non-flipped* methods $\diamond_F$ and pu_F of the monad F):

$$f^{:Z \to A \to F^B} \tilde{\diamond}_L g^{:Z \to B \to F^C} \stackrel{\text{def}}{=} z^{:Z} \to f(z) \diamond_F g(z) \quad , \qquad \tilde{\text{pu}}_L \stackrel{\text{def}}{=} _^{:Z} \to \text{pu}_F \quad .$$

To verify the laws, it is convenient to substitute an arbitrary $z^{:Z}$ and define $\tilde{\diamond}_L$ and $\tilde{\text{pu}}_L$ by:

$$z \triangleright (f \tilde{\diamond}_L g) \stackrel{\text{def}}{=} (z \triangleright f) \diamond_F (z \triangleright g) \quad , \qquad z \triangleright \tilde{\text{pu}}_L \stackrel{\text{def}}{=} \text{pu}_F \quad .$$

For the left-hand side of the associativity law, we write:

$$z \triangleright \left(\left(f \,\tilde{\diamond}_L\, g\right) \tilde{\diamond}_L\, h\right) = \left(z \triangleright \left(f \,\tilde{\diamond}_L\, g\right)\right) \diamond_F \left(z \triangleright h\right) = \left(z \triangleright f\right) \diamond_F \left(z \triangleright g\right) \diamond_F \left(z \triangleright h\right) \quad ,$$

where we omitted parentheses around $\diamond_F$ since its associativity is given. The right-hand side:

$$z \triangleright \left(f \,\tilde{\diamond}_L\, \left(g \,\tilde{\diamond}_L\, h\right)\right) = \left(z \triangleright f\right) \diamond_F \left(z \triangleright \left(g \,\tilde{\diamond}_L\, h\right)\right) = \left(z \triangleright f\right) \diamond_F \left(z \triangleright g\right) \diamond_F \left(z \triangleright h\right) \quad .$$

We obtained the same expression for both sides of the law.

Assuming now that F is a monad with a lawful `pure` method, we verify L's identity laws:

$$\text{left identity law of } L: \quad z \triangleright \left(\tilde{\text{pu}}_L \,\tilde{\diamond}_L\, f\right) = \left(z \triangleright \tilde{\text{pu}}_L\right) \diamond_F \left(z \triangleright f\right) = \underline{\text{pu}_F \diamond_F} \left(z \triangleright f\right) = z \triangleright f \quad ,$$

$$\text{right identity law of } L: \quad z \triangleright \left(f \,\tilde{\diamond}_L\, \tilde{\text{pu}}_L\right) = \left(z \triangleright f\right) \diamond_F \left(z \triangleright \tilde{\text{pu}}_L\right) = \left(z \triangleright f\right) \underline{\diamond_F \text{pu}_F} = z \triangleright f \quad .$$

It remains to convert the flipped Kleisli methods $\tilde{\diamond}_L$ and $\tilde{\text{pu}}_L$ to the code of L's `flatMap` and `pure`:

```
type L[A] = Z => F[A]           // The type Z and a semimonad F must be already defined.
def flatMap_L[A, B](la: L[A])(f: A => L[B]): L[B] = { z => la(z).flatMap(a => f(a)(z)) }
def pure_L[A](a: A): L[A] = { _ => implicitly[Monad[F]].pure(a) }
```

Statement 10.2.8.6 For any contrafunctor H^A, the functor $L^A \stackrel{\text{def}}{=} H^A \to A$ is a monad.

Proof We use the flipped Kleisli formulation for L. The flipped L-Kleisli functions have types $H^B \to A \to B$ instead of $A \to H^B \to B$. The flipped Kleisli composition $\tilde{\diamond}_L$ of an $f^{:H^B \to A \to B}$ and a $g^{:H^C \to B \to C}$ must have type $H^C \to A \to C$. To infer this function's code, begin with a typed hole:

$$f^{:H^B \to A \to B} \,\tilde{\diamond}_L\, g^{:H^C \to B \to C} = k^{:H^C} \to ???^{:A \to C} \quad .$$

Looking at the available data, we notice that a value of type $A \to C$ will be found if we apply f and g to some arguments and then compose the resulting functions of types $A \to B$ and $B \to C$:

$$???^{:A \to C} = f\left(???^{:H^B}\right) ⨾ g\left(???^{:H^C}\right) \quad .$$

We already have a value h of type H^C, so we can apply $g(k)$ and get a function of type $B \to C$. The remaining typed hole requires a value of type H^B; but we only have the value $k^{:H^C}$. We cannot create values of type H^B from scratch. The only way of filling that type hole is to transform k into the type H^B. This is possible by lifting a function of type $B \to C$ to the contrafunctor H. We already have such a function, namely $g(k)$. So, we write:

$$???^{:H^B} = k^{:H^C} \triangleright \left(???^{:B \to C}\right)^{\downarrow H} = k \triangleright \left(g(k)\right)^{\downarrow H} \quad .$$

Putting the entire code together and substituting an arbitrary value $k^{:H^C}$, we get:

$$k^{:H^C} \triangleright \left(f^{:H^B \to A \to B} \,\tilde{\diamond}_L\, g^{:H^C \to B \to C}\right) \stackrel{\text{def}}{=} f\left(k \triangleright \left(g(k)\right)^{\downarrow H}\right) ⨾ g(k) \quad . \tag{10.19}$$

The flipped `pure` method ($\tilde{\text{pu}}_L$) is defined by:

$$\tilde{\text{pu}}_L^{:H^A \to A \to A} \stackrel{\text{def}}{=} _^{:H^A} \to \text{id}^{:A \to A} \quad , \qquad \tilde{\text{pu}}_L(k) = \text{id} \quad . \tag{10.20}$$

These formulas yield the following Scala definitions of `pure` and `flatMap` (see Statement 14.5.3.1):

```
type L[A] = H[A] => A                  // The contrafunctor H must be already defined.
def pure_L[A](a: A): L[A] = { _ => a }
def flatMap_L[B, C](lb: L[B])(g: B => L[C]): L[C] = { (k: H[C]) =>
  val bc: (B => C) = { b => g(b)(k) } // Corresponds to g(k) : B → C.
  val hb: H[B] = k.contramap(bc)      // Fill the typed hole ??? : H^B.
  g(lb(hb))(k)           // Corresponds to the composition f(...) ⨾ g(k).
}
```

Let us now verify the monad laws of L in the flipped Kleisli form. The identity laws:

$$\text{expect } g(k): \quad k \triangleright (\tilde{\text{pu}}_L \tilde{\diamond}_L g) = \underline{\tilde{\text{pu}}_L(k \triangleright (g(k))^{\downarrow H})} ⨾ g(k) = \underline{\text{id}} ⨾ g(k) = g(k) \quad ,$$

$$\text{expect } f(k): \quad k \triangleright (f \tilde{\diamond}_L \tilde{\text{pu}}_L) = f(k \triangleright \underline{(\tilde{\text{pu}}_L(k))}^{\downarrow H}) ⨾ \underline{\tilde{\text{pu}}_L(k)} = f(k \triangleright \underline{\text{id}^{\downarrow H}}) ⨾ \underline{\text{id}} = f(k) \quad .$$

To verify the associativity law, write the two sides separately and use Eq. (10.19) repeatedly:

$$\begin{aligned}
& k^{:H^C} \triangleright \big((f \tilde{\diamond}_L g) \tilde{\diamond}_L h\big) = \big(\underline{(f \tilde{\diamond}_L g)(k \triangleright (h(k))^{\downarrow H})}\big) ⨾ h(k) = \big(\underline{k \triangleright (h(k))^{\downarrow H} \triangleright (f \tilde{\diamond}_L g)}\big) ⨾ h(k) \\
& \quad = f\big(k \triangleright (h(k))^{\downarrow H} \triangleright (g(k \triangleright (h(k))^{\downarrow H}))^{\downarrow H}\big) ⨾ g(k \triangleright (h(k))^{\downarrow H}) ⨾ h(k) \quad , \\
& k \triangleright \big(f \tilde{\diamond}_L (g \tilde{\diamond}_L h)\big) = f\big(k \triangleright \underline{((g \tilde{\diamond}_L h)(k))}^{\downarrow H}\big) ⨾ \underline{(g \tilde{\diamond}_L h)(k)} = f\big(k \triangleright \underline{(k \triangleright (g \tilde{\diamond}_L h))}^{\downarrow H}\big) ⨾ \underline{k \triangleright (g \tilde{\diamond}_L h)} \\
& \quad = f\big(k \triangleright (\underline{g(k \triangleright (h(k))^{\downarrow H}) ⨾ h(k)})^{\downarrow H}\big) ⨾ g(k \triangleright (h(k))^{\downarrow H}) ⨾ h(k) \\
\text{lift to } H: \quad & \quad = f\big(k \triangleright (h(k))^{\downarrow H} \triangleright (g(k \triangleright (h(k))^{\downarrow H}))^{\downarrow H}\big) ⨾ g(k \triangleright (h(k))^{\downarrow H}) ⨾ h(k) \quad .
\end{aligned}$$

Now both sides are rewritten into identical expressions. This confirms the associativity law for L. □

Statement 10.2.8.6 applies to type constructors of the form $H^A \to A$ but not to $H^A \to G^A$ with an arbitrary monad G. It turns out that $H^A \to G^A$ is a monad when the contrafunctor H is suitably adapted to the monad G. The necessary properties of H are formulated by the notion of G-filterable contrafunctor, which we briefly introduced in Section 9.4.3. We repeat the definition here:

Definition 10.2.8.7 For a given monad G, a contrafunctor H is **G-filterable** if it has a `lift` method,

$$\text{lift}_{G,H} : (A \to G^B) \to H^B \to H^A \quad ,$$

satisfying the identity and (contravariant) composition laws

$$\text{lift}_{G,H}(\text{pu}_G) = \text{id}^{:H^A \to H^A} \quad , \qquad \text{lift}_{G,H}(f \diamond_G g) = \text{lift}_{G,H}(g) ⨾ \text{lift}_{G,H}(f) \quad .$$

Let us look at some simple examples of G-filterable contrafunctors.

Statement 10.2.8.8 For any given (semi)monad G and any fixed type Z, the following contrafunctors are G-filterable: **(a)** $H^A \stackrel{\text{def}}{=} Z$, **(b)** $H^A \stackrel{\text{def}}{=} G^A \to Z$, **(c)** $H^A \stackrel{\text{def}}{=} A \to G^Z$.

Proof In each case, we need to define the function $\text{lift}_{G,H}$ and verify its laws. If G is only a semimonad, we will derive the composition law of $\text{lift}_{G,H}$ assuming only the associativity law of G (but not assuming any identity laws). For a full monad G, we will derive the identity law of $\text{lift}_{G,H}$ by using the identity laws of G.

(a) Since $H^A \stackrel{\text{def}}{=} Z$ is a constant contrafunctor, we define $\text{lift}_{G,H}(f) \stackrel{\text{def}}{=} \text{id}^{:Z \to Z}$. The identity function will always satisfy the identity and composition laws.

(b) Define the function $\text{lift}_{G,H}$ by:

```
type H[A] = G[A] => Z        // The type Z and the monad G must be already defined.
def lift_G[A, B](k: H[B])(f: A => G[B]): H[A] = { (ga: G[A]) => k(ga.flatMap(f)) }
```

$$\text{lift}_{G,H}(f^{:A \to G^B}) \stackrel{\text{def}}{=} k^{:G^B \to Z} \to ???^{:G^A \to Z} = k^{:G^B \to Z} \to \text{flm}_G(f) ⨾ k \quad , \qquad k \triangleright \text{lift}_{G,H}(f) \stackrel{\text{def}}{=} \text{flm}_G(f) ⨾ k \quad .$$

Verify the laws of $\text{lift}_{G,H}$ using the right identity and the associativity law of G's `flatMap`:

$$\begin{aligned}
\text{identity law; expect to equal } k: \quad & k \triangleright \text{lift}_{G,H}(\text{pu}_G) = \underline{\text{flm}_G(\text{pu}_G)} ⨾ k = \text{id} ⨾ k = k \quad , \\
\text{left-hand side of composition law}: \quad & k \triangleright \text{lift}_{G,H}(f \diamond_G g) = \text{flm}_G(f \diamond_G g) ⨾ k = \underline{\text{flm}_G(f ⨾ \text{flm}_G(g))} ⨾ k \\
\text{associativity law of } \text{flm}_G: \quad & \quad = \text{flm}_G(f) ⨾ \text{flm}_G(g) ⨾ k \quad , \\
\text{right-hand side of composition law}: \quad & k \triangleright \text{lift}_{G,H}(g) ⨾ \text{lift}_{G,H}(f) = k \triangleright \text{lift}_{G,H}(g) \triangleright \underline{\text{lift}_{G,H}(f)} \\
& \quad = \text{flm}_G(f) ⨾ \big(\underline{k \triangleright \text{lift}_{G,H}(g)}\big) = \text{flm}_G(f) ⨾ \text{flm}_G(g) ⨾ k \quad .
\end{aligned}$$

Both sides of the composition law of $\text{lift}_{G,H}$ are now equal.

(c) Define the function $\text{lift}_{G,H}$ by:

```
type H[A] = A => G[Z]  // The type Z and the monad G must be already defined.
def lift_G[A, B](k: H[B])(f: A => G[B]): H[A] = { (a: A) => f(a).flatMap(k) }
```

$$\text{lift}_{G,H}(f^{:A\to G^B}) \stackrel{\text{def}}{=} k^{:B\to G^Z} \to ???^{:A\to G^Z} = k^{:G^B\to Z} \to f \diamond_G k \quad , \qquad k \triangleright \text{lift}_{G,H}(f) \stackrel{\text{def}}{=} f \diamond_G k \quad .$$

The laws of $\text{lift}_{G,H}$ are satisfied because the operation $\diamond_G$ obeys its identity and associativity laws:

$$\begin{aligned}
&\text{identity law; expect to equal } k: && k \triangleright \text{lift}_{G,H}(\text{pu}_G) = \text{pu}_G \diamond_G k = k \quad ,\\
&\text{left-hand side of composition law}: && k \triangleright \text{lift}_{G,H}(f \diamond_G g) = (f \diamond_G g) \diamond_G k \quad ,\\
&\text{right-hand side of composition law}: && k \triangleright \text{lift}_{G,H}(g) ⨾ \text{lift}_{G,H}(f) = k \triangleright \text{lift}_{G,H}(g) \triangleright \underline{\text{lift}_{G,H}(f)}\\
& && \quad = f \diamond_G \left(\underline{k \triangleright \text{lift}_{G,H}(g)}\right) = f \diamond_G (g \diamond_G k) \quad .
\end{aligned}$$

Both sides of the composition law of $\text{lift}_{G,H}$ are equal due to the associativity law of $\diamond_G$. □

We will now use filterable contrafunctors to generalize Statements 10.2.8.5 and 10.2.8.6.

Statement 10.2.8.9 For any (semi)monad G and any G-filterable contrafunctor H, the functor L defined by $L^A \stackrel{\text{def}}{=} H^A \to G^A$ is a (semi)monad.

Proof We use the flipped Kleisli formulation for L and the ordinary Kleisli formulation for G. The flipped L-Kleisli functions have types $H^B \to A \to G^B$ instead of $A \to H^B \to G^B$. The flipped Kleisli composition $\tilde{\diamond}_L$ of an $f^{:H^B\to A\to G^B}$ and a $g^{:H^C\to B\to G^C}$ must have type $H^C \to A \to G^C$. Similarly to what we did in the proof of Statement 10.2.8.6, we will define $\tilde{\diamond}_L$ by applying f and g to typed holes, getting values of types $A \to G^B$ and $B \to G^C$, and using the given operation $\diamond_G$:

$$f^{:H^B\to A\to G^B} \tilde{\diamond}_L g^{:H^C\to B\to G^C} = k^{:H^C} \to ???^{:A\to G^C} = k^{:H^C} \to f(???^{:H^B}) \diamond_G g(???^{:H^C}) \quad .$$

The typed hole $???^{:H^C}$ is filled by k, while $???^{:H^B}$ is computed from k using $\text{lift}_{G,H}$:

$$???^{:H^B} = k \triangleright \text{lift}_{G,H}(???^{:B\to G^C}) = k \triangleright \text{lift}_{G,H}(g(k)) \quad .$$

Putting the code together, we obtain

$$\begin{aligned}
&k^{:H^C} \triangleright \left(f^{:H^B\to A\to G^B} \tilde{\diamond}_L g^{:H^C\to B\to G^C}\right) \stackrel{\text{def}}{=} f\left(k \triangleright \text{lift}_{G,H}(g(k))\right) \diamond_G g(k) \quad ,\\
&k^{:H^C} \triangleright \tilde{\text{pu}}_L \stackrel{\text{def}}{=} \text{pu}_G \quad .
\end{aligned}$$

These definitions reduce to Eqs. (10.19)–(10.20) if we set G to the identity monad, $G^A = \text{Id}^A \stackrel{\text{def}}{=} A$. Any contrafunctor H is Id-filterable because we can define the required method $\text{lift}_{\text{Id},H}$ by:

$$\text{lift}_{\text{Id},H}(f^{:B\to C}) \stackrel{\text{def}}{=} f^{\downarrow H} \quad .$$

The laws of $\text{lift}_{\text{Id},H}$ follow from the identity and composition laws of the contrafunctor H.

Translating the monad L's Kleisli composition into Scala code for `flatMap`, we obtain:

```
type L[A] = H[A] => G[A] // A contrafunctor H and a monad G must be already defined.
def flatMap_L[A, B](la: L[A])(g: A => L[B]): L[B] = { (k: H[B]) =>
  val agb: (A => G[B]) = { a => g(a)(k) }    // Corresponds to flipped g(k).
  val ha: H[A] = lift_G(k)(agb)    // Corresponds to k ▷ lift_{G,H}(g(k)).
  la(ha).flatMap(agb)    // Corresponds to the Kleisli composition f(...) ◇_G g(k).
}
```

We now verify the laws via derivations similar to those in the proof of Statement 10.2.8.6.

We will need to use the laws of $\text{lift}_{G,H}$ (we will write simply "lift" for brevity). The identity laws of L follow from the identity laws of $\text{lift}_{G,H}$ and G:

$$\begin{aligned}
&\text{expect } g(k): && k \triangleright \left(\tilde{\text{pu}}_L \tilde{\diamond}_L g\right) = \underline{\tilde{\text{pu}}_L(k \triangleright \text{lift}\,(g(k)))} \diamond_G g(k) = \text{pu}_G \diamond_G g(k) = g(k) \quad ,\\
&\text{expect } f(k): && k \triangleright \left(f \tilde{\diamond}_L \tilde{\text{pu}}_L\right) = f\left(k \triangleright \underline{\text{lift}\,(\tilde{\text{pu}}_L(k))}\right) ⨾ \underline{\tilde{\text{pu}}_L(k)} = f(k \triangleright \underline{\text{id}}) \diamond_G \text{pu}_G = f(k) \quad .
\end{aligned}$$

The associativity law of L follows from the associativity of $\diamond_G$ and the composition law of $\text{lift}_{G,H}$:

$$\begin{aligned}
&k^{:H^C} \triangleright \left((f\,\tilde{\diamond}_L\,g)\,\tilde{\diamond}_L\,h\right) = \left(\underline{(f\tilde{\diamond}_L g)(k \triangleright \text{lift}\,(h(k)))}\right) \diamond_G h(k) = \left(\underline{k \triangleright \text{lift}\,(h(k)) \triangleright (f\,\tilde{\diamond}_L\,g)}\right) \diamond_G h(k) \\
&\quad = f\left(k \triangleright \text{lift}\,(h(k)) \triangleright \text{lift}\,(g(k \triangleright \text{lift}\,(h(k))))\right) \diamond_G g\left(k \triangleright \text{lift}\,(h(k))\right) \diamond_G h(k) \quad , \\
&k \triangleright \left(f\,\tilde{\diamond}_L\,(g\,\tilde{\diamond}_L\,h)\right) = f\left(k \triangleright \text{lift}\,(\underline{(g\,\tilde{\diamond}_L\,h)(k)})\right) \diamond_G \underline{(g\,\tilde{\diamond}_L\,h)(k)} = f\left(k \triangleright \text{lift}\,(\underline{k \triangleright (g\,\tilde{\diamond}_L\,h)})\right) \diamond_G \underline{k \triangleright (g\,\tilde{\diamond}_L\,h)} \\
&\quad = f\left(k \triangleright \text{lift}\,(\underline{g(k \triangleright \text{lift}\,(h(k))) \diamond_G h(k)})\right) \diamond_G g(k \triangleright \text{lift}\,(h(k))) \diamond_G h(k) \\
&\quad = f\left(k \triangleright \text{lift}\,(h(k)) \,;\, \text{lift}\,(g(k \triangleright \text{lift}\,(h(k))))\right) \diamond_G g\left(k \triangleright \text{lift}\,(h(k))\right) \diamond_G h(k) \quad .
\end{aligned}$$

Both sides of the associativity law are now converted into identical expressions. □

Some examples of monads obtained with these constructions for $G^A \stackrel{\text{def}}{=} \mathbb{1} + A$ are:

$$\text{Sel}^{Z,A} \stackrel{\text{def}}{=} (A \to Z) \to A \quad , \qquad \text{Search}^{Z,A} \stackrel{\text{def}}{=} (A \to \mathbb{1} + Z) \to \mathbb{1} + A \quad , \qquad L^A \stackrel{\text{def}}{=} (\mathbb{1} + A \to Z) \to \mathbb{1} + A \quad .$$

The first two are known as the selector monad and the search monad.[9] Statement 10.2.8.9 shows how to implement the monad methods for these monads and guarantees that the laws hold.

Recursive types The tree-like monads can be generalized to arbitrary shape of leaves and branches. This gives two constructions that create a monad and a semimonad from an arbitrary functor.

Statement 10.2.8.10 For any given functor F, the recursively defined functor $L^A \stackrel{\text{def}}{=} A + F^{L^A}$ is a (tree-like) monad. This monad is called the **free monad on** F, for reasons explained in Chapter 13.

Proof Begin by implementing the monad methods for L as usual for a tree-like monad:

```
type L[A] = Either[A, F[L[A]]] // The functor F must be already defined.
def pure_L[A](a: A): L[A] = Left(a)
def flatMap_L[A, B](la: L[A])(f: A => L[B]): L[B] = la match {
  case Left(a)     => f(a)
  case Right(fla) => Right(fla.map { (x: L[A]) => flatMap_L(x)(f) })  // Recursive call of flatMap_L.
}
def flatten_L[A]: L[L[A]] => L[A] = {   // Match on L[L[A]] = Either[Either[A, F[L[A]]], F[L[L[A]]]].
  case Left(a)                => Left(a)
  case Left(Right(fla))       => Right(fla)
  case Right(Right(flla))     => Right(flla.map(x => flatten_L(x)))     // Recursive call of flatten_L.
}
```

$$\text{pu}_L \stackrel{\text{def}}{=} \left|\begin{array}{c||cc} & A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} \end{array}\right| \quad , \qquad \text{ftn}_L^A : L^{L^A} \to L^A \stackrel{\text{def}}{=} \left|\begin{array}{c||cc} & A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \end{array}\right| \quad .$$

To verify the monad laws for L, we need to lift pu_L and ftn_L to the functor L. The lifting code is

$$(f^{:A\to B})^{\uparrow L} = \left|\begin{array}{c||cc} & B & F^{L^B} \\ \hline A & f & \mathbb{0} \\ F^{L^A} & \mathbb{0} & (f^{\overline{\uparrow L}})^{\uparrow F} \end{array}\right| \quad .$$

In order to compute function compositions such as $\text{pu}_L^{\uparrow L} \,;\, \text{ftn}_L$ and $\text{ftn}_L^{\uparrow L} \,;\, \text{ftn}$ in the matrix notation, we need to expand the matrices of pu_L and ftn_L, so that their output types are fully split:

$$\text{pu}_L^{\uparrow L} = \left|\begin{array}{c||cc} & L^A & F^{L^{L^A}} \\ \hline A & \text{pu}_L & \mathbb{0} \\ F^{L^A} & \mathbb{0} & (\text{pu}_L^{\overline{\uparrow L}})^{\uparrow F} \end{array}\right| = \left|\begin{array}{c||ccc} & A & F^{L^A} & F^{L^{L^A}} \\ \hline A & \text{id} & \mathbb{0} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \mathbb{0} & (\text{pu}_L^{\overline{\uparrow L}})^{\uparrow F} \end{array}\right| \quad ,$$

[9] See http://math.andrej.com/2008/11/21/a-haskell-monad-for-infinite-search-in-finite-time/

$$\text{ftn}_L^{\uparrow L} = \begin{array}{c||cc} & L^A & F^{L^{L^A}} \\ \hline L^{L^A} & \text{ftn}_L & \mathbb{0} \\ F^{L^{L^{L^A}}} & \mathbb{0} & (\overline{\text{ftn}_L^{\uparrow L}})^{\uparrow F} \end{array} = \begin{array}{c||ccc} & A & F^{L^A} & F^{L^{L^A}} \\ \hline A & \text{id} & \mathbb{0} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} & \mathbb{0} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} & \mathbb{0} \\ F^{L^{L^{L^A}}} & \mathbb{0} & \mathbb{0} & (\overline{\text{ftn}_L^{\uparrow L}})^{\uparrow F} \end{array} .$$

We can now verify the identity and associativity laws of pu_L and ftn_L. Identity laws:

$$\text{pu}_L^{L^A} ⨾ \text{ftn}_L = \begin{array}{c||ccc} & A & F^{L^A} & F^{L^{L^A}} \\ \hline A & \text{id} & \mathbb{0} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} & \mathbb{0} \end{array} ⨾ \begin{array}{c||cc} & A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \end{array} = \begin{array}{c||cc} & A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} \end{array} = \text{id} \quad ,$$

$$\text{pu}_L^{\uparrow L} ⨾ \text{ftn}_L = \begin{array}{c||ccc} & A & F^{L^A} & F^{L^{L^A}} \\ \hline A & \text{id} & \mathbb{0} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \mathbb{0} & (\overline{\text{pu}_L^{\uparrow L}})^{\uparrow F} \end{array} ⨾ \begin{array}{c||cc} & A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \end{array}$$

$$= \begin{array}{c||cc} & A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \underline{(\overline{\text{pu}_L^{\uparrow L}})^{\uparrow F} ⨾ \overline{\text{ftn}_L}^{\uparrow F}} \end{array} = \begin{array}{c||cc} & A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} \end{array} = \text{id} \quad ,$$

where in the last line we used the inductive assumption $(\overline{\text{pu}_L^{\uparrow L}})^{\uparrow F} ⨾ \overline{\text{ftn}_L}^{\uparrow F} = \text{id}$.

To verify the associativity law, simplify its both sides separately and use fully split matrices:

$$\text{ftn}_L^{L^A} ⨾ \text{ftn}_L = \begin{array}{c||ccc} & A & F^{L^A} & F^{L^{L^A}} \\ \hline A & \text{id} & \mathbb{0} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} & \mathbb{0} \\ F^{L^{L^A}} & \mathbb{0} & \mathbb{0} & \text{id} \\ F^{L^{L^{L^A}}} & \mathbb{0} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \end{array} ⨾ \begin{array}{c||cc} & A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \end{array} = \begin{array}{c||cc} & A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \\ F^{L^{L^{L^A}}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} ⨾ \overline{\text{ftn}_L}^{\uparrow F} \end{array} \quad ,$$

$$\text{ftn}_L^{\uparrow L} ⨾ \text{ftn}_L = \begin{array}{c||ccc} & A & F^{L^A} & F^{L^{L^A}} \\ \hline A & \text{id} & \mathbb{0} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} & \mathbb{0} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} & \mathbb{0} \\ F^{L^{L^{L^A}}} & \mathbb{0} & \mathbb{0} & (\overline{\text{ftn}_L^{\uparrow L}})^{\uparrow F} \end{array} ⨾ \begin{array}{c||cc} & A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \end{array} = \begin{array}{c||cc} & A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \\ F^{L^{L^{L^A}}} & \mathbb{0} & (\overline{\text{ftn}_L^{\uparrow L}})^{\uparrow F} ⨾ \overline{\text{ftn}_L}^{\uparrow F} \end{array} .$$

The two resulting matrices are equal due to the inductive assumption: $\overline{\text{ftn}_L} ⨾ \overline{\text{ftn}_L} = \overline{\text{ftn}_L^{\uparrow L}} ⨾ \overline{\text{ftn}_L}$.

Statement 10.2.8.11 For any functor F, the recursive functor $L^A \stackrel{\text{def}}{=} F^A + F^{L^A}$ (representing a tree with F-shaped leaves and branches) is a semimonad but not a monad.

Proof We have already implemented the `flatMap` method for an equivalent functor in Section 10.1.4. The corresponding Scala code for the functor `L` and for its `flatten` method is:

```
type L[A] = Either[F[A], F[L[A]]]      // The functor F must be already defined.
```

```
def flatten_L[A]: L[L[A]] => L[A] = { // Match on L[L[A]] = Either[F[L[A]], F[L[L[A]]]].
  case Left(fla)      => Right(fla)
  case Right(flla)    => Right(flla.map(x => flatten_L(x)))
}
```

Write the matrix notation for the code of `flatten` and for the lifting of `flatten` to L:

$$\text{ftn}_L^A = \begin{array}{c||cc} & F^A & F^{L^A} \\ \hline F^{L^A} & \mathbb{0} & \text{id} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \end{array} \quad , \qquad \text{ftn}_L^{L^A} = \begin{array}{c||cc} & F^{L^A} & F^{L^{L^A}} \\ \hline F^{L^{L^A}} & \mathbb{0} & \text{id} \\ F^{L^{L^{L^A}}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \end{array} \quad ,$$

$$(f^{:A\to B})^{\uparrow L} = \begin{array}{c||cc} & F^B & F^{L^B} \\ \hline F^A & f^{\uparrow F} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & (f^{\overline{\uparrow L}})^{\uparrow F} \end{array} \quad , \qquad \text{ftn}_L^{\uparrow L} = \begin{array}{c||cc} & F^{L^A} & F^{L^{L^A}} \\ \hline F^{L^{L^A}} & \overline{\text{ftn}_L}^{\uparrow F} & \mathbb{0} \\ F^{L^{L^{L^A}}} & \mathbb{0} & (\text{ftn}_L^{\overline{\uparrow L}})^{\uparrow F} \end{array} \quad .$$

To verify the associativity law, write both its parts separately:

$$\text{ftn}_L^{L^A} \fatsemi \text{ftn}_L = \begin{array}{c||cc} & F^{L^A} & F^{L^{L^A}} \\ \hline F^{L^{L^A}} & \mathbb{0} & \text{id} \\ F^{L^{L^{L^A}}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \end{array} \fatsemi \begin{array}{c||cc} & F^A & F^{L^A} \\ \hline F^{L^A} & \mathbb{0} & \text{id} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \end{array} = \begin{array}{c||cc} & F^A & F^{L^A} \\ \hline F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}}^{\uparrow F} \\ F^{L^{L^{L^A}}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \fatsemi \overline{\text{ftn}_L}^{\uparrow F} \end{array} \quad ,$$

$$\text{ftn}_L^{\uparrow L} \fatsemi \text{ftn}_L = \begin{array}{c||cc} & F^{L^A} & F^{L^{L^A}} \\ \hline F^{L^{L^A}} & \overline{\text{ftn}_L}^{\uparrow F} & \mathbb{0} \\ F^{L^{L^{L^A}}} & \mathbb{0} & (\text{ftn}_L^{\overline{\uparrow L}})^{\uparrow F} \end{array} \fatsemi \begin{array}{c||cc} & F^A & F^{L^A} \\ \hline F^{L^A} & \mathbb{0} & \text{id} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \end{array} = \begin{array}{c||cc} & F^A & F^{L^A} \\ \hline F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \\ F^{L^{L^{L^A}}} & \mathbb{0} & (\text{ftn}_L^{\overline{\uparrow L}})^{\uparrow F} \fatsemi \overline{\text{ftn}_L}^{\uparrow F} \end{array} \quad .$$

The two resulting matrices are equal due to the inductive assumption: $\overline{\text{ftn}_L}^{\uparrow F} \fatsemi \overline{\text{ftn}_L}^{\uparrow F} = (\text{ftn}_L^{\overline{\uparrow L}})^{\uparrow F} \fatsemi \overline{\text{ftn}_L}^{\uparrow F}$.

Could L be a full monad? Suppose we were given some code for the `pure` method (pu_L). If we substitute an arbitrary value $k^{:L^A}$ into the left identity law, we expect to obtain again k:

$$k^{:L^A} \triangleright \text{pu}_L \fatsemi \text{ftn}_L = k \triangleright \text{pu}_L \triangleright \begin{array}{c||cc} & F^A & F^{L^A} \\ \hline F^{L^A} & \mathbb{0} & \text{id} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L}^{\uparrow F} \end{array} \stackrel{?}{=} k \quad .$$

We notice that the matrix for ftn_L has no elements in the F^A column. This suggests information loss: the value $k \triangleright \text{pu}_L \fatsemi \text{ftn}_L$ will be always of the form $\mathbb{0} + q^{:F^{L^A}}$ with some q, regardless of k and of the implementation of pu_L. So, if we set $k \stackrel{\text{def}}{=} p^{:F^A} + \mathbb{0}$ with some p, the law cannot hold. The same problem occurs with the right identity law. We conclude that L cannot be a full monad.

10.2.9 Exercises: laws and structure of monads

Exercise 10.2.9.1 For fixed types U and V, define `flatten` for the semimonad $F^A \stackrel{\text{def}}{=} A \times U \times V$ as:

$$\text{ftn}_F \stackrel{\text{def}}{=} (a^{:A} \times u_1^{:U} \times v_1^{:V}) \times u_2^{:U} \times v_2^{:V} \to a \times u_1 \times v_2 \quad .$$

A `pure` method could be defined for F if some fixed values $u_0^{:U}$ and $v_0^{:V}$ were given:

$$\text{pu}_F \stackrel{\text{def}}{=} a^{:A} \to a \times u_0 \times v_0 \quad .$$

Show that F would still not be a full monad because identity laws fail to hold.

Exercise 10.2.9.2 Suppose a monad M is defined using the `pure` and `flatten` methods that satisfy all laws, in particular, the left identity law $\text{pu}_M \fatsemi \text{ftn}_M = \text{id}$. The `flatMap` function is then defined through `flatten` by $\text{flm}_M(f) \stackrel{\text{def}}{=} f^{\uparrow M} \fatsemi \text{ftn}_M$. Show that this `flatMap` function will satisfy its left identity law: $\text{pu}_M \fatsemi \text{flm}_M(f) = f$ for any $f^{:A\to M^B}$.

Exercise 10.2.9.3 Verify the associativity law for the `Reader` monad by an explicit derivation.

Exercise 10.2.9.4 Show that the laws hold for a non-standard `List` monad whose `flatten` method is:

```
def flatten[A](p: List[List[A]]): List[A] = if (p.exists(_.isEmpty)) Nil else p.flatten
```

Exercise 10.2.9.5 Given an arbitrary monad M, show that the functor $F^A \stackrel{\text{def}}{=} \mathbb{2} \times M^A$ (which is equivalent to $M^A + M^A$) is a semimonad but not a full monad.

Exercise 10.2.9.6 Show that $P^A \stackrel{\text{def}}{=} Z + W \times A$ is a (full) monad if W is a monoid.

Exercise 10.2.9.7 If W and R are arbitrary fixed types, is any of the functors $F^A \stackrel{\text{def}}{=} W \times (R \to A)$ and $G^A \stackrel{\text{def}}{=} R \to (W \times A)$ a lawful semimonad? Hint: try to define `flatten` for these functors.

Exercise 10.2.9.8 Show that $F^A \stackrel{\text{def}}{=} (P \to A) + (Q \to A)$ is not a semimonad (cannot define `flatMap`) when P and Q are fixed types that are different and not related by subtyping.

Exercise 10.2.9.9* Consider the functor $D^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$ (in Scala, `type D[A] = Option[(A, A)]`). Implement the `flatten` and `pure` methods for D in at least two different ways. Show that some of the monad laws fail to hold for those implementations.[10]

Exercise 10.2.9.10 For the type constructor $F^A \stackrel{\text{def}}{=} Z \times (A \to Z)$, a programmer defined `flatMap` by:

```
type F[A] = (Z, A => Z) // The fixed type Z must be already defined.
def flatMap_F[A, B](fa: F[A])(f: A => F[B]): F[B] = fa match { case (z, g) => (z, _ => z) }
```

$$\text{flm}_F(f^{:A\to Z\times(B\to Z)}) \stackrel{\text{def}}{=} z^{:Z} \times g^{:A\to Z} \to z \times (_^{:B} \to z) \quad .$$

Show that this implementation fails to satisfy the semimonad's law. (Here, Z is a fixed type.)

Exercise 10.2.9.11* Prove (by induction on p) that `flatten` and `concat` satisfy the distributive law:

$$(p +\!\!+ q) \triangleright \text{ftn} = (p \triangleright \text{ftn}) +\!\!+ (q \triangleright \text{ftn}) \quad , \qquad \forall p : \text{List}^{\text{List}^A}, q : \text{List}^{\text{List}^A} \quad .$$

Exercise 10.2.9.12 Given a monad M, consider a function `toUnit` (denoted tu_M) with type signature:

```
def toUnit[M[_]: Monad, A]: M[A] => M[Unit] = _.map(_ => ())
```

$$\text{tu}_M : M^A \to M^{\mathbb{1}} \stackrel{\text{def}}{=} (_^{:A} \to 1)^{\uparrow M} \quad .$$

Show that this function satisfies the equation:

$$\text{pu}_M^{:A\to M^A} \fatsemi \text{tu}_M^{:M^A\to M^1} = _^{:A} \to \text{pu}_M(1) \quad .$$

Exercise 10.2.9.13 Show that `M[W]` is a monoid if `M[_]` is a monad and `W` is a monoid.

Exercise 10.2.9.14 A monad is called **commutative** if swapping the order of its effects gives the same results. In terms of functor block code, this means that `result1` and `result2` are equal:

```
result1 = for { // Assume p: M[A],  q: M[B],
    x <- p    // and q does not depend on x.
    y <- q
} yield f(x, y)   // For any f: (A, B) => C.
```

```
result2 = for { // Assume p: M[A],  q: M[B],
    y <- q    // and p does not depend on y.
    x <- p
} yield f(x, y)   // For any f: (A, B) => C.
```

(a) Find out which of the monads `Option`, `List`, `Reader`, `Writer`, and `State` are commutative monads.

(b) Consider the type $M^{\mathbb{1}}$ (in Scala, this is `M[Unit]`). Since the `Unit` type is a (trivial) monoid, Exercise 10.2.9.13 shows that $M^{\mathbb{1}}$ is also a monoid. Prove that if the monad M is commutative then the binary operation $\oplus_M$ of the monoid $M^{\mathbb{1}}$ is also commutative.

[10] One can prove that $D^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$ cannot be a lawful monad; see https://stackoverflow.com/questions/49742377/

Exercise 10.2.9.15 Use monad constructions (so that law checking is unnecessary) to implement `flatten` and `pure` for $F^A \stackrel{\text{def}}{=} A + (R \to A)$, where R is a fixed type.

Exercise 10.2.9.16 Given a monad G, consider an alternative way of defining the `pure` method for the functor $L^A \stackrel{\text{def}}{=} A + G^A$ (see Statement 10.2.8.4). Show that the identity laws will *not* hold for L if `pure` is defined as `a => Right(Monad[G].pure(a))` (in the code notation, $\text{pu}_L \stackrel{\text{def}}{=} a \to \mathbb{0} + a \triangleright \text{pu}_G$).

Exercise 10.2.9.17 Implement the functor and monad methods for $F^A \stackrel{\text{def}}{=} (Z \to \mathbb{1} + A) \times \text{List}^A$ using the known constructions (so that law checking is unnecessary).

Exercise 10.2.9.18 Use monad constructions to show that the functors F defined below are monads:

(a) $F^A \stackrel{\text{def}}{=} A + A + ... + A$ and $F^A \stackrel{\text{def}}{=} A \times A \times ... \times A$ (with a fixed number of A's).

(b) A polynomial functor $F^A \stackrel{\text{def}}{=} p(A)$ when $p(x)$ is a polynomial of the form $p(x) = x^{n_1} + x^{n_2} + ... + x^{n_k}$ with some positive integers $n_1, ..., n_k$. For example, $F^A \stackrel{\text{def}}{=} A \times A + A \times A \times A + A \times A \times A \times A \times A$.

(c) A polynomial functor $F^A \stackrel{\text{def}}{=} p(A)$ when $p(x)$ is a polynomial of the form $p(x) = c_1 x^{n_1} + c_2 x^{n_2} + ... + c_{k-1} x^{n_{k-1}} + x^{n_k}$ with some positive integers $n_1, ..., n_k$, $c_1, ..., c_{k-1}$, where the coefficient at the highest power (x^{n_k}) *must* be 1. For example, $F^A \stackrel{\text{def}}{=} \mathbb{5} \times A + \mathbb{2} \times A \times A \times A + A \times A \times A \times A$.

Exercise 10.2.9.19* For a monad M, show that $L^A \stackrel{\text{def}}{=} M^{M^A}$ is a semimonad but not always a monad.

10.3 Further developments

10.3.1 Why monads must be covariant functors

We defined monads as functors with extra properties. It turns out that monads cannot be contrafunctors, and that the functor laws of a monad can be derived from the laws of `flatMap`.

A monad's code can be specified in three equivalent ways: via `flatMap` and `pure`, via `flatten` and `pure`, or via the Kleisli composition and `pure`. These methods have different laws and different uses. The `flatMap` method is used most frequently, while the Kleisli composition is rarely used. Since:

$$\text{flm}_M(f) = f^{\uparrow M} ⨾ \text{ftn}_M \quad ,$$

we could imagine that `flatMap` already has the capability of lifting a function ($f^{\uparrow M}$). Let us extract that capability from the code of `flatMap`: compose $f^{\uparrow M}$ with the right identity law:

$$\text{pu}_M^{\uparrow M} ⨾ \text{ftn}_M = \text{id} \quad , \quad \text{so} \quad f^{\uparrow M} ⨾ \underline{\text{pu}_M^{\uparrow M} ⨾ \text{ftn}_M} = f^{\uparrow M} \quad .$$

The left-hand side equals $\text{flm}_M(f ⨾ \text{pu}_M)$. We found a formula that expresses `map` through `flatMap`:

```
p.map(f) == p.flatMap(f andThen M.pure)
```

$$f^{\uparrow M} = \text{flm}_M(f ⨾ \text{pu}_M) \quad . \tag{10.21}$$

The laws of `map` will then follow if we assume that `flatMap` is lawful. The identity law of `map`:

$$\begin{aligned} \text{id}^{\uparrow M} &= \text{flm}_M(\text{id} ⨾ \text{pu}_M) = \underline{\text{flm}_M(\text{pu}_M)} \\ \text{right identity law of } \text{flm}_M : \quad &= \text{id} \quad . \end{aligned}$$

The composition law of `map`:

$$\begin{aligned} \text{expect to equal } (f ⨾ g)^{\uparrow M} : \quad & f^{\uparrow M} ⨾ g^{\uparrow M} = \text{flm}_M(f ⨾ \text{pu}_M) ⨾ \text{flm}_M(g ⨾ \text{pu}_M) \\ \text{associativity law of } \text{flm}_M : \quad & = \text{flm}_M(f ⨾ \underline{\text{pu}_M ⨾ \text{flm}_M}(g ⨾ \text{pu}_M)) \\ \text{left identity law of } \text{flm}_M : \quad & = \text{flm}_M(f ⨾ g ⨾ \text{pu}_M) = (f ⨾ g)^{\uparrow M} \quad . \end{aligned}$$

If a monad M is specified via its Kleisli composition $\diamond_M$, we can first define `flatMap` by:

$$\text{flm}_M(f^{:A \to M^B}) \stackrel{\text{def}}{=} \text{id}^{:M^A \to M^A} \diamond_M f \quad ,$$

and then derive `map` from `flatMap` as before.

If a monad's code is defined via `pure` and `flatten`, we cannot derive the `map` method. So, we could indeed say heuristically that `flatten` has the functionality of `flatMap` without `map`.

In the previous chapter, we have seen that the laws of filterable functors are straightforwardly adapted to filterable contrafunctors. Could we do that with monads to obtain "contramonads"? It turns out that nontrivial "contramonads" do not exist.

The first problem is defining a method analogous to `flatMap` for contrafunctors. We turn to category theory (Section 9.4.3) for guidance. The type signature of a monad's `flatMap` is that of a lifting,

$$\text{flm}_M : (A \to M^B) \to M^A \to M^B \quad ,$$

corresponding to a functor from the M-Kleisli category to the M-lifted category. To adapt this construction to a contrafunctor H^A, we need to define a lifting called, say, "`contraFlatMap`", with the type:

$$\text{cflm}_H : (A \to H^B) \to H^B \to H^A \quad .$$

Note a curious feature of this type signature: it is covariant in B (since it contains the contrafunctor H^B in contravariant positions). So, cflm_H does not have the type of a natural transformation with respect to B, unlike all other liftings we have seen, such as `map`, `liftOpt`, and `flatMap`. If the code of cflm_H is fully parametric then cflm_H must satisfy a corresponding naturality law. It turns out that the naturality law gives a constraint so strong that it forces H to be a *constant* contrafunctor.

To see that, consider the left naturality law of cflm_H (with respect to the type parameter A):

$$\text{cflm}_H(f^{:A\to B} \fatsemi g^{:B\to H^C}) = \text{cflm}_H(g) \fatsemi f^{\downarrow H} \quad . \tag{10.22}$$

If H were a "contramonad", it would have a method analogous to `pure` with type signature $A \to H^A$ satisfying the right identity law:

$$\text{cflm}_H(\text{pu}_H) = \text{id} \quad .$$

Substitute $g = \text{pu}_H$ into Eq. (10.22) and obtain

$$\text{cflm}_H(f \fatsemi \text{pu}_H) = \text{id} \fatsemi f^{\downarrow H} = f^{\downarrow H} \quad . \tag{10.23}$$

The method `pure` for a contrafunctor H is equivalent to a value wu_H of type $H^{\mathbb{1}}$ (Section 8.3.8). In other words, the naturality law for $\text{pu}_H : A \to H^A$ forces it to be a function that ignores its argument and always returns a fixed value of type H^A, denoted by cpu_H in Sections 8.3.7–8.3.8:

$$\text{pu}_H = _^{:A} \to \text{cpu}^A_H = _^{:A} \to \text{wu}_H \triangleright (_^{:A} \to 1)^{\downarrow H} \quad .$$

Since pu_H ignores its argument, we have $f \fatsemi \text{pu}_H = \text{pu}_H$ for any function f. Then Eq. (10.23) gives:

$$f^{\downarrow H} = \text{cflm}_H(f \fatsemi \text{pu}_H) = \text{cflm}_H(\text{pu}_H) = \text{id} \quad .$$

The equation $f^{\downarrow H} = \text{id}$ holds only for constant contrafunctors $H^A \stackrel{\text{def}}{=} Z$ (with some fixed type Z). Constant contrafunctors are functors at the same time. We know from Section 10.2.8 that constant functors $F^A = Z$ are monads only when $Z = \mathbb{1}$. So, nontrivial "contramonads" are not possible.

10.3.2 Equivalence of a natural transformation and a "lifting"

In this and the previous chapters, we have derived the laws for three typeclasses: filterable functors, filterable contrafunctors, and monads. In each case, the laws were formulated in terms of lifting-like functions (`liftOpt`, `flatMap`), with type signatures:

$$\begin{aligned}
\text{for a filterable functor } F: &\quad \text{liftOpt}_F : (A \to \text{Opt}^B) \to F^A \to F^B \quad ,\\
\text{for a filterable contrafunctor } F: &\quad \text{liftOpt}_F : (A \to \text{Opt}^B) \to F^B \to F^A \quad ,\\
\text{for a monad } F: &\quad \text{flm}_F : (A \to F^B) \to F^A \to F^B \quad .
\end{aligned}$$

The laws were equivalently written in terms of natural transformations (`deflate`, `inflate`, `flatten`):

$$\begin{aligned} \text{for a filterable functor } F: &\quad \text{deflate}_F : F^{\text{Opt}^A} \to F^A \quad , \\ \text{for a filterable contrafunctor } F: &\quad \text{inflate}_F : F^A \to F^{\text{Opt}^A} \quad , \\ \text{for a monad } F: &\quad \text{ftn}_F : F^{F^A} \to F^A \quad . \end{aligned}$$

How can the liftings be *equivalent* to their corresponding natural transformations, which have fewer type parameters and simpler types? The equations relating these functions have a clear pattern:

$$\begin{aligned} &\text{deflate}_F = \text{liftOpt}_F(\text{id}) \quad , \qquad \text{liftOpt}_F(f) = f^{\uparrow F} \fatsemi \text{deflate} \quad , \\ &\text{inflate}_F = \text{liftOpt}_F(\text{id}) \quad , \qquad \text{liftOpt}_F(f) = \text{inflate} \fatsemi f^{\downarrow F} \quad , \\ &\text{ftn}_F = \text{flm}_F(\text{id}) \quad , \qquad \text{flm}_F(f) = f^{\uparrow F} \fatsemi \text{ftn}_F \quad . \end{aligned}$$

The pattern is that the code of the lifting-like function contains both the functionality of `map` (lifting the function f to $f^{\uparrow F}$ or $f^{\downarrow F}$) and the functionality of the corresponding natural transformation. Setting f to an identity function will cancel the lifting (since $\text{id}^{\uparrow F} = \text{id}$ and $\text{id}^{\downarrow F} = \text{id}$), so that only the functionality of the natural transformation remains.

We have proved the equivalence of lifting-like functions and their corresponding natural transformations for filterable functors in Statement 9.2.3.1 and for monads in Statement 10.2.2.1. It turns out that these equivalence properties are special cases of a more general construction. To generalize the type signatures of `deflate` and `flatten`, consider arbitrary functors F, G, H, and a natural transformation `tr` with the type signature:

$$\text{tr} : F^{G^A} \to H^A \quad .$$

A lifting-like function `ftr` corresponding to the natural transformation `tr` is defined by:

$$\text{ftr} : (A \to G^B) \to F^A \to H^B \quad , \qquad \text{ftr}\,(f^{:A\to G^B}) \stackrel{\text{def}}{=} f^{\uparrow F} \fatsemi \text{tr} \quad . \tag{10.24}$$

For filterable functors, we set $G = \text{Opt}$ and $F = H$ and for monads, we set additionally $G = F$.

By setting $f = \text{id}$ in Eq. (10.24), we find that $\text{tr} = \text{ftr}\,(\text{id})$. When are `tr` and `ftr` equivalent?

Statement 10.3.2.1 The natural transformation `tr` is equivalent to the "lifting-like" function `ftr` via:

$$\begin{tikzcd} & F^{G^B} \arrow[dr, "\text{tr}"] & \\ F^A \arrow[ur, "(f^{:A\to G^B})^{\uparrow F}"] \arrow[rr, "\text{ftr}\,(f^{:A\to G^B})"'] & & H^B \end{tikzcd}$$

$$\text{ftr}\,(f) = f^{\uparrow F} \fatsemi \text{tr} \quad , \qquad \text{tr} = \text{ftr}\,(\text{id}) \quad , \tag{10.25}$$

provided that `ftr` obeys the naturality law with respect to the type parameter A:

$$g^{\uparrow F} \fatsemi \text{ftr}\,(f) = \text{ftr}\,(g \fatsemi f) \quad . \tag{10.26}$$

Proof We need to derive the equivalence between `tr` and `ftr` in both directions:

(a) Given a function `tr`, we first define `ftr` via Eq. (10.24) and then define a new function `tr'` via Eq. (10.25). We then show that `tr'` equals `tr`:

$$\text{tr}' = \text{ftr}\,(\text{id}) = \underline{\text{id}^{\uparrow F}} \fatsemi \text{tr} = \text{tr} \quad .$$

If the function `ftr` is defined via `tr`, the naturality law (10.26) holds automatically:

$$g^{\uparrow F} \fatsemi \text{ftr}\,(f) = \underline{g^{\uparrow F} \fatsemi f^{\uparrow F}} \fatsemi \text{tr} = (g \fatsemi f)^{\uparrow F} \fatsemi \text{tr} = \text{ftr}\,(g \fatsemi f) \quad .$$

(b) Given a function `ftr` that satisfies the law (10.26), we define `tr` via Eq. (10.25) and then define a new function `ftr'` via Eq. (10.24). We then show that `ftr'` equals `ftr`. For an arbitrary $f^{:A\to G^B}$, write:

$$\begin{aligned} &\text{expect to equal ftr}(f): \quad \text{ftr}'(f) = f^{\uparrow F} \fatsemi \text{tr} = f^{\uparrow F} \fatsemi \text{ftr}(\text{id}) \\ &\text{naturality law (10.26)}: \quad = \text{ftr}(f \fatsemi \text{id}) = \text{ftr}(f) \quad . \end{aligned}$$

The full equivalence between `tr` and `ftr` would not hold without the naturality law (10.26). To build intuition for this, consider that `ftr` has a more complicated type signature with one more type parameter than `tr`. So, heuristically, there are "many more" possible functions with the type of `ftr`. The imposed naturality law (10.26) constrains the code of `ftr` so much that all possible functions `ftr` satisfying Eq. (10.26) are in a 1-to-1 correspondence with all possible functions `tr`.

□

A similar statement can be proved for contrafunctors F, by considering the type signatures:

$$\text{tr} : H^B \to F^{G^B} \quad , \qquad \text{ftr} : (A \to G^B) \to H^B \to F^A \quad .$$

An equivalence between `ftr` and `tr` also holds with functors F and the type signatures:

$$\text{tr} : H^A \to F^{G^A} \quad , \qquad \text{ftr} : (G^A \to B) \to H^A \to F^B \quad .$$

By referring to these constructions, we can omit detailed proofs of equivalence in all such cases.

10.3.3 Monads, effects, and runners

Monads allow us to compose "monadic programs" (i.e., values of type M^A for some monad M). For list-like and tree-like monads M, a value of type M^A represents a collection of values of type A and is a result of nested iterations. For a pass/fail monad M, a value of type M^A holds either a result of type A or information about a failure. However, for many other monads M, a monadic program $m : M^A$ represents a single value of type A wrapped by an "effectful wrapper" of some sort. In an application, we usually need to extract that value of type A from the "wrapper" m.

To see how this works in general, consider monads of function type (such as `Reader`, `State`, and `Cont`). These monads delay their computations until a runner is applied. A monadic program $m : M^A$ is a function whose body somehow wraps a value of type A, and a runner extracts that value. The function-type monads are often defined as case classes with a single part called `run`, for example:

```
final case class Reader[Z, A](run: Z => A)
final case class State[S, A](run: S => (A, S))
final case class Cont[R, A](run: (A => R) => R)
```

In Scala syntax, this makes monadic programs appear to have a method called `run`:

```
val s: State[S, A] = for { ... } yield { ... }   // A monadic program in the State monad.
val init: S = ???                                 // An initial state.
val result: A = s.run(init)._1                    // Run the monadic program and extract the result.
```

We can package this code into a `runner` function:

```
def runner[S, A](init: S): State[S, A] => A = _.run(init)._1
```

The runner for the `State` monad computes a final result value of type A by performing all the state updates and other computations encapsulated by a monadic program (`s: State[S, A]`). We say that a runner "runs the monad's effects" in order to extract the result value.

These examples motivate a general definition: A **runner** for a monad M is a function of type $M^A \to A$, which we will denote by $\theta^A : M^A \to A$.

To be useful in practice, a runner θ must satisfy certain laws. To motivate those laws, consider how we would use runners with a monadic program $m : M^B$ that is composed from two parts: the first part is a monadic program $m_1 : M^A$, and the second part is a function $m_2 : A \to M^B$ that depends on the result (of type A) of the first monadic program.

```
val m = for {  // m == m1.flatMap(m2)
  x <- m1
  y <- m2(x)
} yield y
```

The composition of m_1 and m_2 can be written as a functor block or as an application of the `flatMap` method,

$$m = m_1 \triangleright \text{flm}_M(m_2) \quad . \tag{10.27}$$

We may imagine that $\theta(m)$ first runs the effects of m_1 obtaining a value x, and then runs the effects of $m_2(x)$ obtaining a value y. So, it is natural to require that a runner θ applied to m should give the same results as applying θ to m_1, which extracts a value x, and then applying the runner to $m_2(x)$. We can formulate this requirement as a law called the runner's **composition law**:

```
runner(m) == runner(m2(runner(m1)))
```

$$m \triangleright \theta = m_1 \triangleright \theta \triangleright m_2 \triangleright \theta \quad .$$

Substituting m from Eq. (10.27), we get $m_1 \triangleright \text{flm}_M(m_2) ⨾ \theta = m_1 \triangleright \theta ⨾ m_2 ⨾ \theta$. Since this equation must hold for any m_1, we can simplify it to:

$$\text{flm}_M(m_2) ⨾ \theta = \theta ⨾ m_2 ⨾ \theta \quad .$$

Let us reformulate the composition law in terms of the monad M's `flatten` method:

$$m_2^{\uparrow M} ⨾ \text{ftn}_M ⨾ \theta = \theta ⨾ m_2 ⨾ \theta \quad .$$

We would like to move m_2 to the left of θ in the right-hand side, so that m_2 could drop out of the equation. To interchange the order of composition $\theta ⨾ m_2$, we assume a naturality law for θ:

$$f^{\uparrow M} ⨾ \theta = \theta ⨾ f \quad . \tag{10.28}$$

It follows that $\theta ⨾ m_2 ⨾ \theta = m_2^{\uparrow M} ⨾ \theta ⨾ \theta$ and that $\theta ⨾ \theta = \theta^{\uparrow M} ⨾ \theta$. The runner's composition law becomes:

$$\begin{tikzcd} M^{M^A} \arrow[r, "\theta^{\uparrow M}", "\theta^{M^A}"'] \arrow[d, "\text{ftn}_M"'] & M^A \arrow[d, "\theta"] \\ M^A \arrow[r, "\theta"] & A \end{tikzcd}$$

$$\text{ftn}_M ⨾ \theta = \theta ⨾ \theta = \theta^{\uparrow M} ⨾ \theta \quad . \tag{10.29}$$

Another reasonable requirement for runners is that a monadic value with an "empty effect", such as `pure(x)`, should be mapped by the runner to just `x`:

$$x \triangleright \text{pu}_M \triangleright \theta = x \quad , \quad \text{or equivalently:} \quad \text{pu}_M ⨾ \theta = \text{id} \quad .$$

This is the **identity law** of monad runners.

If a runner has fully parametric code (such as the runner for the `State` monad shown above), the naturality law (10.28) will hold automatically due to the parametricity theorem. However, monad runners are not always fully parametric. For example, the runner for the continuation monad shown in Section 10.1.9 uses Scala's `Future` and `Promise` classes, which involve mutable values. (Nevertheless, the naturality, identity, and composition laws hold for that runner.)

We have argued that list-like, tree-like, and pass/fail monads do not need runners; but those monads *cannot* have fully parametric runners. For instance, a runner for the `Option` monad should be a function with type $\theta^A : \mathbb{1} + A \to A$ that must produce a value of type A even if the `Option` value is empty. However, it is impossible to produce a value of an arbitrary type A from scratch using fully parametric code. For the `Option` monad and other pass/fail monads, we could only use runners θ^A that work for a specific type A, for example, for $A = \texttt{Int}$:

```
def runner: Option[Int] => Int = _.getOrElse(0) // For empty Option values, return a default.
```

Even if we restrict all types to `Int`, this runner will fail to obey the composition law:

```
val m1: Option[Int] = None
val m2: Int => Option[Int] = { x => Some(x + 1) }
val m: Option[Int] = for { x <- m1; y <- m2(x) } yield y

scala> runner(m)   // Composition law: runner(m) == runner(m2(runner(m1))).
res0: Int = 0

scala> runner(m2(runner(m1)))
res1: Int = 1
```

Similar arguments apply to list-like and tree-like monads. These monads may have an empty value, which cannot be correctly handled by a runner θ^A whose result must be of parametric type A.

A solution is to define runners not as functions of type $M^A \to A$ but as functions of type $M^A \to N^A$, where N is another monad (the "target" monad of the runner). For example, if M is a pass/fail monad, we may choose the target monad as $N^A = E + A$, where a fixed type E represents error information. We can then define a runner for the `Option` monad like this:

```
val error: E = ... // Describe the error.
def run[A]: Option[A] => Either[E, A] = {
  case None    => Left(error)
  case Some(a) => Right(a)
}
```

$$\theta^{:\mathbb{1}+A\to E+A} \stackrel{\text{def}}{=} \begin{array}{c||cc|} & E & A \\ \hline \mathbb{1} & _ \to \text{error} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} \quad .$$

In the next section, we will formulate the laws for runners of type $M^A \to N^A$.

10.3.4 Monads in category theory. Monad morphisms

For any monad M, one defines a category, called the M-Kleisli category, whose objects are all types (`Int`, `String`, etc.) and whose morphisms between types A and B are Kleisli functions of type $A \to M^B$.

One axiom of a category requires us to have an identity morphism $A \to M^A$ for every object A; this is the monad M's `pure` method, $\text{pu}_M : A \to M^A$. Another axiom is the associativity of morphism composition operation, which must combine functions of types $A \to M^B$ and $B \to M^C$ into a function of type $A \to M^C$. The Kleisli composition $\diamond_M$ is precisely that operation, and its associativity law, $f \diamond_M (g \diamond_M h) = (f \diamond_M g) \diamond_M h$, is equivalent to a law of `flatMap` (Statements 10.2.6.3–10.2.6.5).

So, a functor M is a monad if and only if the corresponding M-Kleisli category is lawful. This is an concise way of formulating the monad laws.

We have seen that, for some monads, proofs of the laws are easier when written in terms of Kleisli morphisms. It turns out that the properties of monad runners also have a concise formulation in the language of categories.

A monad M's `pure` method has type $A \to M^A$, while a runner θ_M has type $M^A \to A$. Since the type parameter A itself can be viewed as the identity monad $\text{Id}^A \stackrel{\text{def}}{=} A$, we can write the types as:

$$\text{pu}_M : \text{Id}^A \to M^A \quad , \qquad \theta_M : M^A \to \text{Id}^A \quad .$$

These two types can be generalized to a transformation between two monads M and N:

$$\phi : M^A \to N^A \quad .$$

As we have seen in the previous section, some monads M require runners of this more general type, rather than of type $M^A \to A$.

Another use case for the runner type $M^A \to N^A$ comes from our code for the continuation monad's runner. The code first transforms a value of type `Cont[R, A]` into a `Future[A]` and then waits for the `Future` to complete. So, `Cont`'s runner can be seen as a composition of two transformations:

$$\text{Cont}^{R,A} \xrightarrow{\theta_{\text{Cont-Future}}} \text{Future}^A \xrightarrow{\theta_{\text{Future}}} A \qquad\qquad \theta_{\text{Cont}} = \theta_{\text{Cont-Future}} \,⨾\, \theta_{\text{Future}} \quad .$$

The intermediate runner $\theta_{\text{Cont-Future}}$ converts continuation-based monadic programs into `Future`-based ones. That conversion will be compatible with the way we write and refactor functor blocks only if we impose laws similar to the laws of runners shown in the previous section. Lawful runners of type $M^A \to N^A$ are called "monad morphisms". The following definition states their laws:

Definition 10.3.4.1 A natural transformation $\phi^A : M^A \to N^A$, also denoted by $\phi : M \rightsquigarrow N$, is called a **monad morphism** between monads M and N if the following two laws hold:

$$\text{identity law of } \phi : \quad \text{pu}_M \,⨾\, \phi = \text{pu}_N \quad , \tag{10.30}$$

$$\text{composition law of } \phi : \quad \text{ftn}_M \,⨾\, \phi = \phi^{\uparrow M} \,⨾\, \phi \,⨾\, \text{ftn}_N \quad . \tag{10.31}$$

$$\begin{tikzcd} A \arrow[d, "\text{pu}_M"'] \arrow[dr, "\text{pu}_N"] & \\ M^A \arrow[r, "\phi"] & N^A \end{tikzcd} \qquad \begin{tikzcd} M^{M^A} \arrow[r, "\text{ftn}_M"] \arrow[d, "\phi^{\uparrow M}"'] & M^A \arrow[dr, "\phi"] & \\ M^{N^A} \arrow[r, "\phi^{N^A}"] & N^{N^A} \arrow[r, "\text{ftn}_N"] & N^A \end{tikzcd}$$

The composition law can be equivalently expressed using the `flatMap` method:

$$\text{flm}_M(f^{:A\to M^B}) ⨾ \phi^{:M^B\to N^B} = \phi ⨾ \text{flm}_N(f ⨾ \phi) \quad . \tag{10.32}$$

In terms of the Kleisli composition operations $\diamond_M$ and $\diamond_N$, the composition law is:

$$(f^{:A\to M^B} ⨾ \phi^{:M^B\to N^B}) \diamond_N (g^{:B\to M^C} ⨾ \phi^{:M^C\to N^C}) = (f \diamond_M g) ⨾ \phi^{:M^C\to N^C} \quad .$$

This formulation shows more visually that a monad morphism $\phi : M \rightsquigarrow N$ replaces M-effects by N-effects while preserving the composition of effectful computations.

The name "monad morphism" is motivated by considering the *category of monads*. The objects of that category are all the possible monads (type constructors such as `Option`, `Try`, etc.). The morphisms between objects M and N of that category are monad morphisms $M \rightsquigarrow N$ as defined above: natural transformations that preserve the structure of the monadic operations.

At the same time, a monad morphism ϕ can be viewed as a correspondence $g = f ⨾ \phi$ between Kleisli functions $f : A \to M^B$ and $g : A \to N^B$. The laws of monad morphisms guarantee that the mapping ϕ is compatible with the Kleisli composition operations $\diamond_M$ and $\diamond_N$. So, it can be seen also as expressing a (categorical) *functor* between the M-Kleisli and the N-Kleisli categories.

Monad morphisms between arbitrary monads will be used in Chapters 13 and 14. To build up more intuition, let us look at some examples of monad morphisms.

Example 10.3.4.2 Show that the function $\phi^A : Z + A \to \mathbb{1} + A$ defined below is a monad morphism between the `Either` and `Option` monads. The implementation of ϕ is:

```
def toOption[Z, A]: Either[Z, A] => Option[A] = {
  case Left(z)    => None
  case Right(a)   => Some(a)
}
```

$$\phi \stackrel{\text{def}}{=} \begin{array}{c||cc} & \mathbb{1} & A \\ \hline Z & _\to 1 & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} \quad .$$

Solution To verify the identity law (10.30):

$$\text{pu}_{\text{Either}} ⨾ \phi = \begin{array}{c||cc} & Z & A \\ \hline A & \mathbb{0} & \text{id} \end{array} ⨾ \begin{array}{c||cc} & \mathbb{1} & A \\ \hline Z & _\to 1 & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} = \begin{array}{c||cc} & \mathbb{1} & A \\ \hline A & \mathbb{0} & \text{id} \end{array} = \text{pu}_{\text{Opt}} \quad .$$

To verify the composition law (10.31), show that both sides are equal:

$$\text{left-hand side}: \quad \text{ftn}_{\text{Either}} ⨾ \phi = \begin{array}{c||cc} & Z & A \\ \hline Z & \text{id} & \mathbb{0} \\ Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} ⨾ \begin{array}{c||cc} & \mathbb{1} & A \\ \hline Z & _\to 1 & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} = \begin{array}{c||cc} & \mathbb{1} & A \\ \hline Z & _\to 1 & \mathbb{0} \\ Z & _\to 1 & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} \quad ,$$

$$\text{right-hand side}: \quad \phi^{\uparrow\text{Either}} ⨾ \phi ⨾ \text{ftn}_{\text{Opt}} = \begin{array}{c||ccc} & Z & \mathbb{1} & A \\ \hline Z & \text{id} & \mathbb{0} & \mathbb{0} \\ Z & \mathbb{0} & _\to 1 & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} ⨾ \begin{array}{c||ccc} & \mathbb{1} & \mathbb{1} & A \\ \hline Z & _\to 1 & \mathbb{0} & \mathbb{0} \\ \mathbb{1} & \mathbb{0} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} ⨾ \begin{array}{c||cc} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array}$$

$$= \begin{array}{c||ccc|} & \mathbb{1} & \mathbb{1} & A \\ \hline Z & _ \to 1 & \mathbb{0} & \mathbb{0} \\ Z & \mathbb{0} & _ \to 1 & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} \fatsemi \begin{array}{c||cc|} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} = \begin{array}{c||cc|} & \mathbb{1} & A \\ \hline Z & _ \to 1 & \mathbb{0} \\ Z & _ \to 1 & \mathbb{0} \\ A & \mathbb{0} & \text{id} \end{array} \quad .$$

The monad morphism ϕ maps the `Either`-effect (error information) to the `Option`-effect (absence of value) by discarding the error information. The laws of the monad morphism guarantee that composition of `Either`-effects is mapped into composition of `Option`-effects. For instance, if some computations caused an error encapsulated by `Either`, the corresponding value after ϕ will be an empty `Option` (i.e., `None`).

The function ϕ is so simple that the preservation of effects appears to be automatic. The next example shows a monad morphism that translates effects in a nontrivial way.

Example 10.3.4.3 Show that `toCont` is a monad morphism between the `Writer` and the `Cont` monads:

```
def toCont[W: Monoid, A]: ((A, W)) => (A => W) => W = { case (a, w) => k => k(a) |+| w }
```

$$\phi : A \times W \to (A \to W) \to W \quad , \qquad \phi \stackrel{\text{def}}{=} a^{:A} \times w^{:W} \to k^{:A \to W} \to k\,(a) \oplus w \quad ,$$

where the fixed type W is a monoid with a binary operation $\oplus$ and empty value e. The continuation monad's callback returns a value of type W that depends on the output w from the `Writer` monad.

Solution To verify the identity law, use the definitions of `pure` for the `Writer` and `Cont` monads:

$$\begin{aligned} &\text{expect to equal } \text{pu}_{\text{Cont}} : \quad \text{pu}_{\text{Writer}} \fatsemi \phi = (a \to a \times e) \fatsemi (a \times w \to k \to k\,(a) \oplus w) \\ &\text{compute composition} : \quad = a \to k \to k\,(a) \underline{\oplus e} = a \to k \to k\,(a) = \text{pu}_{\text{Cont}} \quad . \end{aligned}$$

To verify the composition law, we need an implementation of `flatten` for the `Cont` monad:

```
def flatten[W, A]: ((((A => W) => W) => W) => W) => (A => W) => W = { g => k => g(c => c(k)) }
```

$$\text{ftn}_{\text{Cont}} = g^{:(((A \to W) \to W) \to W) \to W} \to k^{:A \to W} \to g\,(c^{:(A \to W) \to W} \to c\,(k)) \quad .$$

Now we show that the two sides of the law (10.31) reduce to the same function:

$$\begin{aligned} &\text{left-hand side} : \quad \text{ftn}_{\text{Writer}} \fatsemi \phi = (a \times u \times w \to a \times (u \oplus w)) \fatsemi (a \times w \to k \to k\,(a) \oplus w) \\ &\qquad = a \times u \times w \to k \to k\,(a) \oplus u \oplus w \quad , \\ &\text{right-hand side} : \quad \phi^{\uparrow \text{Writer}} \fatsemi \phi^{\text{Cont}^{W,A}} \fatsemi \text{ftn}_{\text{Cont}} \\ &\qquad = \underline{(a \times u \times w \to (k \to k\,(a) \oplus u) \times w) \fatsemi (c \times w \to h \to h\,(c) \oplus w)} \fatsemi \text{ftn}_{\text{Cont}} \\ &\text{compute composition} : \quad = (a \times u \times w \to h \to h\,(k \to k\,(a) \oplus u) \oplus w) \fatsemi \underline{\text{ftn}_{\text{Cont}}} \\ &\qquad = (a \times u \times w \to h \to h\,(k \to k\,(a) \oplus u) \oplus w) \fatsemi (g \to p \to g(c \to c\,(p))) \\ &\text{compute composition} : \quad = a \times u \times w \to p \to \underline{(c \to c\,(p))(k \to k\,(a) \oplus u)} \oplus w \\ &\qquad = a \times u \times w \to p \to p\,(a) \oplus u \oplus w \quad . \end{aligned}$$

□

We conclude this section by proving some properties of monad morphisms.

Statement 10.3.4.4 If a function $\phi^A : M^A \to N^A$ satisfies Eqs. (10.30) and (10.32) then **(a)** the function ϕ is a natural transformation, and **(b)** the function ϕ also satisfies Eq. (10.31).

Proof **(a)** The naturality law of ϕ,

$$(g^{:A \to B})^{\uparrow M} \fatsemi \phi = \phi \fatsemi g^{\uparrow N} \quad , \tag{10.33}$$

is derived from Eq. (10.32) by setting $f^{:A\to M^B} \stackrel{\text{def}}{=} g^{:A\to B} \fatsemi \text{pu}_M^{:B\to M^B}$:

$$\begin{aligned}
\text{left-hand side of Eq. (10.32)}: \quad & \text{flm}_M(f) \fatsemi \phi = \underline{\text{flm}_M(f \fatsemi \text{pu}_M)} \fatsemi \phi \\
\text{left naturality of } \text{flm}_M: \quad & = f^{\uparrow M} \fatsemi \underline{\text{flm}_M(\text{pu}_M)} \fatsemi \phi \\
\text{right identity law of } \text{flm}_M: \quad & = f^{\uparrow M} \fatsemi \phi \quad , \\
\text{right-hand side of Eq. (10.32)}: \quad & \phi \fatsemi \text{flm}_N(f \fatsemi \phi) = \phi \fatsemi \text{flm}_N(g \fatsemi \underline{\text{pu}_M \fatsemi \phi}) \\
\text{identity law (10.30)}: \quad & = \phi \fatsemi \text{flm}_N(g \fatsemi \text{pu}_N) \\
\text{left naturality of } \text{flm}_N: \quad & = \phi \fatsemi g^{\uparrow N} \fatsemi \underline{\text{flm}_N(\text{pu}_N)} \\
\text{right identity law of } \text{flm}_N: \quad & = \phi \fatsemi g^{\uparrow N} \quad .
\end{aligned}$$

The two sides of Eq. (10.32) are equal to the two sides of Eq. (10.33).

(b) Substitute the definitions $\text{flm}_M(f) = f^{\uparrow M} \fatsemi \text{ftn}_M$ and $\text{flm}_N(f) = f^{\uparrow N} \fatsemi \text{ftn}_N$ into Eq. (10.32):

$$\begin{aligned}
& f^{\uparrow M} \fatsemi \text{ftn}_M \fatsemi \phi = \underline{\phi \fatsemi (f \fatsemi \phi)^{\uparrow N}} \fatsemi \text{ftn}_N \quad . \\
\text{use Eq. (10.33)}: \quad & = \underline{(f \fatsemi \phi)^{\uparrow M}} \fatsemi \phi \fatsemi \text{ftn}_N = f^{\uparrow M} \fatsemi \phi^{\uparrow M} \fatsemi \phi \fatsemi \text{ftn}_M \quad .
\end{aligned}$$

This equality holds for any f, in particular with $f = \text{id}$, which directly gives Eq. (10.31).

Statement 10.3.4.5 If L, M, N are monads and $\phi : L \rightsquigarrow M$ and $\chi : M \rightsquigarrow N$ are monad morphisms then the composition $\phi \fatsemi \chi : L \rightsquigarrow N$ is also a monad morphism.

Proof The identity law for $\phi \fatsemi \chi$ is verified by:

$$\begin{aligned}
\text{expect to equal } \text{pu}_N: \quad & \underline{\text{pu}_L \fatsemi (\phi \fatsemi \chi)} \\
\text{identity law for } \phi: \quad & = \text{pu}_M \fatsemi \chi \\
\text{identity law for } \chi: \quad & = \text{pu}_N \quad .
\end{aligned}$$

The composition law for $\phi \fatsemi \chi$ is verified by:

$$\begin{aligned}
\text{expect to equal } \text{ftn}_L \fatsemi \phi \fatsemi \chi: \quad & \underline{(\phi \fatsemi \chi)^{\uparrow L}} \fatsemi (\phi \fatsemi \chi) \fatsemi \text{ftn}_N \\
\text{naturality of } \phi: \quad & = \phi^{\uparrow L} \fatsemi \phi \fatsemi \underline{\chi^{\uparrow M} \fatsemi \chi \fatsemi \text{ftn}_N} \\
\text{composition law for } \chi: \quad & = \underline{\phi^{\uparrow L} \fatsemi \phi \fatsemi \text{ftn}_M} \fatsemi \chi \\
\text{composition law for } \phi: \quad & = \text{ftn}_L \fatsemi \phi \fatsemi \chi \quad .
\end{aligned}$$

Statement 10.3.4.6 For any monad M, the method $\text{pu}_M : A \to M^A$ is a monad morphism $\text{pu}_M : \text{Id} \rightsquigarrow M$ between the identity monad and M.

Proof The identity law requires $\text{pu}_{\text{Id}} \fatsemi \text{pu}_M = \text{pu}_M$. This holds because $\text{pu}_{\text{Id}} = \text{id}$. The composition law requires $\text{ftn}_{\text{Id}} \fatsemi \text{pu}_M = \text{pu}_M^{\uparrow \text{Id}} \fatsemi \text{pu}_M \fatsemi \text{ftn}_M$. Since $\text{ftn}_{\text{Id}} = \text{id}$, the left-hand side of the composition law simplifies to pu_M. Transform the right-hand side:

$$\begin{aligned}
\text{expect to equal } \text{pu}_M: \quad & \underline{\text{pu}_M^{\uparrow \text{Id}}} \fatsemi \text{pu}_M \fatsemi \text{ftn}_M \\
\text{lifting to the identity functor}: \quad & = \text{pu}_M \fatsemi \underline{\text{pu}_M \fatsemi \text{ftn}_M} \\
\text{left identity law for } M: \quad & = \text{pu}_M \quad .
\end{aligned}$$

Exercise 10.3.4.7 Suppose M is a given monad, Z is a fixed type, and a *fixed* value $m_0 : M^Z$ is given.

(a) Consider the function f defined as:

$$f : (Z \to A) \to M^A \quad , \qquad f\,(q^{:Z\to A}) \stackrel{\text{def}}{=} m_0 \triangleright q^{\uparrow M} \quad .$$

Prove that f is *not* a monad morphism from the `Reader` monad $R^A \stackrel{\text{def}}{=} Z \to A$ to the monad M^A, despite having the correct type signature.

(b) Under the same assumptions, consider the function ϕ defined as:

$$\phi : (Z \to M^A) \to M^A \quad , \qquad \phi\left(q^{:Z\to M^A}\right) \stackrel{\text{def}}{=} m_0 \triangleright \text{flm}_M(q) \quad .$$

Show that ϕ is *not* a monad morphism from the monad $Q^A \stackrel{\text{def}}{=} Z \to M^A$ to M^A.

10.3.5 Constructions of polynomial monads

"Polynomial monads" are polynomial functors that have lawful monad methods. The product and co-product constructions allow us to create polynomial monads via the following operations:

1. Start with $F^A \stackrel{\text{def}}{=} Z + W \times A$, which is a monad (semimonad) when W is a monoid (semigroup).

2. Given a polynomial monad F^A, create the monad $L^A \stackrel{\text{def}}{=} A + F^A$.

3. Given two polynomial monads F^A and G^A, create the monad $L^A \stackrel{\text{def}}{=} F^A \times G^A$.

4. Given a polynomial monad F^A, create the monad $L^A \stackrel{\text{def}}{=} F^{Z+W\times A}$ (see Section 14.4).

It is an open conjecture (no proof is known) that these are the only constructions available for polynomial monads. If so, we can create an algorithm that recognizes whether a given polynomial functor can be made into a monad by suitable definitions of `flatten` and `pure`.

As an example, consider the fact that the polynomial functor $F^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$ cannot be made into a monad (Exercise 10.2.9.8). One can also show that F^A cannot be obtained through the monad constructions listed above. Indeed, the corresponding polynomial $f(x) = 1 + x^2$ does not contain any first powers of x. However, all constructions either start with a polynomial containing x, or add x, or take a product of two such polynomials. None of these operations could cancel first powers of x since all coefficients are types and cannot be subtracted.

By the same logic, we can conclude that $F^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A \times A$ cannot be obtained through monad constructions. It is likely (although this book does not have a proof) that $F^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A \times A$, $F^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A \times A \times A$, and all other similarly constructed functors are *not* monads.

On the other hand, some polynomial functors can be obtained via the monad constructions in more than one way. A simple example is:

$$F^A \stackrel{\text{def}}{=} A + A \times A \cong A \times (\mathbb{1} + A) \quad .$$

The result is that the functor F has two *inequivalent* monad instances. The first instance is the free pointed monad (construction 2) on the pair monad $G^A \stackrel{\text{def}}{=} A \times A$ (construction 3). The second is the product (construction 3) of the identity monad ($\text{Id}^A \stackrel{\text{def}}{=} A$) and the `Option` monad ($\text{Opt}^A \stackrel{\text{def}}{=} \mathbb{1} + A$, construction 1). The two monad instances are inequivalent because, for instance, they define their `pure` method in different ways:

$$\text{pu}_1 \stackrel{\text{def}}{=} a^{:A} \to a + \mathbb{0}^{:A\times A} \quad , \qquad \text{pu}_2 \stackrel{\text{def}}{=} a^{:A} \to a \times (\mathbb{0}^{:\mathbb{1}} + a) \quad .$$

Both instances are lawful since the constructions preserve the monad laws.

When we say "a functor F is a monad", we mean that there exists *at least one* lawful monad instance for F. When working with a custom type constructor such as F, the programmer can first check whether F is a monad. When F has more than one lawful monad instance, the programmer will then need to choose the instance suitable for a given application.

10.3.6 Constructions of M-filterable functors and contrafunctors

In Chapter 9, we have used the `Opt`-Kleisli category (that is, the M-Kleisli category with M set to the `Option` monad) to formulate the laws of filterable (contra)functors. We found that the laws of a filterable (contra)functor F are equivalent to the requirement that the function `liftOpt` be a lawful (categorical) functor from the `Opt`-Kleisli category to an F-lifted category:

$$\text{filterable functor } F: \quad \text{liftOpt}_F : (A \to \text{Opt}^B) \to F^A \to F^B \quad ,$$
$$\text{filterable contrafunctor } F: \quad \text{liftOpt}_F : (A \to \text{Opt}^B) \to F^B \to F^A \quad .$$

It is natural to generalize this formulation from the `Option` monad to an arbitrary monad M:

$$M\text{-filterable functor } F: \quad \text{lift}_{M,F} : (A \to M^B) \to F^A \to F^B \quad ,$$
$$M\text{-filterable contrafunctor } F: \quad \text{lift}_{M,F} : (A \to M^B) \to F^B \to F^A \quad .$$

This gave us the definitions of M-filterable functors and contrafunctors.

We have shown some simple examples of M-filterable functors in Statement 10.2.8.8. Structural analysis can discover other examples of such functors systematically. As in the case of ordinary filterable functors, it turns out that we must at the same time analyze M-filterable contrafunctors.

In the following constructions, we always assume that M is a fixed, lawful monad.

We omit the proofs of all following statements because they are fully analogous to the proofs of filterable functor and contrafunctor constructions in Sections 9.2.4 and 9.2.6. In those proofs, we only used the properties of the functions liftOpt_F, which are fully analogous to the properties of the function $\text{lift}_{M,F}$ except for replacing the `Option` monad by the monad M and the operation $\diamond_{\text{Opt}}$ by $\diamond_M$.

Type parameters A constant functor $F^A \stackrel{\text{def}}{=} Z$ is M-filterable with $\text{lift}_{M,F}(f) = \text{id}$ (Statement 10.2.8.8). The same statement shows that $F^A \stackrel{\text{def}}{=} M^A \to Z$ and $F^A \stackrel{\text{def}}{=} A \to M^Z$ are M-filterable contrafunctors.

The monad M^A itself is M-filterable; $\text{lift}_{M,M}(f) \stackrel{\text{def}}{=} \text{flm}_M(f)$.

The identity functor is not M-filterable except when M is the identity monad, $M^A = \text{Id}^A \stackrel{\text{def}}{=} A$. (However, with $M = \text{Id}$, the concept of M-filterable functor becomes trivial because all functors and all contrafunctors are Id-filterable. So, we will assume that $M \neq \text{Id}$.)

The (contra)functor $F^A \stackrel{\text{def}}{=} G^{H^A}$ is M-filterable when H is M-filterable and G is any (contra)functor (Statements 9.2.4.1 and 9.2.6.1), where G and H can be functors or contrafunctors independently.

Products If F^A and G^A are M-filterable then $L^A \stackrel{\text{def}}{=} F^A \times G^A$ is M-filterable (Statement 9.2.4.2).

Co-products If F^A and G^A are M-filterable then $L^A \stackrel{\text{def}}{=} F^A + G^A$ is M-filterable (Statement 9.2.4.3).

Function types If F^A is an M-filterable functor and G^A is an M-filterable contrafunctor then $F^A \to G^A$ and $G^A \to F^A$ are M-filterable (contra)functors (Statements 9.2.4.5 and 9.2.6.2).

The filterable contrafunctor $M^A \to Z$ from Statement 10.2.8.8 is obtained from this construction if we set $F^A = M^A$ and $G^A = Z$; both F and G are filterable as we have already seen.

Recursive types If $S^{A,R}$ is a bifunctor that is M-filterable with respect to A, the recursive functor F^A defined by the type equation $F^A \stackrel{\text{def}}{=} S^{A,F^A}$ is M-filterable (Statement 9.2.4.7).

If $S^{A,R}$ is a profunctor contravariant in A and covariant in R, and additionally $S^{\bullet,R}$ is M-filterable (with the type parameter R fixed), then the recursive contrafunctor $F^A \stackrel{\text{def}}{=} S^{A,F^A}$ is M-filterable (see Statement 9.2.6.5).

In summary, any recursively exponential-polynomial type expression F^A will be an M-filterable (contra)functor if it depends on a type parameter A only through type expressions M^A or $A \to M^Z$ (where Z is a fixed type). While there may be other M-filterable (contra)functors, the structural analysis covers a broad class of type expressions of the form:

$$S^{M^A, A \to M^{Z_1}, \ldots, A \to M^{Z_n}} \quad ,$$

where $S^{A,B_1,\ldots,B_n}$ is a type constructor covariant in A and contravariant in B_1, ..., B_n (or vice versa), and Z_1, ..., Z_n are fixed types.

11 Applicative functors and contrafunctors

11.1 Motivation and first examples

In previous chapters, we generalized the `map`, `filter`, and `flatMap` methods from sequences to other type constructors that support such methods obeying suitable laws. Following the same path, we now turn to the `zip` method. Although `zip` is most often used with sequences, many other type constructors can also have a suitable `zip` method. Those type constructors are known as **applicative**.

11.1.1 Generalizing the `zip` method from sequences to other types

Chapter 2 showed the use of the `zip` operation for sequences and other collections. For lists, the `zip` method is equivalent to a function with the following type signature:

```
def zip[A, B](la: List[A], lb: List[B]): List[(A, B)]
```

$$\text{zip} : \text{List}^A \times \text{List}^B \to \text{List}^{A\times B} \quad .$$

It turns out that a broad class of type constructors L can have a similar `zip` method:

$$\text{zip} : L^A \times L^B \to L^{A\times B} \quad .$$

Using this type signature, the `zip` operation may be implemented for many type constructors, not only for `List`-like collections. In order to ensure that the implementation of `zip` is useful and safe, we will establish and verify the laws of the `zip` operation later in this chapter. For now, let us look at some examples of implementing a `zip` operation.

Example 11.1.1.1 A `zip` method can be implemented for the type constructor $L^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$:

```
type L[A] = Option[(A, A)]
def zip[A, B](la: L[A], lb: L[B]): L[(A, B)] = (la, lb) match {
  case (None, _) | (_, None)              => None
  case (Some((a1, a2)), Some((b1, b2)))   => Some(((a1, b1), (a2, b2)))
}

scala> zip(Some((123, 456)), Some(("abc", "def")))
res0: L[(Int, String)] = Some(((123, "abc"), (456, "def")))
```

Example 11.1.1.2 A `zip` method can be implemented for a binary tree. We define the type constructor `BTree[A]` by this code:

```
sealed trait BTree[A]
final case class Leaf[A](a: A) extends BTree[A]
final case class Branch[A](left: BTree[A], right: BTree[A]) extends BTree[A]
```

The `zip` method should have the following type signature:

```
def zip[A, B](ta: BTree[A], tb: BTree[B]): BTree[(A, B)] = ???
```

We would like to preserve the tree structure of `ta` and `tb` as much as possible, zipping together subtrees of equal shape without adding any new branchings. For example, zip( ) should evaluate to the tree . If a subtree of `ta` is a `Leaf(x)` while the corresponding

subtree of `tb` is a `Branch`, the value `x` must be replicated to match the subtree of `tb`. So, the result of evaluating zip( ) should be with replicated b.

Solution Begin writing the pattern-matching code:

```
def zip[A, B](ta: BTree[A], tb: BTree[B]): BTree[(A, B)] = (ta, tb) match {
  case (Leaf(x), Leaf(y))                  => Leaf((x, y))
  case (Branch(lx, rx), Leaf(y))           => ???
  case (Leaf(x), Branch(ly, ry))           => ???
  case (Branch(lx, rx), Branch(ly, ry))    => ???
}
```

The type constructor `BTree` is a functor and has a `map` method:

```
def map[A, B](ta: BTree[A])(f: A => B): BTree[B] = ta match {
  case Leaf(a)          => Leaf(f(a))
  case Branch(ta, tb)   => Branch(map(ta)(f), map(tb)(f))
}
```

When "zipping" a `Leaf` with a `Branch`, we use the `map` method to replicate the value from the leaf:

```
  case (Branch(lx, rx), Leaf(y))  => map(Branch(lx, rx))(x => (x, y))
  case (Leaf(x), Branch(ly, ry))  => map(Branch(ly, ry))(y => (x, y))
```

For the case of two `Branch` values, we use two recursive calls to `zip`:

```
  case (Branch(ax, ay), Branch(bx, by))   => Branch(zip(ax, bx), zip(ay, by))
```

The final code of the `zip` method, after some simplifications, becomes:

```
def zip[A, B](ta: BTree[A], tb: BTree[B]): BTree[(A, B)] = (ta, tb) match {
  case (Leaf(x), Leaf(y))                  => Leaf((x, y))
  case (xa, Leaf(b))                       => map(xa)(a => (a, b))
  case (Leaf(a), xb)                       => map(xb)(b => (a, b))
  case (Branch(ax, ay), Branch(bx, by))    => Branch(zip(ax, bx), zip(ay, by))
}
```

To test our code, let us run the given examples and verify that we get the required results:

```
val ta: BTree[Int] = Branch(Branch(Leaf(1), Leaf(2)), Leaf(3))
val tb: BTree[String] = Branch(Leaf("b"), Leaf("c"))
val tc: BTree[String] = Branch(Branch(Leaf("a"), Branch(Leaf("d"), Leaf("e"))), Leaf("f"))

scala> zip(ta, ta)
res0: BTree[(Int, Int)] = Branch(Branch(Leaf((1, 1)), Leaf((2, 2))), Leaf((3, 3)))

scala> zip(tb, tc)
res1: BTree[(String, String)] = Branch(Branch(Leaf(("b", "a")), Branch(Leaf(("b", "d")), Leaf(("b",
    "e")))), Leaf(("c", "f")))
```

□

The `zip` operation is sometimes defined even for type constructors that are *not* functors:

Example 11.1.1.3 A pair of monoids is also a monoid (the "products" construction of Section 8.3.4). A `Monoid` typeclass instance for a type `A` can be represented by a value of type L^A, where the type constructor L is defined by:

```
type L[A] = (A, (A, A) => A)
```

$$L^A \stackrel{\text{def}}{=} A \times (A \times A \to A) \quad .$$

We can write a function that creates a monoid typeclass instance for the type $A \times B$ when A and B are monoids. That function has the type signature of a `zip` method for the type constructor L:

```
def zip[A, B](la: L[A], lb: L[B]): L[(A, B)] = ( (la._1, lb._1),   // The result has type ((A, B),
  { case ((a1, b1), (a2, b2)) => (la._2(a1, a2), lb._2(b1, b2)) }  // ((A, B), (A, B)) => (A, B)).
)
```

Applying this function to any two monoid instances, we obtain an instance value for the pair:

```
def monoidPair[A, B](implicit ma: Monoid[A], mb: Monoid[B]): Monoid[(A, B)] = zip(ma, mb)
```

Several typeclasses (such as the `Monoid`) have instances whose type constructors are neither covariant nor contravariant yet have a `zip` method. If we implement a `zip` method for typeclass instances, we will obtain automatic typeclass derivation for product types (e.g., case classes).

11.1.2 Gathering all errors during computations

A monadic program using pass/fail monads must stop at the first failure: the code `flatMap(x => expr)` cannot start evaluating `expr` if a previous computation failed to produce a value for `x`. However, if some pass/fail computations are independent of each other's results, we may wish to run all those computations and gather all errors.

As an example, consider the task of implementing "safe arithmetic" where a division by zero or square root of a negative number will give error messages (see Example 3.2.2.4). To be specific, let us perform the computation `(1 / 0) + (2 / 0)` in the "safe arithmetic". A monadic implementation (see Section 10.1.3) will stop the computation after the first error:

```
type Result[A] = Either[String, A]
def add(x: Int, y: Int): Result[Int] = Right(x + y)
def div(x: Int, y: Int): Result[Int] = if (y == 0) Left(s"error: $x / $y") else Right(x / y)

scala> for {
         x <- div(1, 0)
         y <- div(2, 0)
         z <- add(x, y)
       } yield z
res0: Either[String, Int] = Left(error: 1 / 0)
```

We notice that the two `div` operations do not depend on each other and may be computed separately. To achieve this, we define a `map2` function for the type constructor `Result`:

```
def map2[A, B, C](ra: Result[A], rb: Result[B])(f: (A, B) => C): Result[C] = (ra, rb) match {
  case (Left(e1), Left(e2))     => Left(e1 + "\n" + e2)    // Messages are separated by a newline.
  case (Left(e1), _)            => Left(e1)
  case (_, Left(e2))            => Left(e2)
  case (Right(a), Right(b))     => Right(f(a, b))
}
```

We can now use the `map2` function to compute the two `div` operations and gather the errors:

```
scala> for {
         p <- map2(div(1,0), div(2,0)) { (x, y) => (x, y) }    // Create a tuple (x, y).
         z <- add(p._1, p._2)
       } yield z
res1: Either[String, Int] = Left(error: 1 / 0
error: 2 / 0)
```

The result of `map2` is used in further monadic computations. In this way, we can combine code that gathers many errors with ordinary pass/fail monadic code that stops at the first error.

This example can be generalized to the type `Either[E, A]`, where the type `E` is a semigroup with a `combine` operation (denoted by `|+|`). In other words, error messages of type `E` can be combined in some way. Let us implement a `zip` method that works similarly to `map2` by gathering all error messages:

```
def zipE[A, B, E: Semigroup](x: Either[E, A], y: Either[E, B]): Either[E, (A, B)] = (x, y) match {
  case (Left(e1), Left(e2))   => Left(e1 |+| e2)
  case (Right(a), Left(e2))   => Left(e2)
  case (Left(e1), Right(b))   => Left(e1)
  case (Right(a), Right(b))   => Right((a, b))
}
```

In the code notation, the `zipE` function is written like this:

$$\text{zip}_E : (E+A)\times(E+B) \to E + A\times B \quad , \qquad \text{zip}_E \stackrel{\text{def}}{=} \begin{array}{|c||cc|} & E & A\times B \\ \hline E\times E & e_1\times e_2 \to e_1 \oplus e_2 & \mathbb{0} \\ A\times E & a\times e_2 \to e_2 & \mathbb{0} \\ E\times B & e_1\times b \to e_2 & \mathbb{0} \\ A\times B & \mathbb{0} & a\times b \to a\times b \end{array} \quad .$$

When both arguments of `zipE` have error messages, the code should not drop one of the errors and return `Left(e1)`, say (although that would still conform to the type signature of `zipE`). Instead, our code combines both error messages; this preserves more information.

Comparing the code of `map2` and `zip`, we find only one difference. Namely, the code of the `zip` function can be obtained from the code of `map2` if we replace `f` by an identity function. We will see later that this correspondence between `zip` and `map2` works for all applicative functors.

11.1.3 Monadic programs with independent effects

Another motivation for applicative functors comes from considering monadic programs where some source lines do not depend on previous results. As an example, consider the following monadic program involving `Future`:

```
val result: Future[Int] = for {
  x <- Future { func1() }  // func1() returns an Int.
  y <- Future { func2() }  // Similarly for func2().
  z <- Future { func3() }  // Similarly for func3().
} yield x + y + z
```

Recall that Scala's `Future` values are computations that have been scheduled to run on other threads (and may be already running). In this example, the computations in `func1()`, `func2()`, and `func3()` do not depend on the previously computed values (`x` and `y`). So, we would like to run those computations in parallel. However, the monadic program will create a second `Future` value and schedule the computation `func2()` only after `func1()` is done. Similarly, `func3()` will not start until `func2()` is done. This is because the functor block shown above is translated into `flatMap` and `map` methods like this:

```
val result: Future[Int] =
  Future { func1() }.flatMap { x =>
    Future { func2() }.flatMap { y =>
      Future { func3() }.map { z => x + y + z } } }
```

The implementations of `flatMap` and `map` cannot detect whether the next computations actually depend on any previously computed values. So, the code of the `flatMap` and `map` methods *must* wait until all previous values are computed. We need a new function that can take advantage of the independence of effects. A possible type signature for that function is:

```
def map3(f1: Future[Int], f2: Future[Int], f3: Future[Int], f: (Int, Int, Int) => Int): Future[Int]
```

Generalizing this type signature to arbitrary types, we obtain the following `map3` method:

```
def map3[F[_], A, B, C, D](f1: F[A], f2: F[B], f3: F[C])(f: (A, B, C) => D): F[D]
```

It is straightforward to implement this method for `F = Future`:

```
def map3[A, B, C, D](f1: Future[A], f2: Future[B], f3: Future[C])(f: (A, B, C) => D): Future[D] =
  for { x <- f1; y <- f2; z <- f3 } yield f(x, y, z)
```

This will run the three `Future` computations in parallel because the arguments of `map3` must be evaluated before `map3` is called, and evaluating a `Future` will schedule its computation immediately. To support different number of arguments, we can implement similar functions `map2`, `map4`, etc.

It is not obvious that the function `map3` is related to `zip`. However, once we study the properties of those functions in more detail, we will find that `map3` can be expressed through `map2`, and that defining `map2` is equivalent to defining `map` and `zip` (assuming that certain naturality laws hold for these functions). So, applicative functors can be viewed as functors having either a `map2` or a `zip` method in addition to `map`. We will also see that `map4`, `map5`, etc., can be all expressed through `zip`.

11.2 Practical use of applicative functors

Applicative functors appear whenever it is useful to have a `zip` or a `map2` operation. We will now look at a few specific situations where applicative operations help write better code.

11.2.1 Transposing a matrix via `map2`

The standard sequence type (`Seq`) already has a `zip` method, so implementing a `map2` for `Seq` is simple:

```
def map2[A, B, C](as: Seq[A], bs: Seq[B])(f: (A, B) => C): Seq[C] =
  (as zip bs).map { case (a, b) => f(a, b) }

scala> map2(List(1, 2), List(100, 200))(_ + _)
res0: Seq[Int] = List(101, 202)
```

In Example 10.1.2.8(a), we implemented matrix transposition using `flatMap` and index-based access to sequences, which is not available for some sequence types. The index-based access is avoided if we implement `transpose` via `map2`. As before, a matrix is represented by a sequence of sequences:

```
def transpose[A](s: Seq[Seq[A]]): Seq[Seq[A]] = ???

scala> transpose(List(List(1, 2), List(3, 4), List(5, 6)))
res1: Seq[Seq[Int]] = List(List(1, 3, 5), List(2, 4, 6))
```

We would like to define `transpose` by induction. The base case is an "empty" matrix:

```
def transpose[A](s: Seq[Seq[A]]): Seq[Seq[A]] = s.headOption match {
  case None          => Seq()
  case Some(heads)   => val tails = s.tail
                        ???
}
```

For the inductive step, we assume that the `tails` sub-matrix is already transposed. So, it remains to combine `transpose(tails)` with `heads`. Consider the test example shown above:

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{vmatrix} \triangleright \text{transpose} = \begin{vmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{vmatrix} \quad , \quad \begin{vmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{vmatrix} \triangleright \text{head} = \begin{vmatrix} 1 & 2 \end{vmatrix} \quad , \quad \begin{vmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{vmatrix} \triangleright \text{tail} = \begin{vmatrix} 3 & 4 \\ 5 & 6 \end{vmatrix} \quad .$$

Splitting this matrix into the first row (`heads`) and the rest (`tails`), we will obtain `heads == List(1, 2)` and `tails == List(List(3, 4), List(5, 6))`. The inductive assumption is that `transpose(tails)` works correctly and yields `List(List(3, 5), List(4, 6))`. How to combine this with `heads == List(1, 2)` to obtain the desired result `List(List(1, 3, 5), List(2, 4, 6))`? We need to prepend each element of `heads` to the corresponding sub-list in `transpose(tails)`. This is implemented by using `map2`:

```
def transpose[A](s: Seq[Seq[A]]): Seq[Seq[A]] = s.headOption match {
  case None          => Seq()
  case Some(heads)   =>
    val tails = s.tail
    map2(heads, transpose(tails)) { (x, y) => x +: y }
}
```

This code is still incomplete: it returns an empty sequence for all arguments. We need to add another base case when the matrix `s` has only one row (so `tails` is empty). Here is the final code:

```
def transpose[A](s: Seq[Seq[A]]): Seq[Seq[A]] = s.headOption match {
  case None          => Seq()
  case Some(heads)   => s.tail match {
    case Seq()   => heads.map(a => Seq(a))
    case tails   => map2(heads, transpose(tails)) { (x, y) => x +: y }
  }
}
```

11.2.2 Data validation with error reporting

Suppose we need to create a data structure with "validated" parts (the data type could be a simple tuple, a case class, a list, a tree, etc.). While validating any part of the data structure, we may get an error. This situation happens when validating Web input data, reading a set of command-line options, or parsing formatted text (such as JSON). We would like to report all errors to the user, rather than stopping at the first error.

To simplify the reasoning, assume that the required data structure is a case class, such as:

```
final case class MyData(userId: Long, userName: String, userEmails: List[String])
```

More generally, consider the type $A \times B \times C$ containing values of some chosen types A, B, and C:

```
final case class MyData[A, B, C](a: A, b: B, c: C)
```

We assume that the validation functions will return values of types `F[A]` defined by

```
type F[A] = Either[E, A]
```

$$F^A \stackrel{\text{def}}{=} E + A \quad .$$

Here, `A` is the type of a successfully validated value, while the fixed type `E` describes the error information. We will assume that E is a semigroup, so that different pieces of error information can be combined into a larger value of the same type E.

If some of the validations fail, the result should be a value of type E. So, our task is to create a value of type $E + A \times B \times C$ given values of types $E + A$, $E + B$, and $E + C$. Using the type constructor F and the type `MyData[A, B, C]`, we write the requirements as the following type signature:

```
def validated[A, B, C](fa: F[A], fb: F[B], fc: F[C]): F[MyData[A, B, C]] = ???
```

In the type notation, this is written as:

$$\text{validated} : F^A \times F^B \times F^C \to F^{A \times B \times C} \quad .$$

If we implement this type signature, we will be able to perform data validation while collecting all errors. Note that the type signature of `validated` is quite similar to that of `zip` except for using *three* different types rather than two. So, let us define a suitable function `zip3`. The code of `zip3` uses the ordinary `zip` method twice:

```
def zip3[A, B, C](fa: F[A], fb: F[B], fc: F[C]): F[(A, B, C)] =
  zip(fa, zip(fb, fc)).map { case (a, (b, c)) => (a, b, c) }
```

Also, `validated` has to insert a `MyData` type constructor, returning a value of type `F[MyData[A, B, C]]` rather than `F[(A, B, C)]`. So, we may implement `validated` as:

```
def validated[A, B, C](fa: F[A], fb: F[B], fc: F[C]): F[MyData[A, B, C]] =
  zip(fa, zip(fb, fc)).map { case (a, (b, c)) => MyData(a, b, c) }
```

The difference between `zip3` and `validated` is only in the function used in the last `map` step. So, we can make that last function a parameter and define the operation `map3` like this:

```
def map3[A, B, C, D](fa: F[A], fb: F[B], fc: F[C])(f: (A, B, C) => D): F[D] =
  zip(fa, zip(fb, fc)).map { case (a, (b, c)) => f(a, b, c) }
```

Now the validation function can be implemented via `map3` more concisely:

```
def validated[A, B, C](fa: F[A], fb: F[B], fc: F[C]): F[MyData[A, B, C]] =
  map3(fa, fb, fc)(MyData.apply)
```

11.2.3 Implementing the functions `map2`, `map3`, etc. The `ap` method

Working with applicative functors involves using methods such as `map2`, `map3`, and so on. The code of `map2` for the functor $L^A \stackrel{\text{def}}{=} \text{String} + A$ contains a pattern match with four cases (see Section 11.1.2). Writing similar code for `map3` requires *eight* cases (to match a triple of `Either` values), and `map4` would require 16 cases. How can we implement all these functions without writing a lot of code?

One idea is to use the `zip` method repeatedly, as we saw in the previous section. We would then implement `map2` through `zip`, `map3` through `zip3`, and so on. For the purposes of illustration, let us choose the types of all elements to be the same, so that the type signatures become:

$$\text{zip} : L^A \times L^A \to L^{A\times A} \quad , \qquad \text{zip}_3 : L^A \times L^A \times L^A \to L^{A\times A\times A} \quad .$$

We can now implement a general function (zip_n) that uses a list of values of type L^A:

```
def zipN[A](xs: List[Either[String, A]]): Either[String, List[A]] = xs match {
  case Nil            => Right(Nil)
  case head :: tail   => zipE(head, zipN(tail)).map { case (h, t) => h :: t }
}
```

The corresponding function `mapN` is then defined by

```
def mapN[A, Z](xs: List[Either[String, A]])(f: List[A] => Z): Either[String, Z] = zipN(xs).map(f)
```

If we wanted to define general functions `zipN` and `mapN` that could take N arguments of arbitrary types (and not all of type A), we would need to use techniques of dependent-type programming, which is beyond the scope of this book. We will now describe a simpler solution that implements `mapN` via a helper function (`ap`) that performs a recursion step expressing map_N through map_{N-1}.

If the `ap` method can do that, it can also express `map2` via `map`. So, let us determine the functionality that is present in `map2` but missing from `map`. That missing functionality is what `ap` must implement.

To compare `map` and `map2` functions more easily, consider their curried versions `fmap` and `fmap2`:

```
def fmap[A,B](f: A => B): L[A] => L[B]
def fmap2[A,B,C](f: A => B => C): L[A] => L[B] => L[C]
```

$$\text{fmap} : (A \to B) \to L^A \to L^B \quad ,$$
$$\text{fmap}_2 : (A \to B \to C) \to L^A \to L^B \to L^C \quad .$$

If we try implementing `fmap2` via `map`, we get stuck:

```
def fmap2[A, B, C](f: A => B => C): L[A] => L[B] => L[C] = { la: L[A] => la.map(f) ??? }
```

The value `la.map(f)` has type `L[B => C]` instead of the required type `L[B] => L[C]`. So, we need a function that converts between those types. The new function is called `ap` and has the type signature:

```
def ap[B, C](lf: L[B => C]): L[B] => L[C]
```

$$\text{ap}_L : L^{A\to B} \to L^A \to L^B \quad .$$

If we have an implementation of `ap` for a functor L, we can write the code for `fmap2` as:

```
def fmap2[A, B, C](f: A => B => C): L[A] => L[B] => L[C] = { la: L[A] => ap[B, C](la.map(f)) }
```

Written in the point-free style using the code notation, this definition looks like this:

$$\text{fmap}_2(f) \stackrel{\text{def}}{=} f^{\uparrow L} ⨾ \text{ap}_L \quad .$$

Taking into account the curried type signature of `fmap2`, we may rewrite this as `fmap2(f)(la)(lb) == ap(la.map(f))(lb)`. This expression is made clearer if we implement infix syntax for `fmap` and `ap`:

```
implicit class FmapSyntax[A, B](f: A => B) {
  def <@>[F[_] : Functor](fa: F[A]): F[B] = fa.map(f)
}
implicit class ApSyntax[A, B](lab: L[A => B]) {
  def <*>(la: L[A]): L[B] = ap(lab)(la)
}
```

Then we rewrite `fmap2(f)(la)(lb)` as `f <@> la <*> lb`. This works because all infix operators group to the left, so `f <@> la <*> lb` means `(f <@> la) <*> lb`, which is translated into `ap(la.map(f))(lb)`.

The main advantage of this new syntax is in making it easier to implement `fmap3`, `fmap4`, etc. Let us see how we can use the `ap` method to implement `fmap3`:

```
def fmap3[A, B, C, D](f: A => B => C => D): L[A] => L[B] => L[C] => L[D] = { la: L[A] =>
  ap[B, C => D](la.map(f)) andThen ap[C, D] }
```

In the point-free style, this definition is written as:

$$\text{fmap}_3(f) \stackrel{\text{def}}{=} f^{\uparrow L} \fatsemi \text{ap}_L \fatsemi (g \to g \fatsemi \text{ap}_L) \quad .$$

It is harder to write and to understand the code of `fmap3` than that of `fmap2`. An implementation of `fmap4` via `ap` will be even more complicated. But with the infix syntax shown above, we can write:

```
def fmap3[A, B, C, D](f: A => B => C => D)(a: L[A])(b: L[B])(c: L[C]): L[D] = f <@> a <*> b <*> c
def fmap4[A, B, C, D, E](f: A => B => C => D => E)(a: L[A])(b: L[B])(c: L[C])(d: L[D]): L[E] =
  f <@> a <*> b <*> c <*> d
```

These examples show how to implement the function fmap_N for any N and any argument types.

For type constructors that are applicative but not covariant, a method analogous to `map2` must have a different type signature. For instance, consider the type constructor L defined in Example 11.1.1.3. We cannot define `map2` as a composition of `zip` and `map` because L^A does not have a `map` method (as L is not covariant in A). Instead, we note that L is a profunctor and supports an `xmap` method:

```
def xmap[A, B](la: L[A])(f: A => B, g: B => A): L[B] = (f(la._1), (b1, b2) => f(la._2(g(b1), g(b2))))
```

Composing `zip` with `xmap`, we obtain a new method we may call `xmap2`:

```
def xmap2[A,B,C](la: L[A], lb: L[B])(f: ((A,B)) => C, g: C => (A,B)): L[C] = xmap(zip(la, lb))(f, g)
```

Methods such as `xmap3`, `xmap4`, etc., may be defined similarly via `zip3`, `zip4`, etc. However, a method analogous to `ap` does not exist for applicative profunctors.

11.2.4 The applicative `Reader` functor

The `Reader` functor, $L^A \stackrel{\text{def}}{=} E \to A$, is applicative and supports a `zip` method as well:

```
type Reader[A] = E => A    // The fixed type E must be already defined.
def zip[A, B](ra: Reader[A], rb: Reader[B]): Reader[(A, B)] = { e => (ra(e), rb(e)) }
```

The `map2` method is implemented similarly:

```
def map2[A, B, C](ra: Reader[A], rb: Reader[B])(f: A => B => C): Reader[C] = { e => f(ra(e))(rb(e)) }
```

These are the *only* fully parametric implementations of the type signatures of `zip` and `map2` for `Reader`.

Since `Reader` is also a monad (see Section 10.1.5), we may implement the type signature of `map2` as:

```
def map2[A, B, C](ra: Reader[A], rb: Reader[B])(f: A => B => C): Reader[C] = for {
  x <- ra
  y <- rb
} yield f(x)(y)
```

This code is fully parametric. So, this `map2` must be equal to the direct implementation shown above.

It is noteworthy that the `Reader`'s applicative and monad instances agree on the implementation of `map2`. We can understand this intuitively if we consider that `Reader`'s effect is a dependency on a constant "environment" (a value of type E). All `Reader` computations in a given functor block will read the same value of the "environment". So, the `Reader` effects are always independent, and a `map2` function does not need to be implemented separately (it can be expressed via `flatMap`).

11.2.5 Single-traversal `fold` operations. I. Applicative "fold fusion"

We have seen various "`fold`-like" operations (such as `foldLeft`, `foldRight`, or `reduce`) that iterate over a sequence and accumulate a result value. An example is the computation of the average of a list of numbers (see Example 1.4.1.4):

```
def average(s: List[Double]): Double = s.sum / s.size
```

Both operations `s.sum` and `s.size` will iterate over the list `s`. So, `average(s)` iterates *twice* over `s`.

It is sometimes undesirable (or impossible) to iterate over a sequence more than once. An example of that situation is a data stream whose data elements arrive over a network at high speed. Usually, the stream data cannot be stored because the data volume would grow too quickly. So, the program needs to be implemented as an aggregation operation that uses a single traversal of the stream.

Another example is the computation of word distribution statistics in a large text corpus.[1] A "corpus" is a sequence of large chunks of text. Each chunk takes a long time to download, and storing the entire sequence in memory is impossible. It is necessary to minimize the number of traversals of the corpus. Ideally, all computations need to be implemented in a single traversal.

Note that the `sum` and `size` methods are particular cases of a `foldLeft` operation. We could avoid a double traversal in the code of `average` if we implemented it as a single `foldLeft` call:

```
def average(s: List[Double]): Double = {
  val (sum, size) = s.foldLeft((0.0, 0)) { case ((sum, size), x) => (sum + x, size + 1) }
  sum / size
}
```

This maintains a combined accumulated state of type `(Double, Int)` for the `sum` and `size` aggregations.

We could always rewrite any number of aggregations as a single `foldLeft` that combines the aggregation codes into a more complicated updater function operating on a combined state value. How to avoid writing that code manually? The trick known as **fold fusion** allows us to merge any number of `fold`-like operations into a single operation. The code will automatically build and update the accumulated state correctly, traversing the sequence only once.

In this section, we will implement fold fusion in two ways: as an "applicative" fusion and as a "monadic" fusion. The contrast between these approaches will illustrate the difference between applicative and monadic functors.

The idea of fold fusion is to represent `fold`-like operations by values of a certain type, and to implement functions that combine and "run" such operations.

We begin by defining a type constructor that represents a `fold`-like operation as a value. Let us look at the type signature of `foldLeft`:

```
def foldLeft[Z, R](zs: Seq[Z])(init: R)(update: (R, Z) => R): R
```

We should be able to apply a given `fold`-like operation to many different sequences `zs`. So, we translate the type signature of `foldLeft` into a value by omitting the argument `zs`:

```
final case class Fold[Z, R](init: R, update: (R, Z) => R)
```

To run the `fold`-like operation represented by a value of type `Fold[Z, R]`, we implement a "runner":

```
def run[Z, R](zs: Seq[Z], fold: Fold[Z, R]): R = zs.foldLeft(fold.init)(fold.update)
```

Two `fold`-like operations can be combined only if they have the same type `Z`. If we combine an `op1: Fold[Z, R]` with an `op2: Fold[Z, S]`, we should obtain a new `fold`-like operation of type `Fold[Z, (R, S)]`. So, the operation of combining `op1` and `op2` is equivalent to a `zip` operation for the type constructor `Fold[Z, R]` with respect to the type parameter `R`, while the type `Z` is kept fixed:

```
def zipFold[Z, R, S](op1: Fold[Z, R], op2: Fold[Z, S]): Fold[Z, (R, S)] =
  Fold((op1.init, op2.init), (r, z) => (op1.update(r._1, z), op2.update(r._2, z)))
```

The types of `Fold` and `zipFold` are expressed in the short type notation as:

$$\begin{aligned}
&\text{Fold}^{Z,R} \stackrel{\text{def}}{=} R \times (R \times Z \to R) \quad , \\
&\text{zip}_{\text{Fold}} : \text{Fold}^{Z,R} \times \text{Fold}^{Z,S} \to \text{Fold}^{Z,R\times S} \quad , \qquad \text{or equivalently:} \\
&\text{zip}_{\text{Fold}} : R \times (R \times Z \to R) \times S \times (S \times Z \to S) \to (R \times S) \times (R \times S \times Z \to R \times S) \quad .
\end{aligned}$$

A fully parametric, information-preserving implementation of `zipFold` follows unambiguously from its type signature. So, let us use the `curryhoward` library[2] to generate the code automatically:

[1] Such as the "Common Crawl" corpus, see https://commoncrawl.org/

[2] See https://github.com/Chymyst/curryhoward

```
import io.chymyst.ch._       // Import some symbols from the ‘curryhoward‘ library.
def zipFold[Z, R, S](op1: Fold[Z, R], op2: Fold[Z, S]): Fold[Z, (R, S)] = implement
```

With these definitions, we can rewrite `average` as a single-traversal calculation:

```
val sum = Fold[Double, Double](0, _ + _)
val length = Fold[Double, Int](0, (n, _) => n + 1)
val sumLength: Fold[Double, (Double, Int)] = zipFold(sum, length)
val res: (Double, Int) = run(Seq(1.0, 2.0, 3.0), sumLength)

scala> val average = res._1 / res._2
average: Double = 2.0
```

In this way, we may combine any number of `fold`-like operations into a single traversal.

While this simple implementation of applicative fold fusion works, it is not easy to use. The result of "running" a combined `Fold` operation will be a tuple structure that must be transformed in just the right way (do we need `res._1 / res._2` or `res._2 / res._1`?) to obtain the final result. A better approach is to implement a syntax that combines some `Fold` operations and at the same time defines a final transformation to be applied to the results. We would like to write code that directly manipulates `fold`-like operations and produces the final result:

```
val sum = ...            // Define ‘sum‘ and ‘length‘ in a suitable way.
val length = ...
Seq(1.0, 2.0, 3.0).runFold(sum / length)    // Should return ‘2.0‘ here.
```

To enable this kind of code,[3] we need to add a final transformation to the `Fold` type:

```
final case class FoldOp[Z, R, A](init: R, update: (R, Z) => R, transform: R => A)
```

A value of type `FoldOp[Z, R, A]` represents a `fold`-like operation for sequences of type `Seq[Z]`. The operation accumulates an intermediate state of type `R` and computes a final result of type `A`.

The runner is implemented as an extension method on `Seq`:

```
implicit class FoldOpSyntax[Z](zs: Seq[Z]) {
  def runFold[R, A](op: FoldOp[Z, R, A]): A = op.transform(zs.foldLeft(op.init)(op.update))
}
```

How can we combine two `fold`-like operations of types $\text{FoldOp}^{Z,R,A}$ and $\text{FoldOp}^{Z,S,B}$? The combined operation must maintain intermediate states of types R and S. Also, we have two result values of types A and B. It seems that the combined operation needs an intermediate state of type $R \times S$ and a final result of type $A \times B$. So, let us implement a `zip` extension method by using those types:

```
implicit class FoldOpZip[Z, R, A](op: FoldOp[Z, R, A]) {
  def zip[S, B](other: FoldOp[Z, S, B]): FoldOp[Z, (R, S), (A, B)] = implement
  def map[B](f: A => B): FoldOp[Z, R, B] = implement
  def map2[S, B, C](other: FoldOp[Z, S, B])(f: (A, B) => C): FoldOp[Z, (R, S), C] = implement
} // The type signatures unambiguously determine the implementations.
```

The `map` and `map2` methods exist because `FoldOp[Z, R, A]` is covariant with respect to `A`.

We can now implement extension methods allowing us to do arithmetic on `fold`-like operations:

```
implicit class FoldOpMath[Z, R](op: FoldOp[Z, R, Double]) {
  def binaryOp[S](other: FoldOp[Z, S, Double])(f: (Double, Double) => Double): FoldOp[Z, (R, S),
     Double] = op.map2(other) { case (x, y) => f(x, y) }
  def +[S](other: FoldOp[Z, S, Double]): FoldOp[Z, (R, S), Double] = op.binaryOp(other)(_ + _)
  def /[S](other: FoldOp[Z, S, Double]): FoldOp[Z, (R, S), Double] = op.binaryOp(other)(_ / _)
} // May need to define more operations here.
```

After these definitions, the following code will work:

```
val sum = FoldOp[Double, Double, Double](0, (s, i) => s + i, identity)
val length = FoldOp[Double, Double, Double](0, (s, _) => s + 1, identity)
```

[3] A more fully featured library using this approach is `scala-fold`, see https://github.com/amarpotghan/scala-fold

```
scala> Seq(1.0, 2.0, 3.0).runFold(sum / length)
res0: Double = 2.0
```

We note that the type signatures of `zip` and `map2` are different from the usual ones: the accumulated state's type (`R`) is transformed into a pair type `(R, S)`. However, the programmer's code does not need to annotate those types because they are automatically inferred by the Scala compiler.

The standard `scanLeft` method is similar to `foldLeft` except it outputs all intermediate accumulated states (while `foldLeft` outputs only the last one). Can we merge several `scanLeft` operations into a single traversal? Since the type signature of `scanLeft` is the same as that of `foldLeft` except for the return type, the `FoldOp` data structure already stores the information needed to run `scanLeft`. The code for a corresponding runner (called `runScan`) is:

```
implicit class FoldOpScan[Z](zs: Seq[Z]) {
 def runScan[R,A](op: FoldOp[Z,R,A]): Seq[A] = zs.scanLeft(op.init)(op.update).map(op.transform).tail
}
```

As an example of using this functionality, consider the task of computing the running average of a sequence. A running average could be computed from the beginning of the sequence:

$$\text{ave}_k \stackrel{\text{def}}{=} \frac{1}{k}\sum_{i=0}^{k-1} s_i \quad .$$

The code `val average = sum / length` implements a `fold`-like operation computing this average.

Another task is to compute the average over a sliding window of a fixed size n:

$$\text{ave}_{n,k} \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=k-n}^{k-1} s_i \quad .$$

To implement this, we first create a sliding window as a `fold`-like operation and then do averaging:

```
def window[A](n: Int): FoldOp[A, IndexedSeq[A], IndexedSeq[A]] = FoldOp(init = Vector(),
  update = { (window, x) => (if (window.size < n) window else window.drop(1)) :+ x }, identity)

def window_average(n: Int) = window[Double](n).map(_.sum / n)
```

To test that the averaging code works as expected, we run some fold and scan operations:

```
scala> (0 to 10).map(_.toDouble).runFold(average)
res0: Double = 5.0

scala> (0 to 10).map(_.toDouble).runScan(average)
res1: Vector[Double] = Vector(0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0)

scala> (0 to 10).map(_.toDouble).runScan(window_average(3))
res2: Vector[Double] = Vector(0.0, 0.3333333333333333, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)
```

11.2.6 Single-traversal `fold` operations. II. Monadic "fold fusion"

The applicative fold fusion as defined in the previous section works by combining the results of independent `fold`-like operations. For instance, the definition `average = sum / length` uses `sum` and `length` as independent `fold`-like operations (the results computed by `sum` do not depend on the results of `length`). The independence of effects is a defining feature of applicative composition. In contrast, monadic composition supports effects that may depend on previously computed values. To illustrate this contrast, let us implement the monadic version of fold fusion.

The type constructor `FoldOp[Z, R, A]` can be a monad only if it is covariant. So, a `flatMap` method must operate on the type parameter `A`. We begin writing the type signature as:

```
def flatMap[Z, R, A, S, B](op: FoldOp[Z, R, A])(f: A => FoldOp[Z, S, B]): FoldOp[Z, ???]
```

The first `fold`-like operation (`op`) will compute a value of type `A` at each step. We will then apply the given function `f` to that value and obtain a new `fold`-like operation (of type `FoldOp[Z, S, B]`) that also needs to be evaluated. Since `flatMap` will need to run two `fold`-like operations in one traversal, it must maintain a combined accumulated state of type `(R, S)`. So, the type signature of `flatMap` needs to accommodate the change of that type (just as the type signature of `map2` did in the previous section). For convenience, we define `flatMap` as an extension method:

```
implicit class FoldFlatMap[Z, R, A](op: FoldOp[Z, R, A]) {
  def flatMap[S, B](f: A => FoldOp[Z, S, B]): FoldOp[Z, (R, S), B] = ???
}
```

The result of `op1.flatMap(x => op2(x))` must be a `fold`-like operation that combines `op1` and `op2` and supports arbitrary dependency in `op2(x)` on the intermediate result (`x: A`), which is obtained by running `op1`. To achieve that, we implement `flatMap` via the following code:

```
1  implicit class FoldFlatMap[Z, R, A](op: FoldOp[Z, R, A]) {
2     def flatMap[S, B](f: A => FoldOp[Z, S, B]): FoldOp[Z, (R, S), B] = {
3        // To create a new ‘FoldOp()‘ value, we need ‘init‘, ‘update‘, and ‘transform‘.
4        val init: (R, S) = (op.init, f(op.transform(op.init)).init) // Use ‘init‘ from both operations.
5        val update: ((R, S), Z) => (R, S) = { case ((r, s), z) =>   // Run both ‘update‘ functions:
6            val newR = op.update(r, z)                              // First ‘update‘ function.
7            val newOp: FoldOp[Z, S, B] = f(op.transform(newR))      // We may use ‘newR‘ or ‘r‘ here!
8            val newS = newOp.update(s, z)                           // Second ‘update‘ function.
9            (newR, newS)
10       }
11       val transform: ((R, S)) => B = { case (r, s) =>
12           val newOp: FoldOp[Z, S, B] = f(op.transform(r))
13           newOp.transform(s)
14       }
15       FoldOp(init, update, transform)
16   }
17 }
```

In line 7 of the code above, we have a choice of applying `op.transform(...)` to the old accumulated value (`r`) or to the updated value (`newR`). It appears to be better to use the updated value (`newR`).

To motivate this choice, consider a running average operation applied twice:

```
scala> (0 to 10).toList.map(_.toDouble).runScan(average).runScan(average)
res3: List[Double] = List(1.0, 1.25, 1.5, 1.75, 2.0, 2.25, 2.5, 2.75, 3.0, 3.25)
```

Using `flatMap`, we can express the same computation as a single `fold`-like operation:

```
def add(x: Double) = FoldOp[Double, Double, Double](0, (a, _) => a + x, identity)

scala> (0 to 10).toList.map(_.toDouble).runScan(average.flatMap(x => add(x) / length))
res4: List[Double] = List(1.0, 1.25, 1.5, 1.75, 2.0, 2.25, 2.5, 2.75, 3.0, 3.25)
```

The code can be rewritten as a functor block, making it more visually clear:

```
val average2a = for {
      x <- average
      accum <- add(x)
      n <- length
} yield accum / n

scala> (0 to 10).toList.map(_.toDouble).runScan(average2a)
res5: List[Double] = List(1.0, 1.25, 1.5, 1.75, 2.0, 2.25, 2.5, 2.75, 3.0, 3.25)
```

We get the same result as when applying `runScan(average)` twice. The result would not be obtained if we used the old value (`r`) in line 7 in the definition of `flatMap` above.

The `flatMap` function implements a monadic fusion of `fold`-like operations.[4] It is interesting to note

[4] The "`origami`" library (see https://github.com/atnos-org/origami) implements a more general functionality: values computed by its `fold`-like operations are themselves of monadic type. This is similar to using `FoldOp[Z, R, M[A]]` where `M` is a monad and the updater function has type `(R, Z) => M[R]`.

that the monadic fold fusion is compatible with the applicative fold fusion. For instance, one could define `average` equivalently as:

```
val average = for {
  acc <- sum
  n <- length
} yield acc / n
```

This equivalence comes from the fact that the case class `FoldOp[Z, R, A]` depends on `A` only through the `transform` value, which is of type `R => A`. So, the monadic behavior of `FoldOp[Z, R, A]` with respect to `A` is similar to that of the `Reader` monad of type `R => A`. Similarly to the `Reader` monad, the effects in the `FoldOp` monad are independent.

11.2.7 Parsing with applicative and monadic combinators

Applicative functors are used for implementing parsers via the "parser combinator" technique. In this technique, the programmer builds a large parser out of smaller ones by using different functions ("combinators"). We will now look at both applicative and monadic parser combinators.

The parsing techniques will be illustrated on some toy "languages". The first language encodes square roots of positive integers in an XML-like format, as, for example, in the strings `<sqrt>121</sqrt>`, `<sqrt><sqrt>10000</sqrt></sqrt>`, and `123`. Examples of *invalid* strings are `<sqrt>100</sqrt></sqrt>` (junk at end of text), `<sqrt>121<sqrt>` (tag not closed), `</sqrt>10</sqrt>` (tag not opened), and `1.0` (not an integer). The parser should output an integer result or an error message.

To simplify the code, we define a parser as a function taking an input string and returning either a value of some type `A` (the value "parsed out" of the input) or some error information. The parser also returns the unused part of the input string. So, we define the type constructor `P[A]` by:

```
final case class P[A](run: String => (Either[Err, A], String)) // A parser returns a result or fails.
type Err = List[String]           // A list of error messages.
```

In order to parse the toy language, we need to be able to parse integers, opening tags, closing tags, and detect whether the end of input occurs at a point where all tags are balanced. The parser combinator technique begins by defining the simplest necessary parsers. We will need a parser for integers and a parser that expects the input to start with a given, fixed string:

```
val intP: P[Int] = P {
  val numRegex = "^([0-9]+)(.*)$".r
  s => s match {
    case numRegex(num, rest)   => (Right(num.toInt), rest)
    case s                     => (Left(List("no number")), s)
  }
}
def constP(prefix: String, error: String = "no prefix"): P[String] = P { s =>
  if (s startsWith prefix) (Right(prefix), s.stripPrefix(prefix)) else (Left(List(error)), s)
}
```

Let us test that these parsers work as expected:

```
scala> intP.run("123xyz")
res0: (Either[Err, Int], String) = (Right(123), "xyz")

scala> constP("<sqrt>").run("<sqrt>1</sqrt>")
res1: (Either[Err, String], String) = (Right("<sqrt>"), "1</sqrt>")
```

The next step is to define combinators that produce larger parsers from smaller ones. We will define four different combinators corresponding to a functor's `map`, an applicative `zip`, a monoid's `combine`, and a monad's `flatMap` methods. For convenience, we implement the combinators as extension methods on the type constructor `P`. Start with `zip` and use the code from Section 11.1.2:

```
implicit class ParserZipOps[A](parserA: P[A]) {
   def zip[B](parserB: P[B]): P[(A, B)] = P { s =>
      val (resultA, rest) = parserA.run(s)
      val (resultB, restB) = parserB.run(rest)
      val result = (resultA, resultB) match { // Use the ‘zip‘ operation for Either[List[String], A].
        case (Left(x), Left(y))      => Left(x ++ y)
```

```
        case (Left(x), Right(_))    => Left(x)
        case (Right(_), Left(y))    => Left(y)
        case (Right(x), Right(y))   => Right((x, y))
      }
      (result, restB)
    }
}
```

To test this, let us define a parser `p1` for strings of the form `<sqrt>123</sqrt>`:

```
val openTag = constP("<sqrt>", "tag must be open")
val closeTag = constP("</sqrt>", "tag must be closed")
val p1 = openTag zip intP zip closeTag

scala> p1.run("<sqrt>123</sqrt>")
res4: (Either[Err, ((String, Int), String)], String) = (Right((("<sqrt>", 123), "</sqrt>")), "")

scala> p1.run("<sqrt></sqrt>")
res5: (Either[Err, ((String, Int), String)], String) = (Left(List("no number")), "")

scala> p1.run("abc")
res6: (Either[Err, ((String, Int), String)], String) = (Left(List("tag must be open", "no number",
     "tag must be closed")), "abc")
```

Because the parser `p1` is defined via the applicative `zip` method, the code can report several errors at once. With suitable definitions, parsers combined via `zip` could skip some invalid text and attempt to parse further data, possibly reporting more errors to the user.

It is inconvenient that a `zip`-combined parser returns a deeply nested tuple structure as shown above; we are often interested in reading only one value inside that structure. We would like `p1.run("<sqrt>123</sqrt>")` to return just `Right(123, "")`. To achieve this, we define the helper methods `zipLeft` and `zipRight`, whose names indicate the part of the tuple that is being returned:

```
implicit class ParserMoreZipOps[A](parserA: P[A]) {
   def map[B](f: A => B): P[B] = P { s =>
      val (result, rest) = parserA.run(s)
      (result.map(f), rest)
   } //
   def zipLeft [B](parserB: P[B]): P[A] = (parserA zip parserB).map(_._1)
   def zipRight[B](parserB: P[B]): P[B] = (parserA zip parserB).map(_._2)
}

scala> (openTag zipRight intP zipLeft closeTag).run("<sqrt>123</sqrt>")
res7: (Either[Err, Int], String) = (Right(123), "")
```

The next parser combinator corresponds to a monoid's `combine`, but we will call it "`or`" for convenience. The result of `(p or q)` is a parser that succeeds if either `p` or `q` succeed and returns the corresponding result (the parser `p` is tried first):

```
implicit class ParserCombineOps[A](p: P[A]) {
   def or(q: => P[A]): P[A] = P { s =>       // Important: the argument ‘q‘ must be lazy.
   val (result, rest) = p.run(s)
   result match {                            // If ‘p‘ failed to parse the string ‘s‘,
     case Left(err)  => q.run(s)             // ignore the error from ‘p‘ and run ‘q‘.
     case Right(x)   => (Right(x), rest)     // Otherwise, use the result of running ‘p‘.
   }
}
```

We implement `q` as a lazy argument because we should not evaluate `q` when the parser `p` succeeds. This will help us write recursively defined parsers, as we will see next.

Using these combinators, we write a first attempt at parsing the toy language. Since the language allows us to have arbitrarily deep nesting of the tags `<sqrt>...</sqrt>`, we need to define the parser as a recursive function (called `p2`):

```
def p2: P[Int] = intP or (openTag zipRight p2 zipLeft closeTag).map { x => math.sqrt(x).toInt }
```

```
scala> p2.run("121")._1.right.get
res8: Int = 121

scala> p2.run("<sqrt>121</sqrt>")._1.right.get
res9: Int = 11
```

The recursion stops only because the operation "`or`" does not evaluate its second parser when the first parser succeeds. Otherwise the code would go into an infinite loop.

We are able to parse the toy language using only applicative combinators. Let us now consider a more complicated language that *requires* monadic combinators to be parsed correctly. The language represents equations between integers, such as `1=1` or `123=123`, but does not admit `1=2`. To parse this language, we need to get the first integer, skip the equals sign, and then check that the second integer is equal to the first one. So, we need to use a parser that depends on the result of parsing a previous substring. We would like to implement the parser using this straightforward code:

```
val emptyP: P[Unit] = P { s => if (s.isEmpty) (Right(()), s) else (Left(List("junk at end")), s) }

val p3: P[Int] = for {
  x <- intP                // Expect an integer.
  _ <- constP("=")         // Expect an equals sign.
  _ <- constP(x.toString)  // Expect a string corresponding to the previously parsed integer ‘x‘.
  _ <- emptyP              // Expect end of input.
} yield x
```

To make this code work, we define `flatMap` as an extension method on `P`:

```
implicit class ParserMonadOps[A](parserA: P[A]) {
   def flatMap[B](f: A => P[B]): P[B] = P { s =>
      val (result, rest) = parserA.run(s)
      result match {
        case Left(err)   => (Left(err), rest)
        case Right(x)    => f(x).run(rest)
      }
   }
}
```

The result is a parser `p3` that that stops at the first error:

```
scala> p3.run("123=123")
res10: (Either[Err, Int], String) = (Right(123), "")

scala> p3.run("1=2")
res11: (Either[Err, Int], String) = (Left(List("integer 1 not found")), "2")
```

The error reporting in `p3` can be improved if we use both monadic and applicative combinators in the same code. Let us implement and use a special parser (called "`expect`") that will fail with an informative message when a parsed integer is not equal to a given value:

```
val success: P[Unit] = P { s => (Right(()), s) }  // Always-succeeding parser that consumes no input.
def failure(message: String): P[Unit] = P { s => (Left(List(message)), s) } // Always-failing parser.

def expect(n: Int): P[Unit] = intP.flatMap { x =>
  if (x == n) success else failure(s"got $x but expected $n")
}

val p4: P[Int] = for {
   x <- intP zipLeft constP("=")
   _ <- expect(x) zip emptyP
} yield x

scala> p4.run("1=2mumbo-jumbo gobbledy-gook")
res12: (Either[Err, Int], String) = (Left(List("got 2 but expected 1", "junk at end")), "mumbo-jumbo
     gobbledy-gook")
```

Several parser combinator libraries[5] are designed after these principles and provide applicative, monadic, filterable, monoidal, and other functionality for building up large parsers from parts.

11.2.8 Functor block syntax for applicative functors

It is often desirable to combine applicative and monadic computations in a single functor block. This can be done by using `zip`, `zipLeft`, and `zipRight` within source lines, as shown in the previous section.

```
for {
  x       <-   p
  (y, z)  <-   q(x) zip r(x)
  t       <-   s(x, y, z)
  ...
```

In the code shown at left, `q` and `r` cannot depend on each other's returned values (`y` and `z`). However, `q` and `r` may depend on any of the previously computed values (such as `x`). Later computations (such as `s`) may also depend on the previous values (`x`, `y`, `z`).

There have been several proposals[6] to add a special block syntax for applicative functors in Scala. So far, these proposals have *not* been accepted as a part of the Scala language. Some experimental projects implement a `for`/`yield` syntax for applicative functors.[7]

11.2.9 Exercises

Exercise 11.2.9.1 Implement `map2` (or `xmap2` if appropriate) for the following type constructors F^A:

(a) $F^A \stackrel{\text{def}}{=} \mathbb{1} + A + A \times A$.

(b) $F^A \stackrel{\text{def}}{=} E \to A \times A$.

(c) $F^A \stackrel{\text{def}}{=} Z \times A \to A$.

(d) $F^A \stackrel{\text{def}}{=} A \to A \times Z$ where Z is a `Monoid`.

Exercise 11.2.9.2 Implement a `zip` method for a ternary tree `T3[A]` with extra data on branches:

```
sealed trait T3[A]
case class Leaf[A](a: A) extends T3[A]
case class Branch[A](label: A, left: T3[A], center: T3[A], right: T3[A]) extends T3[A]

def zip[A, B](ta: T3[A], tb: T3[B]): T3[(A, B)] = ???
```

Aim to preserve information and avoid changing the tree structures unnecessarily.

Exercise 11.2.9.3 Write a `zip` function that defines an instance of the `Semigroup` type class for a tuple `(A, B)` where the types `A` and `B` are assumed to already have a `Semigroup` instance. Test on an example with `Semigroup` instances for `A = Int` and `B = String`.

Exercise 11.2.9.4 Define a `Monoid` instance for the type F^S where F is an applicative functor that has `map2` and `pure`, while S is itself a monoid type.

Exercise 11.2.9.5 Define a "regexp extractor" as a type constructor R^A describing extraction of various data from strings; the extracted data has type `Option[A]`. Implement `zip` and `map2` for R^A.

Exercise 11.2.9.6 Use fold fusion (Section 11.2.5) to implement a `FoldOp` that computes the standard deviation of a sequence of type `Seq[Double]` in one traversal.

Exercise 11.2.9.7 Use parser combinators defined in Section 11.2.7 (alternatively, use a parser combinator library of your choice) to implement a parser for a toy language that contains a single integer in a sequence of arbitrary nested tags, such as `<a><b>123</b></a>`. Tag names are restricted to lowercase Latin letters. The parser should return that integer or fail if tags are incorrectly nested.

Exercise 11.2.9.8 **(a)** Use parser combinators to parse a toy arithmetic language containing decimal digits and "`+`" symbols, for example `1+321+20`. The parser should return the resulting integer.

(b)* Extend the parser to support parentheses and "`-`" symbols, for example `10-(2+3)`.

[5] See, for example, the "`fastparse`" library: https://com-lihaoyi.github.io/fastparse

[6] See, for example, https://contributors.scala-lang.org/t/4474

[7] See, for example, https://github.com/bkirwi/applicative-syntax where a special macro is used, which may not work with Scala 3. The blog post https://lptk.github.io/programming/2018/03/02/a-dual-to-iterator.html proposes a `for`/`yield` syntax for applicative functors and implements single-traversal applicative folds without using macros.

11.3 Laws and structure of applicative functors

11.3.1 Equivalence of `map2`, `zip`, and `ap`

We have seen in Section 11.1.2 that the implementations of `map2` and `zip` for `Either[E, A]` are quite similar. This similarity holds in the general case: `map2` and `zip` are equivalent assuming a suitable naturality law of `map2`. (Naturality laws will always hold if the code is fully parametric.)

Statement 11.3.1.1 (equivalence of `map2` and `zip`) For any functor L for which `map2` or `zip` can be implemented, the type of functions `zip` (type signature $\text{zip} : L^A \times L^B \to L^{A\times B}$) is equivalent to the type of functions `map2` (type signature $L^A \times L^B \to (A \times B \to C) \to L^C$), assuming that the functions `map2` satisfy the naturality law with respect to the type parameter C.

Proof We need to show equivalence in two directions (from `map2` to `zip` and back).

(1) Given any function $\text{map}_2 : L^A \times L^B \to (A \times B \to C) \to L^C$ satisfying the naturality law:

$$\text{map}_2\,(p^{:L^A} \times q^{:L^B})(f^{:A\times B\to C}) \triangleright (g^{:C\to D})^{\uparrow L} = \text{map}_2\,(p \times q)(f \,⨾\, g) \quad ,$$

we first define a `zip` method:

$$\text{zip} : L^A \times L^B \to L^{A\times B} \quad , \qquad \text{zip}\,(p^{:L^A} \times q^{:L^B}) \stackrel{\text{def}}{=} \text{map}_2\,(p \times q)(\text{id}^{:A\times B\to A\times B}) \quad ,$$

and then define a new `map2'` function through that `zip` method:

$$\text{map}_2'\,(p^{:L^A} \times q^{:L^B})(f^{:A\times B\to C}) \stackrel{\text{def}}{=} (p \times q) \triangleright \text{zip} \triangleright f^{\uparrow L} \quad .$$

Then we need to show that $\text{map}_2' = \text{map}_2$. We apply map_2' to arbitrary arguments and write:

$$\text{map}_2'\,(p \times q)(f) = (p \times q) \triangleright \text{zip} \triangleright f^{\uparrow L} = \text{map}_2\,(p \times q)(\text{id}) \triangleright \underline{f^{\uparrow L}}$$

naturality law of map_2 : $$= \text{map}_2\,(p \times q)(\text{id} \,⨾\, f) = \text{map}_2\,(p \times q)(f) \quad .$$

(2) Given any function $\text{zip} : L^A \times L^B \to L^{A\times B}$, we first define a `map2` method:

$$\text{map}_2\,(p^{:L^A} \times q^{:L^B})(f^{:A\times B\to C}) \stackrel{\text{def}}{=} (p \times q) \triangleright \text{zip} \triangleright f^{\uparrow L} \quad ,$$

and then define a new `zip'` function through that `map2` method:

$$\text{zip}'\,(p^{:L^A} \times q^{:L^B}) \stackrel{\text{def}}{=} \text{map}_2\,(p \times q)(\text{id}^{:A\times B\to A\times B}) \quad .$$

Then we need to show that $\text{zip}' = \text{zip}$. We apply zip' to arbitrary arguments p, q and write:

$$\text{zip}'\,(p^{:L^A} \times q^{:L^B}) = \text{map}_2\,(p \times q)(\text{id}^{:A\times B\to A\times B}) = (p \times q) \triangleright \text{zip} \triangleright \text{id}^{\uparrow L} = (p \times q) \triangleright \text{zip} = \text{zip}\,(p \times q) \quad .$$

It remains to show that the `map2` method will satisfy the naturality law if defined via `zip`:

$$\underline{\text{map}_2\,(p \times q)(f)} \triangleright g^{\uparrow L} = (p \times q) \triangleright \text{zip} \triangleright f^{\uparrow L} \triangleright g^{\uparrow L} = (p \times q) \triangleright \text{zip} \triangleright (f \,⨾\, g)^{\uparrow L}$$

definition of map_2 via zip : $$= \text{map}_2\,(p \times q)(f \,⨾\, g) \quad .$$

□

The equivalence of `map2` and `zip` follows a pattern similar to one shown in Section 10.3.2: a function with one type parameter (a natural transformation) is equivalent to a function with two type parameters if a naturality law holds with respect to one of those type parameters. Here `zip` is playing the role of the natural transformation, and `map2` is a function obeying a naturality law. The proof of Statement 11.3.1.1 is similar to the proof of Statement 10.3.2.1.

However, `map2` is not a "lifting" in the sense of Section 10.3.2. The method "`ap`" better fits the role of a lifting since its type signature ($\text{ap} : L^{A\to B} \to L^A \to L^B$) transforms functions wrapped under L (i.e., functions of type $L^{A\to B}$) to functions of type $L^A \to L^B$. So, let us prove that `ap` is equivalent to `map2`. Since the type signature of `ap` is curried, it is more convenient to use the curried version (`fmap2`) instead of `map2`. Let us write the relationship between `ap` and `fmap2` in the code notation:

$$\text{fmap}_2\,(f^{:A\to B\to C}) = f^{\uparrow L} \,{}^{\circ}\!\!{}_{;}\, \text{ap}^{B,C} \quad ,$$
$$\text{ap}^{A,B} = \text{fmap}_2\,(\text{id}^{:(A\to B)\to A\to B}) \quad .$$

$$\begin{tikzcd} & F^{B\to C} \arrow[dr, "\text{ap}^{B,C}"] & \\ F^A \arrow[ur, "f^{\uparrow L}"] \arrow[rr, "\text{fmap}_2\,(f^{:A\to B\to C})"'] & & (F^B \to F^C) \end{tikzcd}$$

We are now ready to prove the equivalence of `ap` and `fmap2`:

Statement 11.3.1.2 (equivalence of `ap` and `fmap2`) For any functor L for which `fmap2` or `ap` can be implemented, the type of functions `ap` (type signature $\text{ap} : L^{A\to B} \to L^A \to L^B$) is equivalent to the type of functions `fmap2` (type signature $(A \to B \to C) \to L^A \to L^B \to L^C$), assuming that the functions `fmap2` satisfy the naturality law with respect to the type parameter C.

Proof We need to show equivalence in two directions (from `fmap2` to `ap` and back).

(1) Given any function $\text{fmap}_2 : (A \to B \to C) \to L^A \to L^B \to L^C$ satisfying the naturality law,

$$\text{fmap}_2\,(g^{:X\to A} \,{}^{\circ}\!\!{}_{;}\, f^{:A\to B\to C})(p^{:L^X}) = \text{fmap}_2\,(f)(p \triangleright g^{\uparrow L}) \quad ,$$

we first define an `ap` method:

$$\text{ap} : L^{A\to B} \to L^A \to L^B \quad , \qquad \text{ap}\,(r^{:L^{A\to B}}) \stackrel{\text{def}}{=} \text{fmap}_2\,(\text{id}^{:(A\to B)\to A\to B})(r) \quad ,$$

and then define a new `fmap2'` function through that `ap` method:

$$\text{fmap}'_2\,(f^{:A\to B\to C})(p^{:L^A}) \stackrel{\text{def}}{=} p \triangleright f^{\uparrow L} \triangleright \text{ap} \quad .$$

Then we need to show that $\text{fmap}'_2 = \text{fmap}_2$. We apply fmap'_2 to arbitrary arguments and write:

$$\text{fmap}'_2\,(f^{:A\to B\to C})(p^{:L^A}) = p \triangleright f^{\uparrow L} \triangleright \text{ap} = \text{fmap}_2\,(\text{id})(p \triangleright f^{\uparrow L})$$
$$\text{naturality law of fmap}_2 : \quad = \text{fmap}_2\,(f \,{}^{\circ}\!\!{}_{;}\, \text{id})(p) = \text{fmap}_2\,(f)(p) \quad .$$

(2) Given any function $\text{ap} : L^{A\to B} \to L^A \to L^A$, we first define `fmap2`:

$$\text{fmap}_2\,(f^{:A\to B\to C}) \stackrel{\text{def}}{=} f^{\uparrow L} \,{}^{\circ}\!\!{}_{;}\, \text{ap} \quad ,$$

and then define a new `ap'` function through that `fmap2` method:

$$\text{ap}' \stackrel{\text{def}}{=} \text{fmap}_2\,(\text{id}) \quad .$$

Then we need to show that $\text{ap}' = \text{ap}$. We write:

$$\text{ap}' = \text{fmap}_2\,(\text{id}) = \text{id}^{\uparrow L} \,{}^{\circ}\!\!{}_{;}\, \text{ap} = \text{ap} \quad .$$

It remains to show that the `fmap2` method will satisfy the naturality law if defined via `ap`:

$$\text{fmap}_2\,(g \,{}^{\circ}\!\!{}_{;}\, f)(p) = p \triangleright (g \,{}^{\circ}\!\!{}_{;}\, f)^{\uparrow L} \,{}^{\circ}\!\!{}_{;}\, \text{ap} = p \triangleright g^{\uparrow L} \triangleright \underline{f^{\uparrow L} \,{}^{\circ}\!\!{}_{;}\, \text{ap}}$$
$$\text{definition of fmap}_2 \text{ via ap} : \quad = (p \triangleright g^{\uparrow L}) \triangleright \text{fmap}_2\,(f) = \text{fmap}_2\,(f)(p \triangleright g^{\uparrow L}) \quad .$$

□

Since `zip` is equivalent to `map2` and `map2` is equivalent to `ap`, it follows (assuming suitable naturality laws) that `zip` is equivalent to `ap`. In Scala, the relationship between `zip` and `ap` is coded like this:

```
def zip[A, B](la: L[A], lb: L[B]): L[(A, B)] =    // Assuming that ‘ap‘ is already defined.
  ap[B, (A, B)](la.map {a => (b: B) => (a, b)} )(lb)

def ap[A, B](lf: L[A => B])(la: L[A]): L[B] =     // Assuming that ‘zip‘ is already defined.
  zip(lf, la).map { case (f, a) => f(a) }
```

Statement 11.3.1.3 The functions `zip` and `ap` are equivalent assuming the following naturality laws:

$$(f^{\uparrow L} \boxtimes \text{id}) ⨾ \text{zip} = \text{zip} ⨾ (f \boxtimes \text{id})^{\uparrow L} \quad , \quad \big(\text{ap}\,(r^{:L^{A\to B}})(p^{:L^A})\big) \triangleright (f^{:B\to C})^{\uparrow L} = \text{ap}\big(r \triangleright (x^{:A\to B} \to x ⨾ f)^{\uparrow L}\big)(p) \quad .$$

Proof We need to show equivalence in two directions (from `zip` to `ap` and back). To write the relationships between `zip` and `ap` more easily, let us denote by `pair` and `eval` the following two functions:

$$\text{pair}^{:A\to B\to A\times B} \stackrel{\text{def}}{=} a \to b \to a \times b \quad , \qquad \text{eval}^{:(A\to B)\times A\to B} \stackrel{\text{def}}{=} g^{:A\to B} \times a^{:A} \to g(a) \quad .$$

The code of these fully parametric functions follows from their types. Then we can write:

$$\text{zip}\,(p^{:L^A} \times q^{:L^B}) = \text{ap}\,(p \triangleright \text{pair}^{\uparrow L})(q) \quad , \qquad \text{ap}\,(r^{:L^{A\to B}})(p^{:L^A}) = (r \times p) \triangleright \text{zip} \triangleright \text{eval}^{\uparrow L} \quad .$$

The functions `pair` and `eval` satisfy the following property that we will use in the proof:

$$(\text{pair}^{:A\to B\to A\times B} \boxtimes \text{id}^{:B\to B}) ⨾ \text{eval}^{:(B\to A\times B)\times B\to A\times B} = \text{id}^{:A\times B\to A\times B} \quad . \tag{11.1}$$

To prove this property, apply both sides to arbitrary $a^{:A}$ and $b^{:B}$:

$$(a^{:A} \times b^{:B}) \triangleright (\text{pair} \boxtimes \text{id}) ⨾ \text{eval} = \big((z^{:B} \to a \times z) \times b\big) \triangleright \text{eval} = (z^{:B} \to a \times z)(b) = a \times b \quad .$$

(1) Given any function $\text{zip} : L^A \times L^B \to L^{A\times B}$ satisfying the naturality law shown above, we define an `ap` method by $\text{ap}\,(r)(p) \stackrel{\text{def}}{=} (r \times p) \triangleright \text{zip} \triangleright \text{eval}^{\uparrow L}$ and then define a new `zip'` function through that `ap`:

$$\text{zip}'\,(p^{:L^A} \times q^{:L^B}) \stackrel{\text{def}}{=} \text{ap}\,(p \triangleright \text{pair}^{\uparrow L})(q) \quad .$$

Then we need to show that $\text{zip}' = \text{zip}$. We apply zip' to arbitrary arguments and write:

$$\begin{aligned}
&\text{expect to equal zip}\,(p \times q): && \text{zip}'\,(p^{:L^A} \times q^{:L^B}) = \text{ap}\,(p \triangleright \text{pair}^{\uparrow L})(q) \\
&\text{definition of ap}: && = \big((p \triangleright \text{pair}^{\uparrow L}) \times q\big) \triangleright \text{zip} ⨾ \text{eval}^{\uparrow L} = (p \times q) \triangleright \underline{(\text{pair}^{\uparrow L} \times \text{id}) ⨾ \text{zip}} ⨾ \text{eval}^{\uparrow L} \\
&\text{naturality law of zip}: && = (p \times q) \triangleright \text{zip} ⨾ \underline{(\text{pair} \boxtimes \text{id})^{\uparrow L} ⨾ \text{eval}^{\uparrow L}} \\
&\text{use Eq. (11.1)}: && = (p \times q) \triangleright \text{zip} ⨾ \text{id}^{\uparrow L} = (p \times q) \triangleright \text{zip} \quad .
\end{aligned}$$

It remains to show that the `ap` method defined via `zip` will satisfy its required naturality law:

$$\begin{aligned}
&\text{left-hand side}: && \big(\text{ap}\,(r)(p)\big) \triangleright f^{\uparrow L} = (r \times p) \triangleright \text{zip} ⨾ \text{eval}^{\uparrow L} ⨾ f^{\uparrow L} \quad , \\
&\text{right-hand side}: && \text{ap}\big(r \triangleright (x \to x ⨾ f)^{\uparrow L}\big)(p) = \big((r \triangleright (x \to x ⨾ f)^{\uparrow L}) \times p\big) \triangleright \text{zip} ⨾ \text{eval}^{\uparrow L} \\
&\text{naturality law of zip}: && = (r \times p) \triangleright \text{zip} ⨾ \big((x \to x ⨾ f) \boxtimes \text{id}\big)^{\uparrow L} ⨾ \text{eval}^{\uparrow L} \quad .
\end{aligned}$$

The difference between the two sides of the law will disappear if we show that

$$\text{eval} ⨾ f \stackrel{?}{=} \big((x \to x ⨾ f) \boxtimes \text{id}\big) ⨾ \text{eval} \quad .$$

Apply both sides to an arbitrary pair $g \times a$ of type $(A \to B) \times A$ and obtain equal results:

$$\begin{aligned}
&\text{left-hand side}: && (g^{:A\to B} \times a^{:A}) \triangleright \text{eval} ⨾ f = (g \times a) \triangleright \text{eval} \triangleright f = g(a) \triangleright f = a \triangleright g \triangleright f \quad , \\
&\text{right-hand side}: && (g \times a) \triangleright \big((x \to x ⨾ f) \boxtimes \text{id}\big) ⨾ \text{eval} = (g \times a) \triangleright \big((x \to x ⨾ f) \boxtimes \text{id}\big) \triangleright \text{eval} \\
& && = \big((g ⨾ f) \times a\big) \triangleright \text{eval} = a \triangleright g ⨾ f = a \triangleright g ⨾ f \quad .
\end{aligned}$$

(2) Given any function $\text{ap} : L^{A\to B} \to L^A \to L^A$, we first define `zip`:

$$\text{zip}\,(p^{:L^A} \times q^{:L^B}) \stackrel{\text{def}}{=} \text{ap}\big(p \triangleright \text{pair}^{\uparrow L}\big)(q) \quad ,$$

and then define a new `ap'` function through that `zip` method:

$$\text{ap}'\,(r^{:L^{A\to B}})(p^{:L^A}) \stackrel{\text{def}}{=} (r\times p)\triangleright\text{zip}\triangleright\text{eval}^{\uparrow L}\quad.$$

Then we need to show that ap′ = ap. We write:

$$\begin{aligned}
&\text{ap}'\,(r)(p) = (r\times p)\triangleright\text{zip}\triangleright\text{eval}^{\uparrow L} = \big(\text{ap}\,(r\triangleright\text{pair}^{\uparrow L})(p)\big)\triangleright\text{eval}^{\uparrow L}\quad.\\
\text{naturality law of ap}:\quad &= \text{ap}\big(r\triangleright\text{pair}^{\uparrow L}\triangleright(x^{:A\to(A\to B)\times A}\to x\,⨾\,\text{eval})^{\uparrow L}\big)(p)\\
\text{composition under }{}^{\uparrow L}:\quad &= \text{ap}\big(r\triangleright(\text{pair}\,⨾\,(x\to x\,⨾\,\text{eval}))^{\uparrow L}\big)(p)\quad.
\end{aligned}$$

The remaining difference between ap′$(r)(p)$ and ap $(r)(p)$ will disappear if we show that

$$\text{pair}^{:(A\to B)\to A\to(A\to B)\times A}\,⨾\,(x^{:A\to(A\to B)\times A}\to x\,⨾\,\text{eval})\stackrel{?}{=}\text{id}\quad.$$

To see this, apply both sides to an arbitrary value f of type $A\to B$:

$$\begin{aligned}
\text{expect to equal }f:\quad & f^{:A\to B}\triangleright\text{pair}^{:(A\to B)\to A\to(A\to B)\times A}\,⨾\,(x^{:A\to(A\to B)\times A}\to x\,⨾\,\text{eval})\\
&= (a^{:A}\to f\times a)\,⨾\,(x\to x\,⨾\,\text{eval}) = a^{:A}\to(f\times a)\triangleright\text{eval} = a^{:A}\to f(a) = f\quad.
\end{aligned}$$

Finally, we show that the `zip` method will satisfy its naturality law if defined via `ap`:

$$\begin{aligned}
\text{left-hand side}:\quad & (p^{:L^A}\times q^{:L^B})\triangleright(f^{\uparrow L}\boxtimes\text{id})\,⨾\,\text{zip} = \text{zip}\big((p\triangleright f^{\uparrow L})\times q\big)\\
\text{express zip via ap}:\quad & = \text{ap}\big(p\triangleright f^{\uparrow L}\triangleright\text{pair}^{\uparrow L}\big)(q) = \text{ap}\big(p\triangleright(f\,⨾\,\text{pair})^{\uparrow L}\big)(q)\quad,\\
\text{right-hand side}:\quad & (p^{:L^A}\times q^{:L^B})\triangleright\text{zip}\triangleright(f\boxtimes\text{id})^{\uparrow L} = \big(\text{ap}\,(p\triangleright\text{pair}^{\uparrow L})(q)\big)\triangleright(f\boxtimes\text{id})^{\uparrow L}\\
\text{naturality law of ap}:\quad & = \text{ap}\big(p\triangleright\text{pair}^{\uparrow L}\triangleright(x\to x\,⨾\,(f\boxtimes\text{id}))^{\uparrow L}\big)(q)\quad.
\end{aligned}$$

The remaining difference will disappear if for any $f^{:Z\to A}$, the following equation holds:

$$f^{:Z\to A}\,⨾\,\text{pair}^{:A\to B\to A\times B}\stackrel{?}{=}\text{pair}^{:Z\to B\to Z\times B}\,⨾\,(x^{:B\to Z\times B}\to x\,⨾\,(f\boxtimes\text{id}))\quad.$$

To verify this equation, substitute the definition of `pair` into both sides:

$$\begin{aligned}
\text{left-hand side}:\quad & f\,⨾\,\text{pair} = (z^{:Z}\to f(z))\,⨾\,(a^{:A}\to b^{:B}\to a\times b) = z\to b\to f(z)\times b\quad,\\
\text{right-hand side}:\quad & \text{pair}\,⨾\,(x\to x\,⨾\,(f\boxtimes\text{id})) = (z^{:Z}\to b^{:B}\to z\times b)\,⨾\,(x^{:B\to Z\times B}\to x\,⨾\,(f\boxtimes\text{id}))\\
\text{compute composition}:\quad & = z\to(b\to z\times b)\,⨾\,(f\boxtimes\text{id}) = z\to b\to f(z)\times b\quad.
\end{aligned}$$

Both sides are now equal. □

11.3.2 The `Zippable` and `Applicative` typeclasses

The previous section showed that `map2`, `zip`, and `ap` are equivalent to each other, assuming that certain naturality laws hold. (In practice, implementations of these methods always satisfy the naturality laws with respect to each type parameter.) We can encapsulate the functionality of these methods into a "zippable functor" typeclass[8] whose `map2` and `ap` methods are defined via `zip`:

```
abstract class Zippable[L[_]: Functor] {
  def zip[A, B](la: L[A], lb: L[B]): L[(A, B)]
  def map2[A, B, C](la: L[A], lb: L[B])(f: (A, B) => C): L[C] = zip(la, lb).map(Function.untupled(f))
  def ap[A, B](lf: L[A => B], la: L[A]): L[B] = zip(lf, la).map { case (f, a) => f(a) }
}
```

[8] A similar typeclass is called "`Zip`" in `scalaz` and "`Semigroupal`" in `cats`.

Instead of using `zip`, either `map2` or `ap` could be used to implement the other methods.

In addition to these methods, it is helpful to require a `pure` method for the functor `L`. The resulting typeclass is known as `Applicative`. The `pure` method is equivalent to a "wrapped unit" (denoted `wu`, see Section 8.3.5). So, the simplest definition of the `Applicative` typeclass contains just these two methods:

```
trait Applicative[L[_]] {
  def zip[A, B](la: L[A], lb: L[B]): L[(A, B)]
  def wu: L[Unit]
}
```

Other methods (`map2`, `ap`, `pure`) can be defined separately (as extension methods) using the functor instance for `L`. However, this definition of the `Applicative` typeclass can be used also with type constructors `L[A]` that are not covariant in `A`. So, we will use this definition later in this chapter.

11.3.3 Motivation for the laws of `map2`

We now turn to laws that the methods of an applicative functor must satisfy. The motivation for the laws comes from treating `map2(la, lb)(f)` as a replacement for the following code:

```
def monadicMap2(la: L[A], lb: L[B])(f: (A, B) => C): L[C] = for {
  x <- la
  y <- lb
} yield f(x, y)
```

Unlike the code of `monadicMap2`, which uses the monad's methods, `map2(la, lb)` assumes that the effects in `la` and `lb` are independent. However, the type signatures of `monadicMap2` and `map2` are the same, and for some type constructors (such as the `Reader` monad) the results are also the same. So, we will require that `map2` should obey the same laws as the code in `monadicMap2`.

To derive the laws of `map2`, we will specialize the laws of monads (which are imposed on `flatMap` and `pure`) to the code of the form of `monadicMap2` and then express those laws in terms of `map2` and `pure`.

We begin with the associativity law of monads, which says that these three expressions are equal:

```
for {
  x <- la
  y <- lb
  z <- lc
} yield g(x, y, z)
```

```
for {
  x <- la
  (y, z) <- for {
          yy <- lb
          zz <- lc
        } yield (yy, zz)
} yield g(x, y, z)
```

```
for {
  (x, y) <- for {
              xx <- la
              yy <- lb
            } yield (xx, yy)
   z <- lc
} yield g(x, y, z)
```

Expressed through `map2`, this law gives an equation between the following function calls:

```
map3(la, lb, lc) { (a, b, c) => g(a, b, c) }
  == map2(la, map2(lb, lc) { (b, c) => (b, c) } { case (x, (y, z)) => g(x, y, z) }
  == map2(map2(la, lb) { (a, b) => (a, b) }, lc) { case ((x, y), z)) => g(x, y, z) }
```

In the code notation, this law is written as:

$$\begin{aligned}
&\text{map}_3\left(p^{:L^A}\times q^{:L^B}\times r^{:L^C}\right)\left(g^{:A\times B\times C\to D}\right)\\
&=\text{map}_2\left(p\times\text{map}_2\left(q\times r\right)\left(\text{id}^{:B\times C\to B\times C}\right)\right)\left(a^{:A}\times\left(b^{:B}\times c^{:C}\right)\to g(a\times b\times c)\right)\\
&=\text{map}_2\left(\text{map}_2\left(p\times q\right)(\text{id})\times r\right)\left(\left(a^{:A}\times b^{:B}\right)\times c^{:C}\to g(a\times b\times c)\right)\quad .
\end{aligned}$$

This law guarantees that `map3` can be expressed unambiguously through `map2` regardless of the order in which we group the arguments. This is the **associativity law** of `map2`.

Next, we consider the monadic left identity law. That law says that the following codes are equal:

```
for {
  x <- pure(a)
  y <- lb
} yield g(x, y)
```

```
for {
  // No need to define x.
  y <- lb
} yield g(a, y)
```

Writing this in terms of `map2`, we obtain the equation:

```
map2(pure(a), lb)(g) == lb.map { y => g(a, y) }
```

The monadic right identity law says that the following two expressions are equal:

```
for {
  x <- la
  y <- pure(b)
} yield g(x, y)
```

```
for {
  x <- la
  // No need to define y.
} yield g(x, b)
```

Writing this in terms of `map2`, we obtain the equation:

```
map2(la, pure(b))(g) == la.map { x => g(x, b) }
```

In the code notation, the two **identity laws** of `map2` are written as:

$$\text{left identity law}: \quad \text{map}_2\,(\text{pu}_L(a^{:A}) \times q^{:L^B})(g^{:A\times B\to C}) = q \triangleright (b^{:B} \to g(a\times b))^{\uparrow L} \quad ,$$

$$\text{right identity law}: \quad \text{map}_2\,(p^{:L^A} \times \text{pu}_L(b^{:B}))(g^{:A\times B\to C}) = p \triangleright (a^{:A} \to g(a\times b))^{\uparrow L} \quad .$$

To simplify and analyze the laws of `map2`, we will now derive the laws of `zip` that will follow once we express `map2` via `zip`.

11.3.4 Deriving the laws of `zip` from the laws of `map2`

To derive the laws of `zip` that follow from the identity and associativity laws of `map2`, we express `map2` via `zip` and then substitute into those laws:

$$\text{map}_2\,(p^{:L^A} \times q^{:L^B})(f^{:A\times B\to C}) = \text{zip}\,(p\times q) \triangleright f^{\uparrow L} \quad . \tag{11.2}$$

Begin with the associativity law and write its two sides separately:

$$\begin{aligned}
\text{left-hand side}: \quad & \text{map}_2\,(p^{:L^A} \times \text{map}_2\,(q^{:L^B} \times r^{:L^C})(\text{id}^{:B\times C\to B\times C}))\big(a^{:A} \times (b^{:B} \times c^{:C}) \to g(a\times b\times c)\big) \\
& = \text{zip}\,(p \times \text{zip}\,(q\times r)) \triangleright (a\times(b\times c) \to g(a\times b\times c))^{\uparrow L} \\
\text{right-hand side}: \quad & \overset{!}{=} \text{map}_2\,(\text{map}_2\,(p\times q)(\text{id}) \times r)\big((a^{:A}\times b^{:B}) \times c^{:C} \to g(a\times b\times c)\big) \\
& = \text{zip}\,(\text{zip}\,(p\times q)\times r) \triangleright ((a\times b)\times c) \to g(a\times b\times c))^{\uparrow L} \quad .
\end{aligned}$$

Both sides now depend on an arbitrary function $g^{:A\times B\times C\to D}$ in almost the same way, except for a rearrangement of the nested products. We can refactor the code to remove the dependency on g. Define two functions $\varepsilon_{1,23}$ and $\varepsilon_{12,3}$ that rearrange the tuples:

$$\varepsilon_{1,23} \overset{\text{def}}{=} a^{:A} \times (b^{:B} \times c^{:C}) \to a\times b\times c \quad , \qquad \varepsilon_{12,3} \overset{\text{def}}{=} (a^{:A}\times b^{:B}) \times c \to a\times b\times c \quad .$$

Then we can rewrite the two sides of the associativity law as:

$$\begin{aligned}
\text{left-hand side}: \quad & \text{zip}\,(p \times \text{zip}\,(q\times r)) \triangleright \varepsilon_{1,23}^{\uparrow L} \fatsemi g^{\uparrow L} \\
\text{right-hand side}: \quad & \overset{!}{=} \text{zip}\,(\text{zip}\,(p\times q)\times r) \triangleright \varepsilon_{12,3}^{\uparrow L} \fatsemi g^{\uparrow L} \quad .
\end{aligned}$$

We can now omit $g^{\uparrow L}$ (see Exercise 11.5.6.6) and obtain an equivalent law:

$$\text{zip}\big(p \times \text{zip}\,(q\times r)\big) \triangleright \varepsilon_{1,23}^{\uparrow L} \overset{!}{=} \text{zip}\big(\text{zip}\,(p\times q)\times r\big) \triangleright \varepsilon_{12,3}^{\uparrow L} \quad . \tag{11.3}$$

A type diagram illustrating this law is shown below:

The functions $\varepsilon_{1,23}$ and $\varepsilon_{12,3}$ implement the conversion of the equivalent types $(A \times B) \times C$ and $A \times (B \times C)$ to $A \times B \times C$. Since L is a functor, the lifted functions $\varepsilon_{1,23}^{\uparrow L}$ and $\varepsilon_{12,3}^{\uparrow L}$ produce equivalences between the types $L^{(A\times B)\times C}$, $L^{A\times(B\times C)}$, and $L^{A\times B\times C}$. With these equivalences in mind, and using the infix syntax for `zip`, we rewrite the associativity law in a simpler form:

$$p \text{ zip } (q \text{ zip } r) \cong (p \text{ zip } q) \text{ zip } r \quad . \tag{11.4}$$

In Eq. (11.4), the symbol $\cong$ denotes equality up to the type equivalence. To obtain a real equation, one would need to apply $\varepsilon_{1,23}^{\uparrow L}$ and $\varepsilon_{12,3}^{\uparrow L}$ at appropriate places. Apart from that, the law (11.4) has the usual form of an associativity law for a binary operation.

When writing laws with implied type equivalences, as in the formulation (11.4), we help build the intuition about applicative laws. We also save time because we do not write out a number of tuple-swapping functions. To avoid errors, derivations using this technique must first check that all types match up to tuple-swapping isomorphisms.

We now turn to the identity laws. Substitute Eq. (11.2) into `map2`'s left identity law:

$$\begin{aligned}
&\text{left-hand side :} \quad \text{map}_2\,(\text{pu}_L(a^{:A}) \times q^{:L^B})(g) = \text{zip}\,(\text{pu}_L(a) \times q) \triangleright g^{\uparrow L} \\
&\text{right-hand side :} \quad \stackrel{!}{=} q \triangleright (b^{:B} \to g(a \times b))^{\uparrow L} = q \triangleright (b^{:B} \to a \times b)^{\uparrow L} \triangleright g^{\uparrow L} \quad .
\end{aligned}$$

We may remove the common function $g^{\uparrow L}$ from both sides and get an equivalent law:

$$\text{zip}\,(\text{pu}_L(a) \times q) \stackrel{!}{=} q \triangleright (b \to a \times b)^{\uparrow L} \quad .$$

We express `pure` through the "wrapped unit" value (`wu`) as we did in Section 8.3.5:

$$\text{wu} : L^{\mathbb{1}} \quad , \qquad \text{pu}_L(a) = \text{wu} \triangleright (1 \to a)^{\uparrow L} \quad .$$

We obtain:

$$\begin{aligned}
&\text{left-hand side :} \quad \text{zip}\,(\text{pu}_L(a) \times q) = \text{zip}\big((\text{wu} \triangleright (1 \to a)^{\uparrow L}) \times q\big) \\
&\text{naturality law of zip :} \quad = \text{zip}\,(\text{wu} \times q) \triangleright (1 \times b^{:B} \to a \times b)^{\uparrow L} \\
&\text{right-hand side :} \quad \stackrel{!}{=} q \triangleright (b \to a \times b)^{\uparrow L} \quad .
\end{aligned}$$

To simplify this equation further, we note that the function $(1 \times b^{:B} \to a \times b)^{\uparrow L}$ in the left-hand side and the function $(b^{:B} \to a \times b)^{\uparrow L}$ in the right-hand side are similar. The only difference is between the arguments $1 \times b$ and b. But the types $\mathbb{1} \times B$ and B are equivalent. Let us define a function called `ilu` ("insert left unit") implementing this equivalence:

$$\text{ilu} : B \to \mathbb{1} \times B \quad , \qquad \text{ilu} \stackrel{\text{def}}{=} b^{:B} \to 1 \times b \quad .$$

Now we can express both sides of the law as:

$$\text{zip}\,(\text{wu} \times q) \triangleright (1 \times b^{:B} \to a \times b)^{\uparrow L} \stackrel{!}{=} q \triangleright \text{ilu}^{\uparrow L} \triangleright (1 \times b^{:B} \to a \times b)^{\uparrow L} \quad . \tag{11.5}$$

Both sides contain the function $(1 \times b^{:B} \to a \times b)^{\uparrow L}$ with an arbitrary value $a^{:A}$ of arbitrary type A. Is it correct to simplify the law by discarding that function? To show that it is, we first set $A \stackrel{\text{def}}{=} \mathbb{1}$ (since the type A may be chosen arbitrarily) and obtain the function

$$(1 \times b^{:B} \to 1 \times b)^{\uparrow L} = \text{id}^{\uparrow L} = \text{id}^{:L^{\mathbb{1}\times B} \to L^{\mathbb{1}\times B}} \quad .$$

Applying an identity function to both sides of Eq. (11.5), we find:

$$\text{zip}\,(\text{wu} \times q) = q \triangleright \text{ilu}^{\uparrow L} \quad . \tag{11.6}$$

This equation is a consequence of Eq. (11.5). At the same time, we may apply the function $(1 \times b^{:B} \to a \times b)^{\uparrow L}$, this time with an arbitrary $a^{:A}$, to both sides of Eq. (11.6) and recover Eq. (11.5). So, we have justified the simplification of the left identity law to Eq. (11.6).

Since L is a functor, the conversion function $\text{ilu}^{\uparrow L}$ implements the type equivalence $L^B \cong L^{\mathbb{1} \times B}$. Denoting this equivalence by $\cong$ and using the infix syntax for `zip`, we rewrite the left identity law as:

$$\text{wu zip } q \cong q \quad .$$

Apart from the implied conversion function, this is a familiar form of the left identity law: a value $q^{:L^B}$ remains unchanged (up to type equivalence) when "zipping" it with the special value `wu`.

A similar derivation shows that the right identity law of `zip` is:

$$\text{zip}\,(p^{:L^A} \times \text{wu}) = p \triangleright \text{iru}^{\uparrow L} \quad , \qquad \text{iru} \stackrel{\text{def}}{=} a^{:A} \to a \times 1 \quad ,$$

or using the infix syntax and the implied equivalence between the types L^A and $L^{A \times \mathbb{1}}$:

$$p \text{ zip wu} \cong p \quad .$$

So, `wu` is the "empty value" for the binary operation `zip`.

We have shown that the laws of `map2` can be simplified when formulated via `zip`. In that formulation, the laws of applicative functors are similar to the laws of a *monoid* (see Example 8.2.5.6): the binary operation is `zip` and the empty value is `wu`.

11.3.5 Commutative applicative functors and parallel computations

The main feature of applicative functors is that `zip(p, q)` uses the independence of effects expressed by `p` and `q`. For some applicative functors, the effects of `p` and `q` may be computed *in parallel* without affecting the result of `zip`. For instance, assume that `p` and `q` represent processes that compute values. Then `zip(p, q)` is a combined process that computes two values in parallel. We have seen an implementation of this idea using the `Future` type in Section 11.1.3. The same property may hold for other applicative functors as long as `zip(p, q)` does not depend on the order in which the effects of `p` and `q` are combined. The precise condition is the "commutativity" property of the applicative functor.

Definition 11.3.5.1 An applicative functor $L^\bullet$ is **commutative** if its `zip` operation satisfies the commutativity law (in addition to the standard applicative laws):

$$\text{zip}\,(p^{:L^A} \times q^{:L^B}) = \text{zip}\,(q \times p) \triangleright (b^{:B} \times a^{:A} \to a \times b)^{\uparrow L} \quad \text{or equivalently}: \quad \text{swap} \fatsemi \text{zip} = \text{zip} \fatsemi \text{swap}^{\uparrow L} \quad . \tag{11.7}$$

Here we used the standard Scala function `swap` defined by $\text{swap} \stackrel{\text{def}}{=} a \times b \to b \times a$.

The commutativity law says that `zip` commutes with `swap`. If we use `swap` as a type isomorphism between $A \times B$ and $B \times A$, we may write the commutativity law more concisely and suggestively:

$$p \text{ zip } q \cong q \text{ zip } p \quad .$$

This definition agrees with that of the commutative monad (see Exercise 10.2.9.14). A monad's commutativity is defined via code that is equivalent to the `map2` operation:

```
def map2[A, B, C](p: M[A], q: M[B])(f: (A, B) => C): M[C] = for {
    x <- p                // Effects of p and q are independent of
    y <- q                // the values x or y.
} yield f(x, y)
```

So, the definition of a commutative monad is that the corresponding `map2` operation (defined via `map` and `flatMap`) must satisfy the following commutativity property:

$$\text{map}_2(p \times q)(f) = \text{map}_2(q \times p)\big(b \times a \to f(a \times b)\big) \quad .$$

Expressing `map2` through `zip` as $\text{map}_2(p \times q)(f) = \text{zip}\,(p \times q) \triangleright f^{\uparrow L}$, we obtain Eq. (11.7).

Most monads are not commutative; an exception is the `Reader` monad. So, even when the effects are independent of the previous values, it is not possible to parallelize monadic code automatically. In contrast, most applicative functors turn out to be commutative, so their `zip` methods could (in principle) have a parallel implementation. An exception is the parsing functor from Section 11.2.7. One does not expect to be able to parse different pieces of a file in parallel, because correct parsing often depends on the success or failure of parsing of previous portions of the data.

An example of a data structure that can automatically parallelize computations is Apache Spark's `RDD` class. It is important that `RDD[_]` is a commutative applicative functor but *not* a monad.[9] This agrees with the intuition that monadic programs are not automatically parallelizable.

In many cases, a commutative applicative functor will also be a *non-commutative* monad. Some examples are `Option`, `Either`, and `List` functors. For these and other functors, the `map2` and `zip` methods must be defined separately and not derived from the monadic methods. If we *did* choose to define the `map2` method via the monadic methods, there are two possible implementations:

```
map2(p, q)(f) == for { // Direct order.
    a <- p             // p: L[A]
    b <- q             // q: L[B]
} yield f(a, b)
```

```
map2(p, q)(f) == for { // Inverse order.
    b <- q             // q: L[B]
    a <- p             // p: L[A]
} yield f(a, b)
```

The difference is only in the order in which the effects of `p` and `q` are concatenated. The two corresponding `zip` functions differ by the permutation of the effects:

$$\text{zip}\,(p \times q) \stackrel{\text{def}}{=} p \triangleright \text{flm}_L(a \to q \triangleright (b \to a \times b)^{\uparrow L}) \quad , \qquad \text{zip}'\,(p \times q) = \text{zip}\,(q \times p) \triangleright \text{swap}^{\uparrow L} \quad .$$

For commutative monads, there will be no difference between zip′ and zip.

When an applicative functor is commutative, its associativity law has a simplified form:

Statement 11.3.5.2 The associativity law of a *commutative* applicative functor L is equivalent to:

$$\text{zip}\big(p^{:L^A} \times \text{zip}\big(q^{:L^B} \times r^{:L^C}\big)\big) = \text{zip}\,\big(r \times \text{zip}\,(q \times p)\big) \triangleright (c^{:C} \times (b^{:B} \times a^{:A}) \to a \times (b \times c))^{\uparrow L} \quad . \tag{11.8}$$

Up to the tuple-swapping isomorphism, Eq. (11.3.5.2) has the form where only p and r are swapped:

$$p \text{ zip } (q \text{ zip } r) \cong r \text{ zip } (q \text{ zip } p) \quad . \tag{11.9}$$

Proof We need to demonstrate two directions of the equivalence: **(1)** Derive Eq. (11.9) assuming that the associativity law holds. **(2)** Derive the associativity law assuming that Eq. (11.9) holds.

To save time, we will write $\cong$ to imply tuple-swapping isomorphisms wherever necessary.

(1) Begin with the left-hand side of Eq. (11.9) and apply Eq. (11.4):

$$p \text{ zip } (q \text{ zip } r) \cong \underline{(p \text{ zip } q)} \text{ zip } \underline{r}$$

$$\text{twice use the commutativity law of zip}: \quad \cong r \text{ zip } (\underline{p} \text{ zip } \underline{q}) \cong r \text{ zip } (q \text{ zip } p) \quad .$$

This is the right-hand side of Eq. (11.9)

(2) Begin with the right-hand side of Eq. (11.9) and apply the commutativity law:

$$r \text{ zip } (\underline{q} \text{ zip } \underline{p}) = \underline{r} \text{ zip } \underline{(p \text{ zip } q)} = (p \text{ zip } q) \text{ zip } r$$

$$\text{use Eq. (11.9)}: \quad \stackrel{!}{=} p \text{ zip } (q \text{ zip } r) \quad .$$

We have derived Eq. (11.4). □

This statement makes it easier to check the associativity law of `zip` if we know that the commutativity law holds. We just need to write the code for $\text{zip}\,\big(p \times \text{zip}\,(q \times r)\big)$ and swap $p^{:L^A}$ and $r^{:L^C}$ in that code. The result must be the same up to swapping the result values of types A and C.

For commutative applicative functors, it is also sufficient to check only *one* of the identity laws:

[9] The `Spark` library does not support values of type `RDD[RDD[A]]`. The `RDD` class's `cartesian` method has the type signature corresponding to `zip`. A method called "`flatMap`" exists but does not have the type signature of a monad's `flatMap`. See https://spark.apache.org/docs/3.2.1/rdd-programming-guide.html

Statement 11.3.5.3 The two identity laws of a commutative applicative functor L are equivalent.

Proof We need to show the two directions of the isomorphism:

(1) Derive $p \text{ zip } \text{wu} \cong p$ from $\text{wu} \text{ zip } q \cong q$. Use the commutativity law:

$$\underline{p} \text{ zip } \underline{\text{wu}} \cong \text{wu} \text{ zip } p \cong p \quad ,$$

where we use the left identity law with $q \stackrel{\text{def}}{=} p$.

(2) Derive $\text{wu} \text{ zip } q \cong q$ from $p \text{ zip } \text{wu} \cong p$. Use the commutativity law:

$$\underline{\text{wu}} \text{ zip } \underline{q} \cong q \text{ zip } \text{wu} \cong q \quad ,$$

where we use the right identity law with $p \stackrel{\text{def}}{=} q$. □

11.3.6 Constructions of applicative functors

We will now perform structural analysis of applicative functors, going through all possible type constructions. In each case, we will implement the `zip` and `wu` methods and verify their laws, indicating whether the commutativity law holds. We will also compare the applicative functor constructions to the corresponding constructions for monads (Section 10.2.8). As before, we say "F is applicative" when there is at least one lawful implementation of `zip` and `wu`.

Note that any monad will already have a lawful implementation of `zip` and `wu` since the laws of `map2` and `zip` were derived directly from the monad laws. If the monad is non-commutative, there will be two inequivalent implementations of `zip`. So, each monad construction gives at least one applicative construction. We will focus on constructions that generate applicative functors and have no analogous monad constructions.

Type parameters Three type constructions are based on using just type parameters: a constant functor, $L^A \stackrel{\text{def}}{=} Z$, the identity functor $L^A \stackrel{\text{def}}{=} A$, and the functor composition, $L^A \stackrel{\text{def}}{=} F^{G^A}$.

Given a fixed monoid type Z, the constant functor $L^A \stackrel{\text{def}}{=} Z$ is applicative. To see this, consider the operation `zip` of type $L^A \times L^B \to L^{A\times B}$, which in this case becomes just $Z \times Z \to Z$, the monoid Z's `combine` function. The value `wu` of type $L^{\mathbb{1}} = Z$ is just the monoid Z's empty value (e_Z). The laws of `zip` then reduce to the monoid laws of Z. For the same reason, the applicative functor $L^A \stackrel{\text{def}}{=} Z$ will be commutative if the monoid Z is commutative.

Comparing this with the corresponding monad construction, we note that $L^A \stackrel{\text{def}}{=} Z$ is not a monad unless $Z = \mathbb{1}$. For an arbitrary monoid type Z, the functor $L^A \stackrel{\text{def}}{=} Z$ is only a semimonad.

The identity functor $L^A \stackrel{\text{def}}{=} A$ is applicative and commutative: the `zip` operation is the identity function of type $A \times B \to A \times B$, and the wrapped unit (`wu`) is just the unit value. All required laws hold for the identity function.

The functor composition $L \stackrel{\text{def}}{=} F \circ G$ of two applicative functors F and G is again applicative:

Statement 11.3.6.1 For any two applicative functors F and G with lawful `zip` and `pure` methods, the functor $L^A \stackrel{\text{def}}{=} F^{G^A}$ is also applicative. Its `zip` and `pure` methods are defined by:

$$\text{zip}_L : F^{G^A} \times F^{G^B} \to F^{G^{A\times B}} \quad , \qquad \text{zip}_L \stackrel{\text{def}}{=} \text{zip}_F \,⨾\, \text{zip}_G^{\uparrow F} \quad ,$$

$$\text{pu}_L : A \to F^{G^A} \quad , \qquad \text{pu}_L \stackrel{\text{def}}{=} \text{pu}_G \,⨾\, \text{pu}_F \quad , \qquad \text{wu}_L \stackrel{\text{def}}{=} \text{pu}_F(\text{wu}_G) \quad .$$

If F and G are commutative applicative functors, then so is L.

Proof We note that the lifting to L is defined by $f^{\uparrow L} \stackrel{\text{def}}{=} f^{\uparrow G \uparrow F} = (f^{\uparrow G})^{\uparrow F}$.

To verify the left identity law (11.6):

$$\begin{aligned}
&\text{expect to equal } p \triangleright \text{ilu}^{\uparrow L} : && \text{zip}_L(\text{wu}_L \times p^{:F^{G^A}}) = (\underline{\text{pu}_F(\text{wu}_G)} \times p) \triangleright \underline{\text{zip}_F} \,⨾\, \text{zip}_G^{\uparrow F} \\
&\text{left identity law of } \text{zip}_F : && = p \triangleright (g^{:G^A} \to \underline{\text{wu}_G \times g})^{\uparrow F} \,⨾\, \text{zip}_G^{\uparrow F} \\
&\text{composition under } {}^{\uparrow F} : && = p \triangleright \left(g^{:G^A} \to \underline{\text{zip}_G(\text{wu}_G \times g)}\right)^{\uparrow F} \\
&\text{left identity law of } \text{zip}_G : && = p \triangleright (\underline{g \to g} \triangleright \text{ilu}^{\uparrow G})^{\uparrow F} = p \triangleright (\text{ilu}^{\uparrow G})^{\uparrow F} = p \triangleright \text{ilu}^{\uparrow L} \quad .
\end{aligned}$$

To verify the right identity law:

$$\begin{aligned}
&\text{expect to equal } p \triangleright \text{iru}^{\uparrow L}: && \text{zip}_L(p^{:F^{G^A}} \times \text{wu}_L) = (p \times \underline{\text{pu}_F(\text{wu}_G)}) \triangleright \underline{\text{zip}_F} \,\text{⨾}\, \text{zip}_G^{\uparrow F} \\
&\text{right identity law of zip}_F: && = p \triangleright (g^{:G^A} \to \underline{g \times \text{wu}_G})^{\uparrow F} \,\text{⨾}\, \underline{\text{zip}_G^{\uparrow F}} = p \triangleright \left(g^{:G^A} \to \underline{\text{zip}_G(g \times \text{wu}_G)}\right)^{\uparrow F} \\
&\text{right identity law of zip}_G: && = p \triangleright (\underline{g \to g \triangleright} \text{iru}^{\uparrow G})^{\uparrow F} = p \triangleright (\text{iru}^{\uparrow G})^{\uparrow F} = p \triangleright \text{iru}^{\uparrow L} \quad .
\end{aligned}$$

To verify the associativity law, first substitute the definition of zip_L into one side:

$$\begin{aligned}
&\text{left-hand side}: && \text{zip}_L(p \times \text{zip}_L(q \times r)) \triangleright \varepsilon_{1,23}^{\uparrow L} = (p \times \text{zip}_L(q \times r)) \triangleright \text{zip}_F \,\text{⨾}\, \text{zip}_G^{\uparrow F} \,\text{⨾}\, \underline{\varepsilon_{1,23}^{\uparrow L}} \\
&\text{definition of } {}^{\uparrow L}: && = (p \times (\text{zip}_F(q \times r) \triangleright \underline{\text{zip}_G^{\uparrow F}})) \triangleright \underline{\text{zip}_F} \,\text{⨾}\, \text{zip}_G^{\uparrow F} \,\text{⨾}\, \varepsilon_{1,23}^{\uparrow G \uparrow F} \\
&\text{naturality law of zip}_F: && = (p \times \text{zip}_F(q \times r)) \triangleright \text{zip}_F \,\text{⨾}\, (g \times k^{:G^B \times G^C} \to \underline{g \times \text{zip}_G(k)})^{\uparrow F} \,\text{⨾}\, \text{zip}_G^{\uparrow F} \,\text{⨾}\, \varepsilon_{1,23}^{\uparrow G \uparrow F} \\
&\text{composition under } {}^{\uparrow F}: && = \text{zip}_F(p \times \text{zip}_F(q \times r)) \triangleright (g \times (h \times j) \to \text{zip}_G(g \times \text{zip}_G(h \times j)) \triangleright \varepsilon_{1,23}^{\uparrow G})^{\uparrow F} \quad .
\end{aligned}$$

Now rewrite the right-hand side in a similar way:

$$\begin{aligned}
&\text{right-hand side}: && \text{zip}_L(\text{zip}_L(p \times q) \times r) \triangleright \varepsilon_{12,3}^{\uparrow L} = (\text{zip}_L(p \times q) \times r) \triangleright \text{zip}_F \,\text{⨾}\, \text{zip}_G^{\uparrow F} \,\text{⨾}\, \varepsilon_{12,3}^{\uparrow L} \\
&\text{definition of } {}^{\uparrow L}: && = ((\text{zip}_F(p \times q) \triangleright \underline{\text{zip}_G^{\uparrow F}}) \times r) \triangleright \underline{\text{zip}_F} \,\text{⨾}\, \text{zip}_G^{\uparrow F} \,\text{⨾}\, \varepsilon_{12,3}^{\uparrow G \uparrow F} \\
&\text{naturality law of zip}_F: && = (\text{zip}_F(p \times q) \times r) \triangleright \text{zip}_F \,\text{⨾}\, (k^{:G^A \times G^B} \times j \to \underline{\text{zip}_G(k) \times j})^{\uparrow F} \,\text{⨾}\, \text{zip}_G^{\uparrow F} \,\text{⨾}\, \varepsilon_{12,3}^{\uparrow G \uparrow F} \\
&\text{composition under } {}^{\uparrow F}: && = \text{zip}_F(\text{zip}_F(p \times q) \times r) \triangleright ((g \times h) \times j \to \text{zip}_G(\text{zip}_G(g \times h) \times j) \triangleright \varepsilon_{12,3}^{\uparrow G})^{\uparrow F} \quad .
\end{aligned}$$

The two sides become equal after using the associativity laws of zip_F and zip_G:

$$\begin{aligned}
\text{zip}_F(p \times \text{zip}_F(q \times r)) \triangleright \varepsilon_{1,23}^{\uparrow F} &= \text{zip}_F(\text{zip}_F(p \times q) \times r) \triangleright \varepsilon_{12,3}^{\uparrow F} \quad , \\
\text{zip}_G(g \times \text{zip}_G(h \times j)) \triangleright \varepsilon_{1,23}^{\uparrow G} &= \text{zip}_G(\text{zip}_G(g \times h) \times j) \triangleright \varepsilon_{12,3}^{\uparrow G} \quad .
\end{aligned}$$

To verify the commutativity law (11.7) for L, we assume that the law holds for F and G:

$$\begin{aligned}
&\text{expect to equal } (\text{zip}_L \,\text{⨾}\, \text{swap}^{\uparrow L}): && \text{swap} \,\text{⨾}\, \text{zip}_L = \underline{\text{swap} \,\text{⨾}\, \text{zip}_F} \,\text{⨾}\, \text{zip}_G^{\uparrow F} \\
&\text{commutativity law of } F: && = \text{zip}_F \,\text{⨾}\, \underline{\text{swap}^{\uparrow F} \,\text{⨾}\, \text{zip}_G^{\uparrow F}} = \text{zip}_F \,\text{⨾}\, (\underline{\text{swap} \,\text{⨾}\, \text{zip}_G})^{\uparrow F} \\
&\text{commutativity law of } G \text{ under } {}^{\uparrow F}: && = \text{zip}_F \,\text{⨾}\, \text{zip}_G^{\uparrow F} \,\text{⨾}\, \text{swap}^{\uparrow G \uparrow F} = \text{zip}_L \,\text{⨾}\, \text{swap}^{\uparrow L} \quad .
\end{aligned}$$

□

Products Similarly to most other typeclasses, the product of applicative functors is applicative:

Statement 11.3.6.2 For any applicative functors F and G with lawful `zip` and `wu` methods, the functor $L^A \stackrel{\text{def}}{=} F^A \times G^A$ is also applicative. Its `zip` and `wu` methods are defined by:

$$\begin{aligned}
&\text{zip}_L : (F^A \times G^A) \times (F^B \times G^B) \to F^{A \times B} \times G^{A \times B} \quad , \\
&\text{zip}_L((m^{:F^A} \times n^{:G^A}) \times (p^{:F^B} \times q^{:G^B})) \stackrel{\text{def}}{=} \text{zip}_F(m \times p) \times \text{zip}_G(n \times q) \quad , \\
&\text{wu}_L : F^{\mathbb{1}} \times G^{\mathbb{1}} \quad , \qquad \text{wu}_L \stackrel{\text{def}}{=} \text{wu}_F \times \text{wu}_G \quad .
\end{aligned}$$

If F and G are commutative applicative functors, then so is L.

Proof We note that the lifting to L is defined by $f^{\uparrow L} \stackrel{\text{def}}{=} f^{\uparrow F} \boxtimes f^{\uparrow G}$.
To verify the left identity law of zip_L:

$$\begin{aligned}
\text{expect to equal } (p \times q) \triangleright \text{ilu}^{\uparrow L}: \quad & \text{zip}_L(\underline{\text{wu}_L} \times (p^{:F^A} \times q^{:G^A})) = \underline{\text{zip}_L}((\text{wu}_F \times \text{wu}_G) \times (p \times q)) \\
\text{definition of } \text{zip}_L: \quad & = \text{zip}_F(\text{wu}_F \times p) \times \text{zip}_G(\text{wu}_G \times q) \\
\text{left identity laws of } \text{zip}_F \text{ and } \text{zip}_G: \quad & = (p \triangleright \text{ilu}^{\uparrow F}) \times (q \triangleright \text{ilu}^{\uparrow G}) = (p \times q) \triangleright (\text{ilu}^{\uparrow F} \boxtimes \text{ilu}^{\uparrow G}) \\
\text{definition of } {}^{\uparrow L}: \quad & = (p \times q) \triangleright \text{ilu}^{\uparrow L} \quad .
\end{aligned}$$

To verify the right identity law of zip_L:

$$\begin{aligned}
\text{expect to equal } (p \times q) \triangleright \text{iru}^{\uparrow L}: \quad & \text{zip}_L((p^{:F^A} \times q^{:G^A}) \times \underline{\text{wu}_L}) = \underline{\text{zip}_L}((p \times q) \times (\text{wu}_F \times \text{wu}_G)) \\
\text{definition of } \text{zip}_L: \quad & = \text{zip}_F(p \times \text{wu}_F) \times \text{zip}_G(q \times \text{wu}_G) \\
\text{right identity laws of } \text{zip}_F \text{ and } \text{zip}_G: \quad & = (p \triangleright \text{iru}^{\uparrow F}) \times (q \triangleright \text{iru}^{\uparrow G}) = (p \times q) \triangleright (\text{iru}^{\uparrow F} \boxtimes \text{iru}^{\uparrow G}) \\
\text{definition of } {}^{\uparrow L}: \quad & = (p \times q) \triangleright \text{iru}^{\uparrow L} \quad .
\end{aligned}$$

To verify the associativity law, begin with its left-hand side and use the definition of zip_L:

$$\begin{aligned}
\text{left-hand side}: \quad & \text{zip}_L((p_1^{:F^A} \times p_2^{:G^A}) \times \text{zip}_L((q_1^{:F^B} \times q_2^{:G^B}) \times (r_1^{:F^C} \times r_2^{:G^C}))) \triangleright \underline{\varepsilon^{\uparrow L}_{1,23}} \\
& = \text{zip}_L((p_1 \times p_2) \times (\text{zip}_F(q_1 \times r_1) \times \text{zip}_G(q_2 \times r_2))) \triangleright (\varepsilon^{\uparrow F}_{1,23} \boxtimes \varepsilon^{\uparrow G}_{1,23}) \\
& = (\underline{\text{zip}_F(p_1 \times \text{zip}_F(q_1 \times r_1)) \triangleright \varepsilon^{\uparrow F}_{1,23}}) \times (\underline{\text{zip}_G(p_2 \times \text{zip}_G(q_2 \times r_2)) \triangleright \varepsilon^{\uparrow G}_{1,23}}) \quad .
\end{aligned}$$

The right-hand side is rewritten in a similar way:

$$\begin{aligned}
\text{right-hand side}: \quad & \text{zip}_L(\text{zip}_L((p_1^{:F^A} \times p_2^{:G^A}) \times (q_1^{:F^B} \times q_2^{:G^B})) \times (r_1^{:F^C} \times r_2^{:G^C})) \triangleright \underline{\varepsilon^{\uparrow L}_{12,3}} \\
& = \text{zip}_L((\text{zip}_F(p_1 \times q_1) \times \text{zip}_G(p_2 \times q_2)) \times (r_1 \times r_2))) \triangleright (\varepsilon^{\uparrow F}_{12,3} \boxtimes \varepsilon^{\uparrow G}_{12,3}) \\
& = (\underline{\text{zip}_F(\text{zip}_F(p_1 \times q_1) \times r_1) \triangleright \varepsilon^{\uparrow F}_{12,3}}) \times (\underline{\text{zip}_G(\text{zip}_G(p_2 \times q_2) \times r_2) \triangleright \varepsilon^{\uparrow G}_{12,3}}) \quad .
\end{aligned}$$

The underlined expressions in both sides are equal due to associativity laws of zip_F and zip_G.
To verify the commutativity law of L assuming it holds for F and G:

$$\begin{aligned}
\text{expect to equal } \text{zip}_L((p \times q) \times (m \times n)): \quad & \text{zip}_L((m \times n) \times (p \times q)) \triangleright \text{swap}^{\uparrow L} \\
\text{definitions of } \text{zip}_L \text{ and } {}^{\uparrow L}: \quad & = (\text{zip}_F(m \times p) \times \text{zip}_G(n \times q)) \triangleright (\text{swap}^{\uparrow F} \boxtimes \text{swap}^{\uparrow G}) \\
\text{definition of } \boxtimes: \quad & = (\text{zip}_F(m \times p) \triangleright \text{swap}^{\uparrow F}) \times (\text{zip}_G(n \times q) \triangleright \text{swap}^{\uparrow G}) \\
\text{commutativity laws of } F \text{ and } G: \quad & = \text{zip}_F(p \times m) \times \text{zip}_G(q \times n) = \text{zip}_L((p \times q) \times (m \times n)) \quad .
\end{aligned}$$

□

Co-products The co-product $F^A + G^A$ of two arbitrary applicative functors F and G is not always applicative (just as with monads). One case where the `zip` method cannot be implemented for $F^A + G^A$ was shown in Example 8.4.1.10(b). However, the following statements demonstrate that the type constructors $L^A \stackrel{\text{def}}{=} Z + F^A$ and $L^A \stackrel{\text{def}}{=} A + F^A$ are applicative functors.

Statement 11.3.6.3 If $F^\bullet$ is applicative and Z is a fixed monoid type then $L^A \stackrel{\text{def}}{=} Z + F^A$ is applicative:

$$\text{zip}_L : (Z + F^A) \times (Z + F^B) \to Z + F^{A \times B} \quad , \qquad \text{zip}_L \stackrel{\text{def}}{=} \begin{array}{|c||cc|} & Z & F^{A \times B} \\ \hline Z \times Z & z_1 \times z_2 \to z_1 \oplus z_2 & \mathbb{0} \\ F^A \times Z & _^{:F^A} \times z \to z & \mathbb{0} \\ Z \times F^B & z \times _^{:F^B} \to z & \mathbb{0} \\ F^A \times F^B & \mathbb{0} & \text{zip}_F \end{array} \quad .$$

The "wrapped unit" $(\text{wu}_L : Z + F^{\mathbb{1}})$ is defined as $\text{wu}_L \stackrel{\text{def}}{=} \mathbb{0}^{:Z} + \text{wu}_F$. If Z is a commutative monoid and $F^\bullet$ is commutative then $L^\bullet$ is also commutative.

Proof We will verify the laws of zip_L and wu_L, assuming that F is a lawful applicative functor. The lifting to L is defined in the standard way:

$$(f^{:A\to B})^{\uparrow L} \stackrel{\text{def}}{=} \left|\begin{array}{c||cc} & Z & F^B \\ \hline Z & \text{id} & \mathbb{0} \\ F^A & \mathbb{0} & f^{\uparrow F} \end{array}\right| \quad .$$

To verify the left identity law, we use the left identity law of zip_F:

$$\text{zip}_L(\text{wu}_L \times p^{:Z+F^B}) = \text{zip}_L((\mathbb{0}+\text{wu}_F)\times p) = (\text{wu}_F \times p) \triangleright \left|\begin{array}{c||cc} & Z & F^{\mathbb{1}\times B} \\ \hline F^{\mathbb{1}}\times Z & _^{:F^{\mathbb{1}}}\times z \to z & \mathbb{0} \\ F^{\mathbb{1}}\times F^B & \mathbb{0} & \text{zip}_F \end{array}\right|$$

$$= p \triangleright \left|\begin{array}{c||cc} & Z & F^{\mathbb{1}\times B} \\ \hline Z & \text{id} & \mathbb{0} \\ F^B & \mathbb{0} & k^{:F^B} \to \underline{\text{zip}_F(\text{wu}_F \times k)} \end{array}\right| = p \triangleright \left|\begin{array}{c||cc} & Z & F^{\mathbb{1}\times B} \\ \hline Z & \text{id} & \mathbb{0} \\ F^B & \mathbb{0} & k \to k \triangleright \text{ilu}^{\uparrow F} \end{array}\right| = p \triangleright \text{ilu}^{\uparrow L} \quad .$$

To verify the right identity law, we write a similar calculation:

$$\text{zip}_L(p^{:Z+F^A} \times \text{wu}_L) = \text{zip}_L(p \times (\mathbb{0}+\text{wu}_F)) = (p \times \text{wu}_F) \triangleright \left|\begin{array}{c||cc} & Z & F^{A\times\mathbb{1}} \\ \hline Z\times F^{\mathbb{1}} & z\times_^{:F^{\mathbb{1}}} \to z & \mathbb{0} \\ F^A\times F^{\mathbb{1}} & \mathbb{0} & \text{zip}_F \end{array}\right|$$

$$= p \triangleright \left|\begin{array}{c||cc} & Z & F^{A\times\mathbb{1}} \\ \hline Z & \text{id} & \mathbb{0} \\ F^A & \mathbb{0} & k^{:F^A} \to \underline{\text{zip}_F(k \times \text{wu}_F)} \end{array}\right| = p \triangleright \left|\begin{array}{c||cc} & Z & F^{A\times\mathbb{1}} \\ \hline Z & \text{id} & \mathbb{0} \\ F^A & \mathbb{0} & k \to k \triangleright \text{iru}^{\uparrow F} \end{array}\right| = p \triangleright \text{iru}^{\uparrow L} \quad .$$

To verify the associativity law, we use a trick to avoid long derivations. The two sides of the associativity law are expressions of type $Z + F^{A\times B\times C}$:

$$\text{left-hand side}: \quad \text{zip}_L(p^{:Z+F^A} \times \text{zip}_L(q^{:Z+F^B} \times r^{:Z+F^C})) \triangleright \varepsilon^{\uparrow L}_{1,23} \quad ,$$
$$\text{right-hand side}: \quad \text{zip}_L(\text{zip}_L(p^{:Z+F^A} \times q^{:Z+F^B}) \times r^{:Z+F^C}) \triangleright \varepsilon^{\uparrow L}_{12,3} \quad .$$

Since each of the arguments p, q, r may be in one of the two parts of the disjunction type $Z + F^\bullet$, we have 8 cases. We note, however, that the code of $\text{zip}_L(p \times q)$ will return a value of type $Z + \mathbb{0}$ whenever at least one of the arguments (p, q) is of type $Z + \mathbb{0}$. So, a composition of two `zip` operations will also return a value of type $Z + \mathbb{0}$ whenever at least one of the arguments (p, q, r) is of type $Z + \mathbb{0}$. So, we need to consider the following two cases:

(1) At least one of p, q, r is of type $Z + \mathbb{0}$. In this case, any arguments of type $\mathbb{0} + F^\bullet$ are ignored by zip_L, while the arguments of type $Z + \mathbb{0}$ are combined using the monoid Z's binary operation $(\oplus)$. So, the result of the `zip` operation is the same if we replace any arguments (p, q, r) of type $\mathbb{0} + F^\bullet$ by the empty value e_Z. For example:

$$\text{zip}_L\big((z+\mathbb{0})\times(\mathbb{0}+k^{:F^A})\big) = z + \mathbb{0} = \text{zip}_L\big((z+\mathbb{0})\times(e_Z+\mathbb{0})\big) \quad .$$

After this replacement, we have three arguments $(z_1 + \mathbb{0}, z_2 + \mathbb{0}, z_3 + \mathbb{0})$ instead of p, q, r, and the function zip_L reduces to the operation $\oplus$, for which the associativity law holds by assumption.

(2) All of p, q, r are of type $\mathbb{0} + F^\bullet$. In this case, zip_L reduces to zip_F, which satisfies the associativity law by assumption.

To verify the commutativity law of L, use the code matrix for `swap` with the relevant types:

$$\text{swap} \fatsemi \text{zip}_L = \begin{array}{c||cccc|} & Z\times Z & F^B\times Z & Z\times F^A & F^B\times F^A \\ \hline Z\times Z & \text{swap} & \mathbb{0} & \mathbb{0} & \mathbb{0} \\ F^A\times Z & \mathbb{0} & \mathbb{0} & \text{swap} & \mathbb{0} \\ Z\times F^B & \mathbb{0} & \text{swap} & \mathbb{0} & \mathbb{0} \\ F^A\times F^B & \mathbb{0} & \mathbb{0} & \mathbb{0} & \text{swap} \end{array} \fatsemi \begin{array}{c||cc|} & Z & F^{B\times A} \\ \hline Z\times Z & z_1\times z_2 \to z_1\oplus z_2 & \mathbb{0} \\ F^B\times Z & _\times z\to z & \mathbb{0} \\ Z\times F^A & z\times_\to z & \mathbb{0} \\ F^B\times F^A & \mathbb{0} & \text{zip}_F \end{array}$$

$$= \begin{array}{c||cc|} & Z & F^{B\times A} \\ \hline Z\times Z & z_1\times z_2\to z_2\oplus z_1 & \mathbb{0} \\ F^A\times Z & _\times z\to z & \mathbb{0} \\ Z\times F^B & z\times_\to z & \mathbb{0} \\ F^A\times F^B & \mathbb{0} & \text{swap}\fatsemi\text{zip}_F \end{array} .$$

By assumption, $\text{swap}\fatsemi\text{zip}_F = \text{zip}_F\fatsemi\text{swap}^{\uparrow F}$. Next, we need the code for the lifted $\text{swap}^{\uparrow L}$:

$$\text{swap}^{\uparrow L} = \begin{array}{c||cc|} & Z & F^{B\times A} \\ \hline Z & \text{id} & \mathbb{0} \\ F^{A\times B} & & \text{swap}^{\uparrow F} \end{array} .$$

We can now transform the right-hand side of the commutativity law:

$$\text{zip}_L\fatsemi\text{swap}^{\uparrow L} = \begin{array}{c||cc|} & Z & F^{B\times A} \\ \hline Z\times Z & z_1\times z_2\to z_1\oplus z_2 & \mathbb{0} \\ F^A\times Z & _\times z\to z & \mathbb{0} \\ Z\times F^B & z\times_\to z & \mathbb{0} \\ F^A\times F^B & \mathbb{0} & \text{zip}_F\fatsemi\text{swap}^{\uparrow F} \end{array} .$$

The difference between the sides disappears if Z is a commutative monoid ($z_1\oplus z_2 = z_2\oplus z_1$). □

Statement 11.3.6.4 If $F^\bullet$ is applicative then $L^A \stackrel{\text{def}}{=} A + F^A$ is also applicative:

$$\text{zip}_L : (A+F^A)\times(B+F^B)\to A\times B + F^{A\times B} \quad , \quad \text{zip}_L \stackrel{\text{def}}{=} \begin{array}{c||cc|} & A\times B & F^{A\times B} \\ \hline A\times B & \text{id} & \mathbb{0} \\ F^A\times B & \mathbb{0} & (\text{id}\boxtimes\text{pu}_F)\fatsemi\text{zip}_F \\ A\times F^B & \mathbb{0} & (\text{pu}_F\boxtimes\text{id})\fatsemi\text{zip}_F \\ F^A\times F^B & \mathbb{0} & \text{zip}_F \end{array} .$$

The `wu` method is defined by $\text{wu}_L \stackrel{\text{def}}{=} 1 + \mathbb{0}^{:F^1}$. If $F^\bullet$ is commutative then $L^\bullet$ is also commutative.

Proof We will use Statement 11.3.6.7, where the same properties are demonstrated for a more general functor $L^A \stackrel{\text{def}}{=} H^A + F^A$. We will set $H^A \stackrel{\text{def}}{=} A$ in Statement 11.3.6.7 and obtain the present statement because the compatibility law holds automatically for $\text{ex}_H \stackrel{\text{def}}{=} \text{id}$ and $\text{zip}_H \stackrel{\text{def}}{=} \text{id}$. □

It is important that Statements 11.3.6.3–11.3.6.4 define the methods `zip` and `wu` differently for $L^A \stackrel{\text{def}}{=} Z + F^A$ and for $L^A \stackrel{\text{def}}{=} A + F^A$. Only then the applicative laws of `zip` and `wu` will hold for those functors.

Also, those definitions agree with the code of `zip` and `wu` for the `Either` functor (Section 11.1.2). The `Either` functor ($L^A \stackrel{\text{def}}{=} Z + A$ with a monoid type Z) is constructed as $L^A \stackrel{\text{def}}{=} Z + F^A$ with $F^A \stackrel{\text{def}}{=} A$ (via Statement 11.3.6.3) or as $L^A \stackrel{\text{def}}{=} A + F^A$ with $F^A \stackrel{\text{def}}{=} Z$ (via Statement 11.3.6.4).

The following statement generalizes the construction $L^A \stackrel{\text{def}}{=} A + F^A$ to $L^A \stackrel{\text{def}}{=} H^A + F^A$ where $H^\bullet$ is applicative and at the same time co-pointed (see Section 8.3.7). The code for zip_L will use the co-pointed functor's `extract` method. The next statement shows that zip_L will obey the applicative laws if a special **compatibility law** holds for `extract` and `zip`:

$$\text{ex}_H(\text{zip}_H(p^{:H^A} \times q^{:H^B})) = \text{ex}_H(p) \times \text{ex}_H(q) \quad , \quad \text{or equivalently}: \quad \text{zip}_H \,⨾\, \text{ex}_H = \text{ex}_H \boxtimes \text{ex}_H \quad . \tag{11.10}$$

A simple example of a co-pointed applicative functor is $H^A \stackrel{\text{def}}{=} A \times G^A$:

Statement 11.3.6.5 Given any applicative functor $G^\bullet$, the functor $H^A \stackrel{\text{def}}{=} A \times G^A$ is applicative and co-pointed. The compatibility law (11.10) holds if the `extract` method is defined by $\text{ex}_H \stackrel{\text{def}}{=} \pi_1 = a^{:A} \times _^{:G^A} \to a$.

Proof The applicative and co-pointed properties of H follow from the product constructions of applicative functors (Statement 11.3.6.2) and co-pointed functors (Section 8.3.7). To verify the compatibility law (11.10), we write the definition of zip_H:

$$\text{zip}_H\left((a^{:A} \times g_1^{:G^A}) \times (b^{:B} \times g_2^{:G^B})\right) = (a \times b) \times \text{zip}_G(g_1 \times g_2) \quad ,$$

and apply `extract` to the last expression:

$$\begin{aligned} &\text{ex}_H\left(\text{zip}_H\left((a^{:A} \times g_1^{:G^A}) \times (b^{:B} \times g_2^{:G^B})\right)\right) = \text{ex}_H\left((a \times b) \times \text{zip}_G(g_1 \times g_2)\right) = a \times b \quad , \\ &\text{ex}_H(a \times g_1) \times \text{ex}_H(b \times g_2) = a \times b \quad . \end{aligned}$$

□

It is not known (Problem 15.3.2.3) whether there exist co-pointed applicative functors that satisfy the compatibility law (11.10) but are *not* of the form $A \times G^A$. Here is an example of an applicative functor not of the form $A \times G^A$ that violates the compatibility law:

Statement 11.3.6.6 Given any monoid type Z, the functor $F^A \stackrel{\text{def}}{=} Z \times (Z \to A)$ is applicative and co-pointed. However, the compatibility law (11.10) does not hold.

Proof The product construction (Statement 11.3.6.2) shows that $F^\bullet$ is applicative: it is a product of a constant functor (Z) and a `Reader` functor ($Z \to A$). The `zip` method is:

$$\text{zip}_F\left((z_1^{:Z} \times r_1^{:Z\to A}) \times (z_2^{:Z} \times r_2^{:Z\to B})\right) \stackrel{\text{def}}{=} (z_1 \oplus z_2) \times (z^{:Z} \to r_1(z) \times r_2(z)) \quad .$$

The functor $F^\bullet$ is co-pointed because it has a fully parametric `extract` method defined by:

$$\text{ex}_F \stackrel{\text{def}}{=} z^{:Z} \times r^{:Z\to A} \to r(z) \quad .$$

Let us see whether the compatibility law (11.10) holds:

$$\begin{aligned} \text{left-hand side}: \quad & \text{ex}_F\left(\text{zip}_F\left((z_1 \times r_1) \times (z_2 \times r_2)\right)\right) = \text{ex}_F\left((z_1 \oplus z_2) \times (z^{:Z} \to r_1(z) \times r_2(z))\right) \\ & = r_1(z_1 \oplus z_2) \times r_2(z_1 \oplus z_2) \quad , \\ \text{right-hand side}: \quad & \text{ex}_F(z_1 \times r_1) \times \text{ex}_F(z_2 \times r_2) = r_1(z_1) \times r_2(z_2) \quad . \end{aligned}$$

The two sides are not equal (when $Z \neq \mathbb{1}$). So, the compatibility law does not hold. □

We are now ready to prove the co-pointed co-product construction:

Statement 11.3.6.7 The functor $L^A \stackrel{\text{def}}{=} H^A + F^A$ is applicative if $F^\bullet$ and $H^\bullet$ are applicative and in addition $H^\bullet$ is co-pointed with a method $\text{ex}_H : H^A \to A$ that satisfies the compatibility law (11.10).

The applicative methods of $L^\bullet$ are defined by:

$$\mathrm{zip}_L : (H^A + F^A) \times (H^B + F^B) \to H^{A\times B} + F^{A\times B} \quad ,$$

$$\mathrm{zip}_L \stackrel{\text{def}}{=} \left| \begin{array}{c|c|c} & H^{A\times B} & F^{A\times B} \\ \hline H^A \times H^B & \mathrm{zip}_H & \mathbb{0} \\ H^A \times F^B & \mathbb{0} & ((\mathrm{ex}_H \fatsemi \mathrm{pu}_F) \boxtimes \mathrm{id}) \fatsemi \mathrm{zip}_F \\ F^A \times H^B & \mathbb{0} & (\mathrm{id} \boxtimes (\mathrm{ex}_H \fatsemi \mathrm{pu}_F)) \fatsemi \mathrm{zip}_F \\ F^A \times F^B & \mathbb{0} & \mathrm{zip}_F \end{array} \right| \quad .$$

The method $\mathrm{wu}_L : H^{\mathbb{1}} + F^{\mathbb{1}}$ is defined by $\mathrm{wu}_L \stackrel{\text{def}}{=} \mathrm{wu}_H + \mathbb{0}$. If $F^\bullet$ and $H^\bullet$ are commutative applicative functors then $L^\bullet$ is also commutative.

Proof The lifting to $L^\bullet$ is defined by:

$$(f^{:A\to B})^{\uparrow L} \stackrel{\text{def}}{=} \left| \begin{array}{c|c|c} & H^B & F^B \\ \hline H^A & f^{\uparrow H} & \mathbb{0} \\ F^A & \mathbb{0} & f^{\uparrow F} \end{array} \right| \quad .$$

To verify the left identity law, we begin with its left-hand side:

$$\mathrm{zip}_L(\mathrm{wu}_L \times p) = \left((\mathrm{wu}_H^{:H^{\mathbb{1}}} + \mathbb{0}^{:F^{\mathbb{1}}}) \times p^{:H^B+F^B}\right) \triangleright \left| \begin{array}{c|c|c} & H^{\mathbb{1}\times B} & F^{\mathbb{1}\times B} \\ \hline H^{\mathbb{1}} \times H^B & \mathrm{zip}_H & \mathbb{0} \\ H^{\mathbb{1}} \times F^B & \mathbb{0} & ((\mathrm{ex}_H \fatsemi \mathrm{pu}_F) \boxtimes \mathrm{id}) \fatsemi \mathrm{zip}_F \\ F^{\mathbb{1}} \times H^B & \mathbb{0} & (\mathrm{id} \boxtimes (\mathrm{ex}_H \fatsemi \mathrm{pu}_F)) \fatsemi \mathrm{zip}_F \\ F^{\mathbb{1}} \times F^B & \mathbb{0} & \mathrm{zip}_F \end{array} \right|$$

$$= p \triangleright \left| \begin{array}{c|c|c} & H^{\mathbb{1}\times B} & F^{\mathbb{1}\times B} \\ \hline H^B & h \to \mathrm{zip}_H(\mathrm{wu}_H \times h) & \mathbb{0} \\ F^B & \mathbb{0} & f \to \mathrm{zip}_F((\mathrm{wu}_H \triangleright \mathrm{ex}_H \triangleright \mathrm{pu}_F) \times f) \end{array} \right| \quad .$$

Using Eq. (8.12) and the definition of `wu` through `pure`, we find:

$$\mathrm{wu}_H \triangleright \mathrm{ex}_H \triangleright \mathrm{pu}_F = 1 \triangleright \mathrm{pu}_F = \mathrm{wu}_F \quad .$$

Since the identity laws of F and H are assumed to hold, we can transform the last matrix as:

$$\left| \begin{array}{c|c|c} & H^{\mathbb{1}\times B} & F^{\mathbb{1}\times B} \\ \hline H^B & h \to \mathrm{zip}_H(\mathrm{wu}_H \times h) & \mathbb{0} \\ F^B & \mathbb{0} & f \to \mathrm{zip}_F((\mathrm{wu}_H \triangleright \mathrm{ex}_H \triangleright \mathrm{pu}_F) \times f) \end{array} \right| = \left| \begin{array}{c|c|c} & H^{\mathbb{1}\times B} & F^{\mathbb{1}\times B} \\ \hline H^B & \mathrm{ilu}^{\uparrow H} & \mathbb{0} \\ F^B & \mathbb{0} & \mathrm{ilu}^{\uparrow F} \end{array} \right| = \mathrm{ilu}^{\uparrow L} \quad .$$

After this simplification, the left-hand side equals $p \triangleright \mathrm{ilu}^{\uparrow L}$, i.e., the right-hand side of the law.

The right identity law is verified in a similar way:

$$\mathrm{zip}_L(p \times \mathrm{wu}_L) = \mathrm{zip}_L\left(p^{:H^A+F^A} \times (\mathrm{wu}_H^{:H^{\mathbb{1}}} + \mathbb{0}^{:F^{\mathbb{1}}})\right)$$

$$= p \triangleright \left| \begin{array}{c|c|c} & H^{A\times\mathbb{1}} & F^{A\times\mathbb{1}} \\ \hline H^A & h \to \mathrm{zip}_H(h \times \mathrm{wu}_H) & \mathbb{0} \\ F^A & \mathbb{0} & f \to \mathrm{zip}_F(f \times (\mathrm{wu}_H \triangleright \mathrm{ex}_H \triangleright \mathrm{pu}_F)) \end{array} \right|$$

$$= p \triangleright \left| \begin{array}{c|cc} & H^{A\times\mathbb{1}} & F^{A\times\mathbb{1}} \\ \hline H^A & \text{iru}^{\uparrow H} & \mathbb{0} \\ F^A & \mathbb{0} & \text{iru}^{\uparrow F} \end{array} \right| = p \triangleright \text{iru}^{\uparrow L} \quad .$$

The associativity law is an equation between values of type $H^{A\times B\times C} + F^{A\times B\times C}$:

$$\text{zip}_L(p^{:H^A+F^A} \times \text{zip}_L(q^{:H^B+F^B} \times r^{:H^C+F^C})) \triangleright \varepsilon^{\uparrow L}_{1,23} = \text{zip}_L(\text{zip}_L(p\times q)\times r) \triangleright \varepsilon^{\uparrow L}_{12,3} \quad .$$

The operation $\text{zip}_L(p\times q)$ is defined in such a way that it returns a value of type $H^{A\times B}+\mathbb{0}$ only when both p and q are in the left part of the disjunction:

$$\text{zip}_L\big((a^{:H^A}+\mathbb{0}^{:F^A})\times(b^{:H^B}+\mathbb{0}^{:F^B})\big) = \text{zip}_H(a\times b)+\mathbb{0}^{:F^{A\times B}} \quad .$$

Otherwise, $\text{zip}_L(p\times q)$ returns a value of type $\mathbb{0}^{:H^{A\times B}}+F^{A\times B}$. So, we need to consider three cases:

(1) The arguments are $p = a^{:H^A}+\mathbb{0}$, $q = b^{:H^B}+\mathbb{0}$, $r = c^{:H^C}+\mathbb{0}$. In this case, zip_L reduces to zip_H:

$$\text{left-hand side}: \quad \text{zip}_L(p\times\text{zip}_L(q\times r)) \triangleright \varepsilon^{\uparrow L}_{1,23} = \text{zip}_H\big(a\times\text{zip}_H(b\times c)\big) \triangleright \varepsilon^{\uparrow H}_{1,23}+\mathbb{0} \quad ,$$
$$\text{right-hand side}: \quad \text{zip}_L(\text{zip}_L(p\times q)\times r) \triangleright \varepsilon^{\uparrow L}_{12,3} = \text{zip}_H\big(\text{zip}_H(a\times b)\times c\big) \triangleright \varepsilon^{\uparrow H}_{12,3}+\mathbb{0} \quad .$$

The two sides are equal due to the associativity law of zip_H.

(2) The argument q has type $\mathbb{0}+F^B$. In this case, zip_L reduces to zip_F after converting arguments of type $H^\bullet+0$ to type $F^\bullet$ when needed. We may define this conversion as a helper function `toF`:

$$\text{toF} : H^A+F^A \to F^A \quad , \qquad \text{toF} \stackrel{\text{def}}{=} \left| \begin{array}{c|c} & F^A \\ \hline H^A & \text{ex}_H \,\text{⨾}\, \text{pu}_F \\ F^A & \text{id} \end{array} \right| \quad .$$

If q has type $\mathbb{0}+F^B$ then we have:

$$\text{zip}_L(p\times q) = \mathbb{0}+\text{zip}_F(\text{toF}(p)\times\text{toF}(q)) \quad , \qquad \text{zip}_L(q\times r) = \mathbb{0}+\text{zip}_F(\text{toF}(q)\times\text{toF}(r)) \quad .$$

Now the associativity law of zip_L is reduced to the same law of zip_F:

$$\text{left-hand side}: \quad \text{zip}_L(p\times\text{zip}_L(q\times r)) \triangleright \varepsilon^{\uparrow L}_{1,23} = \mathbb{0}+\text{zip}_F\big(\text{toF}(p)\times\text{zip}_F(\text{toF}(q)\times\text{toF}(r))\big) \triangleright \varepsilon^{\uparrow F}_{1,23} \quad ,$$
$$\text{right-hand side}: \quad \text{zip}_L(\text{zip}_L(p\times q)\times r) \triangleright \varepsilon^{\uparrow L}_{12,3} = \mathbb{0}+\text{zip}_F(\text{zip}_F(\text{toF}(p)\times\text{toF}(q))\times\text{toF}(r)) \triangleright \varepsilon^{\uparrow F}_{12,3} \quad .$$

The two sides are equal due to the associativity law of zip_F.

(3) Either $p = \mathbb{0}+a^{:F^A}$ while $q = b^{:H^B}+\mathbb{0}$ and $r = c^{:H^C}+\mathbb{0}$; or $r = \mathbb{0}+c^{:F^C}$ while $p = a^{:H^A}+\mathbb{0}$ and $q = b^{:H^B}+\mathbb{0}$. The two situations are symmetric, so let us consider the first one:

$$\text{left-hand side}: \quad \text{zip}_L(p\times\text{zip}_L(q\times r)) \triangleright \varepsilon^{\uparrow L}_{1,23} = \mathbb{0}+\text{zip}_F\big(\text{toF}(p)\times\text{toF}(\text{zip}_H(b\times c)+\mathbb{0})\big) \triangleright \varepsilon^{\uparrow F}_{1,23} \quad .$$

Simplify the sub-expressions involving `toF` separately:

$$\text{toF}(p) = \text{toF}(\mathbb{0}+a) = a \quad ,$$
$$\text{toF}(\text{zip}_H(b\times c)+\mathbb{0}) = \text{pu}_F(\underline{\text{ex}_H(\text{zip}_H}(b\times c))$$
$$\text{use Eq. (11.10)}: \qquad = \text{pu}_F(\text{ex}_H(b)\times\text{ex}_H(c)) \quad .$$

So, we can rewrite the left-hand side as:

$$\begin{aligned}
\text{left-hand side}: \quad & \mathrm{zip}_L(p \times \mathrm{zip}_L(q \times r)) \triangleright \varepsilon^{\uparrow L}_{1,23} = \mathbb{0} + \underline{\mathrm{zip}_F}\big(a \times \underline{\mathrm{pu}_F}(\mathrm{ex}_H(b) \times \mathrm{ex}_H(c))\big) \triangleright \varepsilon^{\uparrow F}_{1,23} \\
\text{identity law of } \mathrm{zip}_F: \quad & = \mathbb{0} + a \triangleright \big(k^{:A} \to k \times (\mathrm{ex}_H(b) \times \mathrm{ex}_H(c))\big)^{\uparrow F} \triangleright \varepsilon^{\uparrow F}_{1,23} \\
& = \mathbb{0} + a \triangleright \big(k^{:A} \to k \times \mathrm{ex}_H(b) \times \mathrm{ex}_H(c)\big)^{\uparrow F} \quad .
\end{aligned}$$

The right-hand side can be transformed by using `toF` on all arguments:

$$\begin{aligned}
\text{right-hand side}: \quad & \mathrm{zip}_L(\mathrm{zip}_L(p \times q) \times r) \triangleright \varepsilon^{\uparrow L}_{12,3} = \mathbb{0} + \mathrm{zip}_F(\mathrm{zip}_F(\mathrm{toF}(p) \times \mathrm{toF}(q)) \times \mathrm{toF}(r)) \triangleright \varepsilon^{\uparrow F}_{12,3} \\
& = \mathbb{0} + \mathrm{zip}_F(\mathrm{zip}_F(a \times \mathrm{toF}(b + \mathbb{0})) \times \mathrm{toF}(r + \mathbb{0})) \triangleright \varepsilon^{\uparrow F}_{12,3} \quad .
\end{aligned}$$

Simplify the sub-expressions of the form $\mathrm{zip}_F(a \times \mathrm{toF}(b + \mathbb{0}))$:

$$\mathrm{zip}_F\big(a^{:F^A} \times \mathrm{toF}(b^{:H^B} + \mathbb{0})\big) = \mathrm{zip}_F(a \times \mathrm{pu}_F(\mathrm{ex}_H(b)) = a \triangleright (k^{:A} \to k \times \mathrm{ex}_H(b))^{\uparrow F} \quad . \tag{11.11}$$

Using this formula, we continue to transform the right-hand side:

$$\begin{aligned}
\text{right-hand side}: \quad & \mathbb{0} + \underline{\mathrm{zip}_F}(\mathrm{zip}_F(a \times \mathrm{toF}(b + \mathbb{0})) \times \underline{\mathrm{toF}(c + \mathbb{0})}) \triangleright \varepsilon^{\uparrow F}_{12,3} \\
\text{use Eq. (11.11)}: \quad & = \mathbb{0} + \underline{\mathrm{zip}_F}(a \times \underline{\mathrm{toF}(b + \mathbb{0})}) \triangleright \big(k \to k \times \mathrm{ex}_H(c)\big)^{\uparrow F} \triangleright \varepsilon^{\uparrow F}_{12,3} \\
\text{use Eq. (11.11)}: \quad & = \mathbb{0} + a \triangleright \big(k \to k \times \mathrm{ex}_H(b)\big)^{\uparrow F} \triangleright \big(k \to k \times \mathrm{ex}_H(c)\big)^{\uparrow F} \triangleright \varepsilon^{\uparrow F}_{12,3} \\
\text{compute composition}: \quad & = \mathbb{0} + a \triangleright \big(k \to k \times \mathrm{ex}_H(b) \times \mathrm{ex}_H(c)\big)^{\uparrow F} \quad .
\end{aligned}$$

The two sides are now equal.

It remains to verify the commutativity law in case that law holds for F and H:

$$\mathrm{swap} ⨾ \mathrm{zip}_F \overset{!}{=} \mathrm{zip}_F ⨾ \mathrm{swap}^{\uparrow F} \quad , \qquad \mathrm{swap} ⨾ \mathrm{zip}_H \overset{!}{=} \mathrm{zip}_H ⨾ \mathrm{swap}^{\uparrow H} \quad .$$

Begin with the left-hand side of the commutativity law for zip_L:

$$\mathrm{swap} ⨾ \mathrm{zip}_L = \left| \begin{array}{c|cc}
 & H^{B\times A} & F^{B\times A} \\
\hline
H^A \times H^B & \mathrm{swap} ⨾ \mathrm{zip}_H & \mathbb{0} \\
F^A \times H^B & \mathbb{0} & \mathrm{swap} ⨾ ((\mathrm{ex}_H ⨾ \mathrm{pu}_F) \boxtimes \mathrm{id}) ⨾ \mathrm{zip}_F \\
H^A \times F^B & \mathbb{0} & \mathrm{swap} ⨾ (\mathrm{id} \boxtimes (\mathrm{ex}_H ⨾ \mathrm{pu}_F)) ⨾ \mathrm{zip}_F \\
F^A \times F^B & \mathbb{0} & \mathrm{swap} ⨾ \mathrm{zip}_F
\end{array} \right| \quad .$$

Writing out the compositions of `swap` and the pair product functions, we get:

$$\begin{aligned}
\mathrm{swap} ⨾ ((\mathrm{ex}_H ⨾ \mathrm{pu}_F) \boxtimes \mathrm{id}) &= (\mathrm{id} \boxtimes (\mathrm{ex}_H ⨾ \mathrm{pu}_F)) ⨾ \mathrm{swap} \quad , \\
\mathrm{swap} ⨾ (\mathrm{id} \boxtimes (\mathrm{ex}_H ⨾ \mathrm{pu}_F)) &= (\mathrm{pu}_F \boxtimes \mathrm{id}) ⨾ \mathrm{swap} \quad .
\end{aligned}$$

Using these simplifications, we rewrite the left-hand side as:

$$\mathrm{swap} ⨾ \mathrm{zip}_L = \left| \begin{array}{c|cc}
 & H^{B\times A} & F^{B\times A} \\
\hline
H^A \times H^B & \mathrm{swap} ⨾ \mathrm{zip}_H & \mathbb{0} \\
F^A \times H^B & \mathbb{0} & (\mathrm{id} \boxtimes (\mathrm{ex}_H ⨾ \mathrm{pu}_F)) ⨾ \mathrm{swap} ⨾ \mathrm{zip}_F \\
H^A \times F^B & \mathbb{0} & ((\mathrm{ex}_H ⨾ \mathrm{pu}_F) \boxtimes \mathrm{id}) ⨾ \mathrm{swap} ⨾ \mathrm{zip}_F \\
F^A \times F^B & \mathbb{0} & \mathrm{swap} ⨾ \mathrm{zip}_F
\end{array} \right| \quad .$$

The right-hand side is rewritten to the same code after using the laws of zip_F and zip_H:

$$\text{zip}_L \fatsemi \text{swap}^{\uparrow L} = \left\| \begin{array}{c|cc} & H^{A\times B} & F^{A\times B} \\ \hline H^A \times H^B & \text{zip}_H & \mathbb{0} \\ F^A \times H^B & \mathbb{0} & (\text{id} \boxtimes (\text{ex}_H \fatsemi \text{pu}_F)) \fatsemi \text{zip}_F \\ H^A \times F^B & \mathbb{0} & ((\text{ex}_H \fatsemi \text{pu}_F) \boxtimes \text{id}) \fatsemi \text{zip}_F \\ F^A \times F^B & \mathbb{0} & \text{zip}_F \end{array} \right\| \fatsemi \left\| \begin{array}{c|cc} & H^{B\times A} & F^{B\times A} \\ \hline H^{A\times B} & \text{swap}^{\uparrow H} & \mathbb{0} \\ F^{A\times B} & \mathbb{0} & \text{swap}^{\uparrow F} \end{array} \right\|$$

$$= \left\| \begin{array}{c|cc} & H^{B\times A} & F^{B\times A} \\ \hline A \times B & \text{swap} \fatsemi \text{zip}_H & \mathbb{0} \\ F^A \times B & \mathbb{0} & (\text{id} \boxtimes (\text{ex}_H \fatsemi \text{pu}_F)) \fatsemi \text{swap} \fatsemi \text{zip}_F \\ A \times F^B & \mathbb{0} & ((\text{ex}_H \fatsemi \text{pu}_F) \boxtimes \text{id}) \fatsemi \text{swap} \fatsemi \text{zip}_F \\ F^A \times F^B & \mathbb{0} & \text{swap} \fatsemi \text{zip}_F \end{array} \right\| \quad .$$

The two sides are now equal. □

The constructions shown so far define applicative methods for all polynomial functors.

Statement 11.3.6.8 **(a)** Any polynomial functor L^A whose fixed types are monoids can be made into an applicative functor.

(b) If P^A and Q^A are polynomial functors with monoidal fixed types and R^A is any applicative functor then $L^A \stackrel{\text{def}}{=} P^A + Q^A \times R^A$ is applicative.

(c) A recursive polynomial functor $L^\bullet$ defined by $L^A \stackrel{\text{def}}{=} S^{A,L^{F^A}}$ is applicative if $S^{\bullet,\bullet}$ and $F^\bullet$ are polynomial functors with monoidal fixed types.

(d) If all fixed types used in parts **(a)–(c)** are commutative monoids, $L^\bullet$ will be also commutative.

Proof **(a)** Any polynomial functor L^A is built (in at least one way) by combining fixed types and the type parameter A using products and co-products. By rearranging the type expression L^A, we may bring it to the following equivalent form:

$$L^A \cong Z_0 + A \times (Z_1 + A \times (\ldots \times (Z_{n-1} + A \times Z_n)\ldots)) \quad .$$

Here, the fixed types Z_i are all monoids by assumption. Statements 11.3.6.2 and 11.3.6.3 are then sufficient to make L into a lawful applicative functor.

(b) We can express P^A and Q^A equivalently as:

$$P^A = Z_1 + A \times S_1^A \quad , \qquad Q^A = Z_2 + A \times S_2^A \quad ,$$

with some fixed types Z_i and some polynomial functors S_i^A(here $i = 1, 2$). We can then express the type L^A equivalently as:

$$P^A + Q^A \times R^A = Z_1 + A \times S_1^A + (Z_2 + A \times S_2^A) \times R^A \cong (Z_1 + Z_2 \times R^A) + A \times (S_1^A + S_2^A \times R^A) \quad .$$

Since S_1^A and S_2^A are polynomial functors of smaller degree than P^A and Q^A, we may assume by induction that the property we are proving will already hold for the functor $G^A \stackrel{\text{def}}{=} S_1^A + S_2^A \times R^A$. Statements 11.3.6.2, 11.3.6.3, 11.3.6.4, and 11.3.6.7 then show that $F^A \stackrel{\text{def}}{=} Z_1 + Z_2 \times R^A$ is applicative and $F^A + A \times G^A \cong L^A$ is applicative.

(c) The polynomial functor $S^{\bullet,\bullet}$ can be expressed as

$$S^{A,R} = P_0^A + R^A \times (P_1^A + R^A \times (\ldots \times (P_{n-1}^A + R^A \times P_n^A)\ldots)) \quad ,$$

where $P_0^A, \ldots, P_n^A$ are some polynomial functors (with respect to the type parameter A) with monoidal fixed types. Let us assume for now that R^A is an applicative functor. Then it follows from part

(b) that $P^A_{n-1} + R \times P^A_n$ is applicative. In the same way, it follows that $P^A_{n-2} + R \times (P^A_{n-1} + R \times P^A_n)$ is applicative, and so on, until we show that $S^{A,R}$ is applicative.

Now we write the recursive definition $L^A \stackrel{\text{def}}{=} S^{A,L^{F^A}}$. To prove that $S^{A,L^{F^A}}$ is an applicative functor, we may use the inductive assumption that L^A is applicative when used in the recursive position (i.e., as the second argument of $S^{\bullet,\bullet}$). By Statement 11.3.6.1, L^{F^A} is applicative when used in that position. Denoting $R^A \stackrel{\text{def}}{=} L^{F^A}$, we now deduce that $S^{A,R^A} \cong L^A$ is applicative.

(d) The proofs of parts **(a)–(c)** use only Statements 11.3.6.1, 11.3.6.2, 11.3.6.3, 11.3.6.4, and 11.3.6.7. These statements guarantee that L will be commutative if all relevant fixed types are commutative monoids. □

Function types We have seen in Section 11.2.4 that the `Reader` functor ($L^A \stackrel{\text{def}}{=} R \to A$) has a `zip` operation. That `zip` operation can be derived from the monadic methods of the `Reader` monad (which is commutative). Statement 10.2.8.6 generalized the `Reader` monad to a wider class of monads with type $L^A \stackrel{\text{def}}{=} H^A \to A$, where $H^\bullet$ is an arbitrary contrafunctor. The lawful monad gives up to two definitions of a `zip` method for the functors L of this type. However, commutativity is not guaranteed for arbitrary $H^\bullet$. One can implement a `zip` method with type signature

$$\text{zip} : (H^A \to A) \times (H^B \to B) \to H^{A\times B} \to A \times B \quad ,$$

such that the commutavity law (11.8) always holds. That definition of `zip` will, however, fail the associativity law (we omit the proof for brevity). So, there are no new constructions of lawful applicative functors of function type.

Recursive types There exist two different constructions of applicative functors based on recursive types. These constructions generalize the tree-like "free monad" (see Statement 10.2.8.10) and the standard `zip` implementation for the `List` functor.

The first construction generalizes tree-like functors. A simple example is the binary tree:

$$T^A \stackrel{\text{def}}{=} A + T^A \times T^A \quad .$$

This data structure describes trees with leaves of type A and branches that contain two sub-trees. We could generalize this type by adding extra data of type A for each branch:

$$T^A \stackrel{\text{def}}{=} A + A \times T^A \times T^A \quad .$$

The following statement generalizes this type further by using two arbitrary applicative functors to describe the branch shapes and any extra data on branches.

Statement 11.3.6.9 For any applicative functors F and H, the functor L defined recursively by $L^A \stackrel{\text{def}}{=} A + H^A \times F^{L^A}$ is applicative. If F and H are commutative then L is also commutative.

Proof It turns out that we can verify the required laws without long derivations by using type constructions whose validity we already proved. Let us define $N^A \stackrel{\text{def}}{=} H^A \times F^{L^A}$ for brevity. Then the definition of L is equivalently written as $L^A = A + N^A = A + H^A \times F^{L^A}$. This type is built up from applicative functors H, F, and (recursively) L itself. The functor $A + N^A$ is applicative due to Statement 11.3.6.4, while $N^A = H^A \times F^{L^A}$ is a functor product (Statement 11.3.6.2). Finally, F^{L^A} is applicative because it is a functor composition (Statement 11.3.6.1), and because we may assume that L^A is already a lawful applicative functor when we use its methods in recursive calls.

Since all these constructions preserve commutative applicative functors, we conclude that L will be commutative if F and H are.

The constructions also give us the code for the applicative methods `zip` and `wu` of L. This code is similar to the code for a tree-like applicative shown in Example 11.1.1.2. When combining a "leaf" value ($A + \mathbb{0}$) with a "sub-tree" value ($\mathbb{0} + N^A$), we duplicate the "leaf" value as many times as needed

to cover the "sub-tree":

$$\text{zip}_L : (A + N^A) \times (B + N^B) \to A \times B + N^{A\times B} \quad ,$$

$$\text{zip}_L \stackrel{\text{def}}{=} \left| \begin{array}{c||cc} & A\times B & N^{A\times B} \\ \hline A\times B & \text{id} & \mathbb{0} \\ N^A \times B & \mathbb{0} & n\times b \to n \triangleright (a^{:A} \to a\times b)^{\uparrow N} \\ A\times N^B & \mathbb{0} & a\times n \to n \triangleright (b^{:B} \to a\times b)^{\uparrow N} \\ N^A \times N^B & \mathbb{0} & \overline{\text{zip}_N} \end{array} \right| \quad .$$

The "wrapped unit" is $\text{wu}_L \stackrel{\text{def}}{=} 1 + \mathbb{0}^{:N^{\mathbb{1}}}$. The liftings to L and N are defined by:

$$(f^{:A\to B})^{\uparrow L} = \left| \begin{array}{c||cc} & B & N^B \\ \hline A & \text{id} & \mathbb{0} \\ N^A & \mathbb{0} & f^{\uparrow N} \end{array} \right| \quad , \qquad f^{\uparrow N} = f^{\uparrow H} \times f^{\overline{\uparrow L}\uparrow F} = h^{:H^A} \times k^{:F^{L^A}} \to (h \triangleright f^{\uparrow H}) \times (k \triangleright f^{\overline{\uparrow L}\uparrow F}) \quad .$$

For brevity, we denoted by $\overline{\text{zip}_N}$ the following function:

$$\overline{\text{zip}_N} \stackrel{\text{def}}{=} (h_1^{:H^A} \times k_1^{:F^{L^A}}) \times (h_2^{:H^B} \times k_2^{:F^{L^B}}) \to \text{zip}_H(h_1 \times h_2) \times \left(\text{zip}_F(k_1 \times k_2) \triangleright \overline{\text{zip}_L}^{\uparrow F}\right) \quad .$$

The overline reminds us that $\overline{\text{zip}_N}$ uses a recursive call to $\overline{\text{zip}_L}$. □

Statement 11.3.6.9 covers tree-like functors $L^A \stackrel{\text{def}}{=} A + H^A \times F^{L^A}$ with "leaves" of type A and "branch shapes" described by a functor F. In addition, each branch may carry a value of type H^A.

When H^A is a constant functor, the resulting $L^\bullet$ is a monad (Statement 10.2.8.10). It is important that the applicative implementations of `map2` and `zip` for this tree-like monad are *not* compatible with its monad methods. To see this, it is sufficient to note that even the simplest tree-like monad (the binary tree, $L^A \stackrel{\text{def}}{=} A + L^A \times L^A$) is not commutative. The applicative functor L, however, is commutative because it is built from $H^A \stackrel{\text{def}}{=} 1$ and $F^A \stackrel{\text{def}}{=} A \times A$, which are both commutative.

The next statement generalizes the `List` functor,

$$\text{List}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times \text{List}^A \quad ,$$

by introducing *three* arbitrary applicative functors into the above formula for List^A.

Statement 11.3.6.10 **(a)** For any applicative functors F, G, and H, the functor L defined recursively by $L^A \stackrel{\text{def}}{=} G^A + H^A \times F^{L^A}$ is applicative as long as H is co-pointed and the compatibility law (11.10) holds. If F, G, and H are commutative then L is also commutative.

(b) The same properties hold for the functor P defined by $P^A \stackrel{\text{def}}{=} F^{G^A + H^A \times P^A}$ instead.

Proof **(a)** We will avoid long derivations if we show that the functor L is built via known type constructions. At the top level, L^A is the co-pointed co-product construction (Statement 11.3.6.7) with functors $G^\bullet$ and $H^\bullet \times F^{L^\bullet}$. The functor $H^\bullet \times F^{L^\bullet}$ is co-pointed because H is (Section 8.3.7). The compatibility law holds for $H^\bullet \times F^{L^\bullet}$ due to Exercise 11.5.6.11. The functor $F^{L^\bullet}$ is applicative because it is a composition of F and the recursively used L. As usual, we may assume that recursive uses of L's methods will satisfy all required laws.

The constructions give us the code of L's methods `zip` and `wu`. Define $N^A \stackrel{\text{def}}{=} H^A \times F^{L^A}$ and write:

$$\text{ex}_N \stackrel{\text{def}}{=} \pi_1 ⨾ \text{ex}_H = \left(h^{:H^A} \times k^{:F^{L^A}} \to \text{ex}_H(h)\right) \quad ,$$

$$\overline{\text{zip}_N} \stackrel{\text{def}}{=} (h_1^{:H^A} \times k_1^{:F^{L^A}}) \times (h_2^{:H^B} \times k_2^{:F^{L^B}}) \to \text{zip}_H(h_1 \times h_2) \times \left(\text{zip}_F(k_1 \times k_2) \triangleright \overline{\text{zip}_L}^{\uparrow F}\right) \quad .$$

The function `toG` converts values of type N^A to values of type G^A:

$$\text{toG} : N^A \to G^A \quad , \qquad \text{toG} \stackrel{\text{def}}{=} \text{ex}_N ⨾ \text{pu}_G \quad .$$

Then we have the following code:

$$\text{zip}_L : (G^A + N^A) \times (G^B + N^B) \to G^{A\times B} + N^{A\times B} \quad ,$$

$$\text{zip}_L \stackrel{\text{def}}{=} \left\| \begin{array}{c|c||cc|} & & G^{A\times B} & N^{A\times B} \\ \hline & G^A \times G^B & \text{zip}_G & \mathbb{0} \\ & N^A \times G^B & (\text{toG} \boxtimes \text{id}) ⨾ \text{zip}_G & \mathbb{0} \\ & G^A \times N^B & (\text{id} \boxtimes \text{toG}) ⨾ \text{zip}_G & \mathbb{0} \\ & N^A \times N^B & \mathbb{0} & \overline{\text{zip}_N} \\ \hline \end{array} \right\| \quad ,$$

$$\text{wu}_L : G^{\mathbb{1}} + H^{\mathbb{1}} \times F^{L^{\mathbb{1}}} \quad , \qquad \text{wu}_L \stackrel{\text{def}}{=} \mathbb{0}^{:G^{\mathbb{1}}} + \text{wu}_H \times \text{pu}_F(\text{wu}_L) \quad .$$

(b) Let us use the unrolling trick to compare the recursive types L and P:

$$L^A = G^A + H^A \times F^{G^A + H^A \times F^{G^A + \ldots}} \quad , \qquad P^A = F^{G^A + H^A \times F^{G^A + H^A \times F^{G^A + \ldots}}} \quad .$$

It suggests that $P^A \cong F^{L^A}$. Then the required properties hold for P due to the functor composition construction (Statement 11.3.6.1). To establish the type equivalence $P^A \cong F^{L^A}$ rigorously, we use Statement 11.3.6.11 below, where we need to set $R^\bullet \stackrel{\text{def}}{=} F^\bullet$, $S^T \stackrel{\text{def}}{=} G^A + H^A \times T$, $U \stackrel{\text{def}}{=} P^A$, and $V \stackrel{\text{def}}{=} L^A$. □

Note that the construction shown in Statement 11.3.6.10 needs to define *both* `zip` and `wu` recursively. This leads to a problem in case `wu` does not have the type of a function. A simple example when this problem occurs is with the `List` functor. The value wu_{List} is then defined as:

$$\text{List}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times \text{List}^A \quad , \qquad \text{wu}_{\text{List}} \stackrel{\text{def}}{=} \mathbb{0} + 1 \times \text{wu}_{\text{List}} \quad .$$

This value apparently represents an *infinite* list of unit values:

$$\text{wu}_L = \mathbb{0} + 1 \times (\mathbb{0} + \mathbb{1} \times (\mathbb{0} + 1 \times (\ldots))) \qquad ???$$

We need a "lazy list" (or another lazy collection) if we want to define `wu` recursively by the code shown above. However, the data structure List^A is eagerly evaluated and can only hold a finite number of values. So, we must conclude that `wu` is undefined for the functor `List`, which then cannot be considered a fully lawful applicative functor. Nevertheless, the associativity and the commutativity laws hold for the `zip` method of `List`. The absence of a well-defined value `wu` does not lead to practical disadvantages when working with finite lists.

For certain choices of F, G, and H, the functor L will be a monad. It is important that the implementation of applicative methods `pure`, `map2`, and `zip` for this functor will be, in general, incompatible with its monad methods. The `List` functor again gives an example where the monad is not commutative while the applicative functor is. So, the commutative `zip` method of `List` cannot be defined via its `map` and `flatMap` methods. Also, the `List` monad's `pure` method (returning a list with a single value) differs from the applicative `pure` method (returning an infinite list of values, as we just saw).

There could exist other recursive constructions that produce lawful applicative functors. For instance, one could assume an "applicative bifunctor" $P^{A,R}$ having a `zip` method with type signature $P^{A,R} \times P^{B,S} \to P^{A\times B, R\times S}$ and defining a functor L via the recursive type equation $L^A \stackrel{\text{def}}{=} P^{A,L^A}$. The functor $L^\bullet$ will be applicative when suitable laws are imposed on P. To find what applicative bifunctors P exist, one could continue with structural analysis (considering products $P_1^{\bullet,\bullet} \times P_2^{\bullet,\bullet}$, coproducts $P_1^{\bullet,\bullet} + P_2^{\bullet,\bullet}$, and so on). This book will not pursue this further, because it is not clear what practical use one will get out of those constructions, and because there seems to be no end to this sort of exploration. (A recursive construction for the applicative bifunctor P will involve an "applicative trifunctor", etc.)

Note that any functor defined via a recursive type equation $L^A \stackrel{\text{def}}{=} S^{A,L^\bullet}$ will be applicative if $S^{A,R^\bullet}$ produces an applicative functor whenever $R^\bullet$ is applicative. Statements 11.3.6.9–11.3.6.10 are obtained with $S^{A,R^\bullet} \stackrel{\text{def}}{=} A + H^A \times F^{R^A}$ and $S^{A,R^\bullet} \stackrel{\text{def}}{=} G^A + H^A \times F^{R^A}$. Other examples of such $S^{\bullet,\bullet}$ can be found by combining some type constructions that are known to produce applicative functors.

We conclude this section with a proof of one version of the "unrolling trick" for recursive types:

Statement 11.3.6.11 Given two functors $R^\bullet$ and $S^\bullet$, define two recursive types U and V by $U \stackrel{\text{def}}{=} R^{S^U}$ and $V \stackrel{\text{def}}{=} S^{R^V}$. The "unrolling trick" writes (non-rigorously) $U = R^{S^{R^{S^{\cdot^{\cdot^{\cdot}}}}}}$ and $V = S^{R^{S^{R^{\cdot^{\cdot^{\cdot}}}}}}$, which suggests that U and R^V are the same type. In fact, the type U is rigorously equivalent to the type R^V.

Proof We will show that $U \cong R^V$ by implementing the isomorphisms in two directions.

By definition of the type U, we must have some isomorphisms (called fix_U and unfix_U) between types U and R^{S^U}, and similarly for the type V. So, we assume that the following functions are known and satisfy the properties of isomorphisms:

$$\begin{aligned}
&\text{fix}_U : R^{S^U} \to U \quad , \quad \text{unfix}_U : U \to R^{S^U} \quad , \quad \text{fix}_U \fatsemi \text{unfix}_U = \text{id} \quad , \quad \text{unfix}_U \fatsemi \text{fix}_U = \text{id} \quad ; \\
&\text{fix}_V : S^{R^V} \to V \quad , \quad \text{unfix}_V : V \to S^{R^V} \quad , \quad \text{fix}_V \fatsemi \text{unfix}_V = \text{id} \quad , \quad \text{unfix}_V \fatsemi \text{fix}_V = \text{id} \quad .
\end{aligned}$$

We implement the isomorphism functions (called `toU` and `toRV`) recursively:

$$\begin{aligned}
&\text{toU} : R^V \to U \quad , \qquad \text{toU} \stackrel{\text{def}}{=} \text{unfix}_V^{\uparrow R} \fatsemi \overline{\text{toU}}^{\uparrow S \uparrow R} \fatsemi \text{fix}_U \quad ; \\
&\text{toRV} : U \to R^V \quad , \qquad \text{toRV} \stackrel{\text{def}}{=} \text{unfix}_U \fatsemi \overline{\text{toRV}}^{\uparrow S \uparrow R} \fatsemi \text{fix}_V^{\uparrow R} \quad .
\end{aligned}$$

To verify the isomorphism properties ($\text{toU} \fatsemi \text{toRV} = \text{id}$ and $\text{toRV} \fatsemi \text{toU} = \text{id}$), we use the inductive assumption that those properties already hold for any recursive calls of these functions:

$$\begin{aligned}
\text{toU} \fatsemi \text{toRV} &= \text{unfix}_V^{\uparrow R} \fatsemi \overline{\text{toU}}^{\uparrow S \uparrow R} \fatsemi \underline{\text{fix}_U \fatsemi \text{unfix}_U} \fatsemi \overline{\text{toRV}}^{\uparrow S \uparrow R} \fatsemi \text{fix}_V^{\uparrow R} \\
&= \text{unfix}_V^{\uparrow R} \fatsemi \underline{\overline{\text{toU}}^{\uparrow S \uparrow R} \fatsemi \overline{\text{toRV}}^{\uparrow S \uparrow R}} \fatsemi \text{fix}_V^{\uparrow R} = \text{unfix}_V^{\uparrow R} \fatsemi \text{fix}_V^{\uparrow R} = \text{id} \quad , \\
\text{toRV} \fatsemi \text{toU} &= \text{unfix}_U \fatsemi \overline{\text{toRV}}^{\uparrow S \uparrow R} \fatsemi \underline{\text{fix}_V^{\uparrow R} \fatsemi \text{unfix}_V^{\uparrow R}} \fatsemi \overline{\text{toU}}^{\uparrow S \uparrow R} \fatsemi \text{fix}_U \\
&= \text{unfix}_U \fatsemi \underline{\overline{\text{toRV}}^{\uparrow S \uparrow R} \fatsemi \overline{\text{toU}}^{\uparrow S \uparrow R}} \fatsemi \text{fix}_U = \text{unfix}_U \fatsemi \text{fix}_U = \text{id} \quad .
\end{aligned}$$

11.4 Applicative contrafunctors and profunctors

We have seen in Example 11.1.1.3 that a `zip` method can be implemented for some type constructors that are not covariant. In this section, we will apply structural analysis systematically to discover the non-covariant type constructors (contrafunctors and profunctors) that admit a lawful `zip` method.

11.4.1 Applicative contrafunctors: Laws and constructions

Contrafunctors (see Section 6.1.7) support a `contramap` method instead of `map`. So, contrafunctors cannot have the `map2` or `ap` methods. Nevertheless, the applicative laws can be formulated via `zip` and `wu` methods using `contramap`. That is the formulation we will use for contrafunctors.

Definition 11.4.1.1 A contrafunctor $C^\bullet$ is **applicative** if there exist methods `zip` and `wu` such that:

$$\text{zip}_C : C^A \times C^B \to C^{A \times B} \quad , \qquad \text{wu}_C : C^{\mathbb{1}} \quad ,$$

$$\text{associativity law :} \quad \text{zip}_C(p \times \text{zip}_C(q \times r)) \triangleright \tilde{\varepsilon}_{1,23}^{\downarrow C} = \text{zip}_C(\text{zip}_C(p \times q) \times r) \triangleright \tilde{\varepsilon}_{12,3}^{\downarrow C} \quad , \tag{11.12}$$

$$\text{left and right identity laws :} \quad \text{zip}_C(\text{wu}_C \times p) \triangleright \text{ilu}^{\downarrow C} = p \quad , \qquad \text{zip}_C(p \times \text{wu}_C) \triangleright \text{iru}^{\downarrow C} = p \quad . \tag{11.13}$$

Here the tuple-rearranging isomorphisms $\tilde{\varepsilon}_{1,23}$, $\tilde{\varepsilon}_{12,3}$, `ilu`, and `iru` are defined by:

$$\begin{aligned}
&\tilde{\varepsilon}_{1,23} \stackrel{\text{def}}{=} a \times b \times c \to a \times (b \times c) \quad , \qquad \tilde{\varepsilon}_{12,3} \stackrel{\text{def}}{=} a \times b \times c \to (a \times b) \times c \quad , \\
&\text{ilu} \stackrel{\text{def}}{=} a \to 1 \times a \quad , \qquad \text{iru} = a \to a \times 1 \quad .
\end{aligned}$$

It is important to require the laws to hold. Otherwise, a function with the type signature of `zip` could be implemented for any contrafunctor C:

$$\text{badZip} : C^A \times C^B \to C^{A\times B} \quad , \qquad \text{badZip} \stackrel{\text{def}}{=} p^{:C^A} \times _^{:C^B} \to p \triangleright (a \times b \to a)^{\downarrow C} \quad .$$

This function loses information because it ignores one of its arguments. Intuitively, we expect that some laws will fail for this function. Indeed, the left identity law cannot hold: `badZip(wu, p)` ignores `p`, so `badZip(wu, p).contramap(ilu)` cannot be equal to `p`.

Once the `zip` and `wu` methods are known, we can define `cpure` and `cmap2`:

$$\text{cpu}_C : \forall A.\, C^A \quad , \qquad \text{cpu}_C \stackrel{\text{def}}{=} \text{wu}_C \triangleright (_^{:A} \to 1)^{\downarrow C} \quad ,$$
$$\text{cmap}_2 : (D \to A \times B) \to C^A \times C^B \to C^D \quad , \qquad \text{cmap}_2(f^{:D\to A\times B}) \stackrel{\text{def}}{=} \text{zip}_C \fatsemi f^{\downarrow C} \quad .$$

The commutativity law is formulated for applicative contrafunctors like this:

$$\text{zip}_C(q \times p) = \text{zip}_C(p \times q) \triangleright \text{swap}^{\downarrow C} \quad , \qquad \text{or equivalently :} \quad \text{swap} \fatsemi \text{zip}_C = \text{zip}_C \fatsemi \text{swap}^{\downarrow C} \quad .$$

The rest of this section proves some constructions that produce lawful applicative contrafunctors.

Type parameters A constant contrafunctor $C^A \stackrel{\text{def}}{=} Z$ (where Z is a fixed type) is at the same time a constant functor. We already showed that a constant functor is applicative when Z is a monoid.

Another construction for applicative contrafunctors is the composition with functors:

Statement 11.4.1.2 If $F^\bullet$ is an applicative *functor* and $G^\bullet$ is an applicative contrafunctor then the contrafunctor $C^A \stackrel{\text{def}}{=} F^{G^A}$ (equivalently written as $C \stackrel{\text{def}}{=} F \circ G$) is applicative.

Proof We follow the proof of Statement 11.3.6.1. The methods `zip` and `wu` are defined by:

$$\text{zip}_C : F^{G^A} \times F^{G^B} \to F^{G^{A\times B}} \quad , \qquad \text{zip}_C \stackrel{\text{def}}{=} \text{zip}_F \fatsemi \text{zip}_G^{\uparrow F} \quad ,$$
$$\text{wu}_C \stackrel{\text{def}}{=} \text{pu}_F(\text{wu}_G) \quad .$$

Note that the lifting to C is defined by $f^{\downarrow C} \stackrel{\text{def}}{=} f^{\downarrow G\uparrow F} = (f^{\downarrow G})^{\uparrow F}$.

To verify the left identity law in Eq. (11.13):

$$\begin{aligned}
\text{expect to equal } p : \quad & \text{zip}_C(\text{wu}_C \times p^{:F^{G^A}}) \triangleright \text{ilu}^{\downarrow C} = (\underline{\text{pu}_F(\text{wu}_G)} \times p) \triangleright \underline{\text{zip}_F} \fatsemi \text{zip}_G^{\uparrow F} \fatsemi \text{ilu}^{\downarrow G\uparrow F} \\
\text{left identity law of } \text{zip}_F : \quad & = p \triangleright (g^{:G^A} \to \underline{\text{wu}_G \times g)^{\uparrow F} \fatsemi \text{zip}_G^{\uparrow F} \fatsemi \text{ilu}^{\downarrow G\uparrow F}} \\
\text{composition under } {}^{\uparrow F} : \quad & = p \triangleright \big(g^{:G^A} \to \underline{\text{zip}_G(\text{wu}_G \times g) \triangleright \text{ilu}^{\downarrow G}}\big)^{\uparrow F} \\
\text{left identity law of } \text{zip}_G : \quad & = p \triangleright (\underline{g \to g})^{\uparrow F} = p \triangleright \text{id}^{\uparrow F} = p \quad .
\end{aligned}$$

To verify the right identity law:

$$\begin{aligned}
\text{expect to equal } p : \quad & \text{zip}_C(p^{:F^{G^A}} \times \text{wu}_C) \triangleright \text{iru}^{\downarrow C} = (p \times \underline{\text{pu}_F(\text{wu}_G)}) \triangleright \underline{\text{zip}_F} \fatsemi \text{zip}_G^{\uparrow F} \fatsemi \text{iru}^{\downarrow G\uparrow F} \\
\text{right identity law of } \text{zip}_F : \quad & = p \triangleright (g^{:G^A} \to \underline{g \times \text{wu}_G)^{\uparrow F} \fatsemi \text{zip}_G^{\uparrow F} \fatsemi \text{iru}^{\downarrow G\uparrow F}} \\
\text{composition under } {}^{\uparrow F} : \quad & = p \triangleright \big(g^{:G^A} \to \underline{\text{zip}_G(g \times \text{wu}_G) \triangleright \text{iru}^{\downarrow G}}\big)^{\uparrow F} \\
\text{right identity law of } \text{zip}_G : \quad & = p \triangleright (\underline{g \to g})^{\uparrow F} = p \triangleright \text{id}^{\uparrow F} = p \quad .
\end{aligned}$$

To verify the associativity law, first substitute the definition of zip_C into one side:

$$\begin{aligned}
\text{left-hand side :} \quad & \text{zip}_C\big(p \times \text{zip}_C(q \times r)\big) \triangleright \tilde{\varepsilon}_{1,23}^{\downarrow C} = \big(p \times \underline{\text{zip}_C(q \times r)}\big) \triangleright \text{zip}_F \fatsemi \text{zip}_G^{\uparrow F} \fatsemi \tilde{\varepsilon}_{1,23}^{\downarrow C} \\
& = \big(p \times \big(\text{zip}_F(q \times r) \triangleright \underline{\text{zip}_G^{\uparrow F}}\big)\big) \triangleright \underline{\text{zip}_F} \fatsemi \text{zip}_G^{\uparrow F} \fatsemi \tilde{\varepsilon}_{1,23}^{\downarrow G\uparrow F} \\
\text{naturality law of } \text{zip}_F : \quad & = \big(p \times \text{zip}_F(q \times r)\big) \triangleright \text{zip}_F \fatsemi \big(g \times k^{:G^B \times G^C} \to \underline{g \times \text{zip}_G(k)\big)^{\uparrow F} \fatsemi \text{zip}_G^{\uparrow F} \fatsemi \tilde{\varepsilon}_{1,23}^{\downarrow G\uparrow F}} \\
\text{composition under } {}^{\uparrow F} : \quad & = \text{zip}_F\big(p \times \text{zip}_F(q \times r)\big) \triangleright \big(g \times (h \times j) \to \text{zip}_G(g \times \text{zip}_G(h \times j)) \triangleright \tilde{\varepsilon}_{1,23}^{\downarrow G}\big)^{\uparrow F} \quad .
\end{aligned}$$

Rewrite the right-hand side of the associativity law in a similar way:

$$\begin{aligned}
\text{right-hand side}: \quad & \mathrm{zip}_C(\mathrm{zip}_C(p\times q)\times r)\triangleright\tilde{\varepsilon}^{\downarrow C}_{12,3} = (\underline{\mathrm{zip}_C(p\times q)}\times r)\triangleright\mathrm{zip}_F\,\text{⨾}\,\mathrm{zip}^{\uparrow F}_G\,\text{⨾}\,\tilde{\varepsilon}^{\downarrow C}_{12,3} \\
& = ((\mathrm{zip}_F(p\times q)\triangleright\underline{\mathrm{zip}^{\uparrow F}_G})\times r)\triangleright\underline{\mathrm{zip}_F}\,\text{⨾}\,\mathrm{zip}^{\uparrow F}_G\,\text{⨾}\,\tilde{\varepsilon}^{\downarrow G\uparrow F}_{12,3} \\
\text{naturality law of } \mathrm{zip}_F: \quad & = (\mathrm{zip}_F(p\times q)\times r)\triangleright\mathrm{zip}_F\,\text{⨾}\,(k^{:G^A\times G^B}\times j\to\underline{\mathrm{zip}_G(k)\times j})^{\uparrow F}\,\text{⨾}\,\mathrm{zip}^{\uparrow F}_G\,\text{⨾}\,\tilde{\varepsilon}^{\downarrow G\uparrow F}_{12,3} \\
\text{composition under } {}^{\uparrow F}: \quad & = \mathrm{zip}_F(\mathrm{zip}_F(p\times q)\times r)\triangleright((g\times h)\times j\to\mathrm{zip}_G(\mathrm{zip}_G(g\times h)\times j)\triangleright\tilde{\varepsilon}^{\downarrow G}_{12,3})^{\uparrow F} \quad .
\end{aligned}$$

The two sides become equal after using the associativity laws of zip_F and zip_G:

$$\begin{aligned}
\mathrm{zip}_F(p\times\mathrm{zip}_F(q\times r))\triangleright\varepsilon^{\uparrow F}_{1,23} &= \mathrm{zip}_F(\mathrm{zip}_F(p\times q)\times r)\triangleright\varepsilon^{\uparrow F}_{12,3} \quad , \\
\mathrm{zip}_G(g\times\mathrm{zip}_G(h\times j))\triangleright\tilde{\varepsilon}^{\downarrow G}_{1,23} &= \mathrm{zip}_G(\mathrm{zip}_G(g\times h)\times j)\triangleright\tilde{\varepsilon}^{\downarrow G}_{12,3} \quad .
\end{aligned}$$

To verify the commutativity law for C, we assume that the law holds for F and G:

$$\begin{aligned}
\text{expect to equal } (\mathrm{zip}_C\,\text{⨾}\,\mathrm{swap}^{\downarrow C}): \quad & \mathrm{swap}\,\text{⨾}\,\mathrm{zip}_C = \underline{\mathrm{swap}\,\text{⨾}\,\mathrm{zip}_F}\,\text{⨾}\,\mathrm{zip}^{\uparrow F}_G \\
\text{commutativity law of } F: \quad & = \mathrm{zip}_F\,\text{⨾}\,\underline{\mathrm{swap}^{\uparrow F}\,\text{⨾}\,\mathrm{zip}^{\uparrow F}_G} = \mathrm{zip}_F\,\text{⨾}\,(\underline{\mathrm{swap}\,\text{⨾}\,\mathrm{zip}_G})^{\uparrow F} \\
\text{commutativity law of } G \text{ under } {}^{\uparrow F}: \quad & = \mathrm{zip}_F\,\text{⨾}\,\mathrm{zip}^{\uparrow F}_G\,\text{⨾}\,\mathrm{swap}^{\downarrow G\uparrow F} = \mathrm{zip}_C\,\text{⨾}\,\mathrm{swap}^{\downarrow C} \quad .
\end{aligned}$$

□

Products This construction for applicative contrafunctors is similar to Statement 11.4.1.3:

Statement 11.4.1.3 If $F^\bullet$ and $G^\bullet$ are applicative contrafunctors then the contrafunctor $C^A \stackrel{\text{def}}{=} F^A\times G^A$ is also applicative. If F and G are commutative then C is also commutative.

Proof Exercise 11.5.6.12. □

Co-products The co-product of applicative contrafunctors is always applicative:

Statement 11.4.1.4 If $F^\bullet$ and $G^\bullet$ are applicative contrafunctors then the contrafunctor $C^A \stackrel{\text{def}}{=} F^A+G^A$ is also applicative. If F and G are commutative then C is also commutative.

Proof We need to implement the methods zip_C and wu_C. Since the type F^A+G^A is completely symmetric in F and G and the requirements for F and G are the same, there are two ways of defining the applicative methods for F^A+G^A where either F or G is in some sense "preferred".

To implement zip_C, we first transform its type signature to an equivalent type:

$$\begin{aligned}
& C^A\times C^B\to C^{A\times B}\cong(F^A+G^A)\times(F^B+G^B)\to F^{A\times B}+G^{A\times B} \\
& \cong F^A\times F^B+F^A\times G^B+G^A\times F^B+G^A\times G^B\to F^{A\times B}+G^{A\times B} \quad .
\end{aligned}$$

The code must fill a matrix of the following form:

$$\mathrm{zip}_C \stackrel{\text{def}}{=} \begin{array}{|c||cc|}
 & F^{A\times B} & G^{A\times B} \\
\hline
F^A\times F^B & \mathrm{zip}_F & \mathbb{0} \\
F^A\times G^B & ??? & ??? \\
G^A\times F^B & ??? & ??? \\
G^A\times G^B & \mathbb{0} & \mathrm{zip}_G
\end{array} \quad .$$

The line with type $F^A\times G^B\to F^{A\times B}+G^{A\times B}$ must hard-code the decision of whether to return $F^{A\times B}+\mathbb{0}$ or $\mathbb{0}+G^{A\times B}$. Choosing arbitrarily to "prefer" F over G, we decide to always return $F^{A\times B}+\mathbb{0}$ in that line. To convert F^A into $F^{A\times B}$, we use `contramap` with the projection function $\pi_1 : A\times B\to A$,

obtaining $\pi_1^{\downarrow F} : F^A \to F^{A\times B}$. The line with type $G^A \times F^B \to F^{A\times B} + G^{A\times B}$ is handled similarly:

$$\text{zip}_C \stackrel{\text{def}}{=} \left|\begin{array}{c||cc} & F^{A\times B} & G^{A\times B} \\ \hline F^A \times F^B & \text{zip}_F & \mathbb{0} \\ F^A \times G^B & p^{:F^A} \times _^{:G^B} \to p \triangleright \pi_1^{\downarrow F} & \mathbb{0} \\ G^A \times F^B & _^{:G^A} \times q^{:F^B} \to q \triangleright \pi_2^{\downarrow F} & \mathbb{0} \\ G^A \times G^B & \mathbb{0} & \text{zip}_G \end{array}\right| \quad .$$

This function will sometimes ignore its argument when that argument has type $\mathbb{0} + G^\bullet$.

Looking at the possible implementations of wu_C (of type $F^{\mathbb{1}} + G^{\mathbb{1}}$), we find two choices:

$$\text{wu}_C \stackrel{\text{def}}{=} \text{wu}_F + \mathbb{0} \quad , \qquad \text{or alternatively :} \quad \text{wu}_C \stackrel{\text{def}}{=} \mathbb{0} + \text{wu}_G \quad .$$

Let us check whether the identity laws hold with any of these choices. The left identity law is:

$$\text{zip}_C(\text{wu}_C \times p) \triangleright \text{ilu}^{\downarrow C} \stackrel{?}{=} p \quad .$$

We know that zip_C will sometimes ignore its argument of type $\mathbb{0} + G^\bullet$, and yet we need to guarantee that no information is lost from the argument p. At the same time, it is acceptable if $\text{zip}_C(\text{wu}_C \times p)$ ignored the argument wu_C. So, we need to choose $\text{wu}_C \stackrel{\text{def}}{=} \mathbb{0} + \text{wu}_G$.

With this choice, we can now verify the left identity law:

$$\text{zip}_C(\text{wu}_C \times p) \triangleright \text{ilu}^{\downarrow C} = \left((\mathbb{0}^{:F^{\mathbb{1}}} + \text{wu}_G) \times p\right) \triangleright \left|\begin{array}{c||cc} & F^{\mathbb{1}\times B} & G^{\mathbb{1}\times B} \\ \hline F^{\mathbb{1}} \times F^B & \text{zip}_F & \mathbb{0} \\ F^{\mathbb{1}} \times G^B & p \times _^{:G^B} \to p \triangleright \pi_1^{\downarrow F} & \mathbb{0} \\ G^{\mathbb{1}} \times F^B & _^{:G^{\mathbb{1}}} \times q^{:F^B} \to q \triangleright \pi_2^{\downarrow F} & \mathbb{0} \\ G^{\mathbb{1}} \times G^B & \mathbb{0} & \text{zip}_G \end{array}\right| \fatsemi \text{ilu}^{\downarrow C}$$

$$= p \triangleright \left|\begin{array}{c||cc} & F^{\mathbb{1}\times B} & G^{\mathbb{1}\times B} \\ \hline F^B & \pi_2^{\downarrow F} & \mathbb{0} \\ G^B & \mathbb{0} & g \to \text{zip}_G(\text{wu}_G \times g) \end{array}\right| \fatsemi \left|\begin{array}{c||cc} & F^B & G^B \\ \hline F^{\mathbb{1}\times B} & \text{ilu}^{\downarrow F} & \mathbb{0} \\ G^{\mathbb{1}\times B} & \mathbb{0} & \text{ilu}^{\downarrow G} \end{array}\right|$$

$$= p \triangleright \left|\begin{array}{c||cc} & F^B & G^B \\ \hline F^B & \underline{\pi_2^{\downarrow F} \fatsemi \text{ilu}^{\downarrow F}} & \mathbb{0} \\ G^B & \mathbb{0} & g \to \underline{\text{zip}_G(\text{wu}_G \times g) \triangleright \text{ilu}^{\downarrow F}} \end{array}\right| = p \triangleright \left|\begin{array}{c||cc} & F^B & G^B \\ \hline F^B & \text{id} & \mathbb{0} \\ G^B & \mathbb{0} & g \to g \end{array}\right| = p \triangleright \text{id} = p \quad .$$

The right identity law is verified in a similar way:

$$\text{zip}_C(p \times \text{wu}_C) \triangleright \text{iru}^{\downarrow C} = p \triangleright \left|\begin{array}{c||cc} & F^A & G^A \\ \hline F^A & \underline{\pi_1^{\downarrow F} \fatsemi \text{iru}^{\downarrow F}} & \mathbb{0} \\ G^A & \mathbb{0} & g \to \underline{\text{zip}_G(g \times \text{wu}_G) \triangleright \text{iru}^{\downarrow F}} \end{array}\right|$$

$$= p \triangleright \left|\begin{array}{c||cc} & F^A & G^A \\ \hline F^A & \text{id} & \mathbb{0} \\ G^A & \mathbb{0} & g \to g \end{array}\right| = p \triangleright \text{id} = p \quad .$$

If we defined $\text{wu}_C \stackrel{\text{def}}{=} \text{wu}_F + \mathbb{0}$, the identity laws would have failed to hold. It follows that the choice of "preferred" F in the code of zip_C needs to be accompanied by the definition $\text{wu}_C \stackrel{\text{def}}{=} \mathbb{0} +$

wu_G. The other choice ("prefer G") in zip_C would require us to define $\text{wu}_C \stackrel{\text{def}}{=} \text{wu}_F + \mathbb{0}$. Both definitions yield lawful `zip` and `wu` methods for the contrafunctor C and differ only by swapping F and G. In the rest of this proof, we stick to the definition that "prefers" F.

Next, we verify the associativity law (11.12) of zip_C. The expression $\text{zip}_C(p \times \text{zip}_C(q \times r))$ has eight cases depending on whether p, q, and r are in the F or in the G parts of their disjunctive types. Let us look at the conditions for the result of a `zip` operation to be of type $F^\bullet + \mathbb{0}$ or $\mathbb{0} + G^\bullet$. According to the code matrix of zip_C, the result of computing $\text{zip}_C(p \times q)$ is type $\mathbb{0} + G^\bullet$ only when both p and q are in their G parts. In that case, we find that zip_C is reduced to zip_G:

$$\text{zip}_C\left(p^{:\mathbb{0}+G^A} \times q^{:\mathbb{0}+G^B}\right) = \text{zip}_C\left((\mathbb{0} + g^{:G^A}) \times (\mathbb{0} + h^{:G^B})\right) = \mathbb{0}^{:F^{A\times B}} + \text{zip}_G(g \times h) \quad .$$

So, if all of p, q, r are of type $\mathbb{0} + G^\bullet$, the associativity law of zip_C is reduced to the associativity law of zip_G. Similarly, if all of p, q, r are of type $F^\bullet + \mathbb{0}$, the associativity law of zip_C is reduced to the associativity law of zip_F. Since the laws of zip_F and zip_G hold by assumption, we will not need to consider these two cases any further.

To verify the associativity law in the remaining cases, write $p \times \text{zip}_C(q \times r)$ separately:

$$p \times \text{zip}_C(q \times r) = (q \times r) \triangleright \left| \begin{array}{c||cc|} & C^A \times F^{B\times C} & C^A \times G^{B\times C} \\ \hline F^B \times F^C & g \times h \to p \times \text{zip}_F(g \times h) & \mathbb{0} \\ F^B \times G^C & f \times _ \to p \times (f \triangleright \pi_1^{\downarrow F}) & \mathbb{0} \\ G^B \times F^C & _ \times f \to p \times (f \triangleright \pi_2^{\downarrow F}) & \mathbb{0} \\ G^B \times G^C & \mathbb{0} & g \times h \to p \times \text{zip}_G(g \times h) \end{array} \right| \quad .$$

We can now compute $\text{zip}_C(p \times \text{zip}_C(q \times r))$, which always returns values of type $F^\bullet + \mathbb{0}$:

$$\text{zip}_C(p \times \text{zip}_C(q \times r)) = (p \times q \times r) \triangleright \left| \begin{array}{c||cc|} & F^{A\times(B\times C)} & G^{A\times(B\times C)} \\ \hline F^A \times F^B \times G^C & f \times g \times _ \to \text{zip}_F(f \times (g \triangleright \pi_1^{\downarrow F})) & \mathbb{0} \\ F^A \times G^B \times F^C & f \times _ \times h \to \text{zip}_F(f \times (h \triangleright \pi_2^{\downarrow F})) & \mathbb{0} \\ F^A \times G^B \times G^C & f \times _ \times _ \to f \triangleright \pi_1^{\downarrow F} & \mathbb{0} \\ G^A \times F^B \times F^C & _ \times g \times h \to \text{zip}_F(g \times h) \triangleright \pi_2^{\downarrow F} & \mathbb{0} \\ G^A \times F^B \times G^C & _ \times g \times _ \to g \triangleright \pi_1^{\downarrow F} \fatsemi \pi_2^{\downarrow F} & \mathbb{0} \\ G^A \times G^B \times F^C & _ \times _ \times h \to h \triangleright \pi_2^{\downarrow F} \fatsemi \pi_2^{\downarrow F} & \mathbb{0} \end{array} \right| \quad .$$

The expression $\text{zip}_C(\text{zip}_C(p \times q) \times r)$ in the right-hand side of the associativity law is written as:

$$\text{zip}_C(\text{zip}_C(p \times q) \times r) = (p \times q \times r) \triangleright \left| \begin{array}{c||cc|} & F^{(A\times B)\times C} & G^{(A\times B)\times C} \\ \hline F^A \times F^B \times G^C & f \times g \times _ \to \text{zip}_F(f \times g) \triangleright \pi_1^{\downarrow F} & \mathbb{0} \\ F^A \times G^B \times F^C & f \times _ \times h \to \text{zip}_F((f \triangleright \pi_1^{\downarrow F}) \times h) & \mathbb{0} \\ F^A \times G^B \times G^C & f \times _ \times _ \to f \triangleright \pi_1^{\downarrow F} \fatsemi \pi_1^{\downarrow F} & \mathbb{0} \\ G^A \times F^B \times F^C & _ \times g \times h \to \text{zip}_F((g \triangleright \pi_2^{\downarrow F}) \times h) & \mathbb{0} \\ G^A \times F^B \times G^C & _ \times g \times _ \to g \triangleright \pi_2^{\downarrow F} \fatsemi \pi_1^{\downarrow F} & \mathbb{0} \\ G^A \times G^B \times F^C & _ \times _ \times h \to h \triangleright \pi_2^{\downarrow F} & \mathbb{0} \end{array} \right| \quad .$$

To show that the two sides are equal, it remains to use the naturality law of zip_F:

$$\text{zip}_F\left((p \triangleright s^{\downarrow F}) \times q\right) = \text{zip}_F(p \times q) \triangleright (a \times b \to s(a) \times b)^{\downarrow F} \quad ,$$

and to apply the tuple-rearranging isomorphisms. For instance, in the row for $F^A \times G^B \times F^C$ we get:

$$\begin{aligned}
&\text{zip}_F(f \times (h \triangleright \pi_2^{\downarrow F})) \triangleright \tilde{\varepsilon}_{1,23}^{\downarrow F} = \text{zip}_F(f \times h) \triangleright (a \times (b \times c) \to a \times c)^{\downarrow F} \fatsemi (a \times b \times c \to a \times (b \times c))^{\downarrow F} \\
&\quad = \text{zip}_F(f \times h) \triangleright (a \times b \times c \to a \times c)^{\downarrow F} \quad . \\
&\text{zip}_F((f \triangleright \pi_1^{\downarrow F}) \times h) \triangleright \tilde{\varepsilon}_{12,3}^{\downarrow F} = \text{zip}_F(f \times h) \triangleright ((a \times b) \times c \to a \times c)^{\downarrow F} \fatsemi (a \times b \times c \to (a \times b) \times c)^{\downarrow F} \\
&\quad = \text{zip}_F(f \times h) \triangleright (a \times b \times c \to a \times c)^{\downarrow F} \quad .
\end{aligned}$$

In a similar way, we can show that the two sides are equal for all other rows of the matrices.

To verify the commutativity law of zip_C when that law holds for zip_F and zip_G:

$$\text{left-hand side :} \quad \text{swap} \fatsemi \text{zip}_C = \left| \begin{array}{c|c|c} & F^{B\times A} & G^{B\times A} \\ \hline F^A \times F^B & \text{swap} \fatsemi \text{zip}_F & \mathbb{0} \\ F^A \times G^B & p^{:F^A} \times _^{:G^B} \to p \triangleright \pi_2^{\downarrow F} & \mathbb{0} \\ G^A \times F^B & _^{:G^A} \times q^{:F^B} \to q \triangleright \pi_1^{\downarrow F} & \mathbb{0} \\ G^A \times G^B & \mathbb{0} & \text{swap} \fatsemi \text{zip}_G \end{array} \right| \quad ,$$

$$\text{right-hand side :} \quad \text{zip}_C \fatsemi \text{swap}^{\downarrow C} = \left| \begin{array}{c|c|c} & F^{B\times A} & G^{B\times A} \\ \hline F^A \times F^B & \text{zip}_F \fatsemi \text{swap}^{\downarrow F} & \mathbb{0} \\ F^A \times G^B & p^{:F^A} \times _^{:G^B} \to p \triangleright \pi_1^{\downarrow F} \fatsemi \text{swap}^{\downarrow F} & \mathbb{0} \\ G^A \times F^B & _^{:G^A} \times q^{:F^B} \to q \triangleright \pi_2^{\downarrow F} \fatsemi \text{swap}^{\downarrow F} & \mathbb{0} \\ G^A \times G^B & \mathbb{0} & \text{zip}_G \fatsemi \text{swap}^{\downarrow G} \end{array} \right| \quad .$$

The two sides are equal due to the commutativity laws of zip_F and zip_G, and due to the properties

$$\text{swap} \fatsemi \pi_1 = (a \times b \to b \times a) \fatsemi (c \times d \to c) = a \times b \to b = \pi_2 \quad , \qquad \text{swap} \fatsemi \pi_2 = \pi_1 \quad .$$

□

Function types The construction $P^A \stackrel{\text{def}}{=} H^A \to G^A$ for applicative contrafunctors has no analog for applicative *functors*. Exercise 11.5.6.2 shows simple examples where a function type construction fails to produce applicative functors. However, the function type construction works for a wide class of applicative contrafunctors:

Statement 11.4.1.5 If $G^\bullet$ is an applicative contrafunctor and $H^\bullet$ is *any functor* then the contrafunctor $P^A \stackrel{\text{def}}{=} H^A \to G^A$ is applicative. If $G^\bullet$ is commutative then $P^\bullet$ is also commutative.

Proof We implement the `zip` and `wu` methods like this:

$$\begin{aligned}
&\text{zip}_P : (H^A \to G^A) \times (H^B \to G^B) \to H^{A\times B} \to G^{A\times B} \quad , \\
&\text{zip}_P\left(p^{:H^A \to G^A} \times q^{:H^B \to G^B}\right) \stackrel{\text{def}}{=} h^{:H^{A\times B}} \to \text{zip}_G\left(p(h \triangleright \pi_1^{\uparrow H}) \times q(h \triangleright \pi_2^{\uparrow H})\right) \quad , \\
&\text{wu}_P \stackrel{\text{def}}{=} _^{:H^1} \to \text{wu}_G \quad .
\end{aligned}$$

We can equivalently write the definition of zip_P in a point-free form, omitting the argument h:

$$\text{zip}_P(p \times q) \stackrel{\text{def}}{=} \Delta \fatsemi (\pi_1^{\uparrow H} \boxtimes \pi_2^{\uparrow H}) \fatsemi (p \boxtimes q) \fatsemi \text{zip}_G \quad .$$

The code for lifting to P is standard:

$$f^{\downarrow P} = p^{:H^A \to G^A} \to f^{\uparrow H} \fatsemi p \fatsemi f^{\downarrow G} \quad .$$

To verify the left identity law, we need to show that

$$h^{:H^A} \triangleright \left(\text{zip}_P(\text{wu}_P \times p^{:H^A \to G^A}) \triangleright \text{ilu}^{\downarrow P}\right) \stackrel{?}{=} h \triangleright p = p(h) \quad .$$

We use the properties such as $\text{ilu} ⨾ \pi_2 = \text{id}$ and compute:

$$h \triangleright \text{ilu}^{\uparrow H} \triangleright \text{zip}_P(\text{wu}_P \times p) \triangleright \text{ilu}^{\downarrow G} = \text{zip}_G\big(\text{wu}_P(h \triangleright \text{ilu}^{\uparrow H} \triangleright \pi_1^{\uparrow H}) \times p(h \triangleright \text{ilu}^{\uparrow H} \triangleright \pi_2^{\uparrow H})\big) \triangleright \text{ilu}^{\downarrow G}$$
$$= \text{zip}_G(\text{wu}_G \times p(h)) \triangleright \text{ilu}^{\downarrow G} = p(h) \quad .$$

The right identity law is verified by a similar calculation:

$$h \triangleright \text{iru}^{\uparrow H} \triangleright \text{zip}_P(p \times \text{wu}_P) \triangleright \text{iru}^{\downarrow G} = \text{zip}_G(p(h \triangleright \text{iru}^{\uparrow H} \triangleright \pi_1^{\uparrow H}) \times \text{wu}_P(h \triangleright \text{iru}^{\uparrow H} \triangleright \pi_2^{\uparrow H})) \triangleright \text{iru}^{\downarrow G}$$
$$= \text{zip}_G(p(h) \times \text{wu}_G) \triangleright \text{iru}^{\downarrow G} = p(h) \quad .$$

To verify the associativity law, we use properties such as $\varepsilon_{12,3} ⨾ \pi_2 = \pi_3$ and so on:

$$h^{:H^{A\times B\times C}} \triangleright \tilde{\varepsilon}_{1,23}^{\uparrow H} \triangleright \text{zip}_P(p \times \text{zip}_P(q \times r)) \triangleright \tilde{\varepsilon}_{1,23}^{\downarrow G} = \text{zip}_G\big(p(h \triangleright \underline{\tilde{\varepsilon}_{1,23}^{\uparrow H} ⨾ \pi_1^{\uparrow H}}) \times \text{zip}_P(q \times r)(h \triangleright \underline{\tilde{\varepsilon}_{1,23}^{\uparrow H} ⨾ \pi_2^{\uparrow H}})\big) \triangleright \tilde{\varepsilon}_{1,23}^{\downarrow G}$$
$$= \text{zip}_G\big(p(h \triangleright \pi_1^{\uparrow H}) \times \text{zip}_G\big(q(h \triangleright \pi_2^{\uparrow H}) \times r(h \triangleright \pi_3^{\uparrow H})\big)\big) \triangleright \tilde{\varepsilon}_{1,23}^{\downarrow G} \quad ,$$
$$h^{:H^{A\times B\times C}} \triangleright \tilde{\varepsilon}_{12,3}^{\uparrow H} \triangleright \text{zip}_P(\text{zip}_P(p \times q) \times r) \triangleright \tilde{\varepsilon}_{12,3}^{\downarrow G} = \text{zip}_G\big(\text{zip}_P(p \times q)(h \triangleright \underline{\tilde{\varepsilon}_{12,3}^{\uparrow H} ⨾ \pi_1^{\uparrow H}}) \times r(h \triangleright \underline{\tilde{\varepsilon}_{12,3}^{\uparrow H} ⨾ \pi_2^{\uparrow H}})\big) \triangleright \tilde{\varepsilon}_{12,3}^{\downarrow G}$$
$$= \text{zip}_G\big(\text{zip}_G\big(p(h \triangleright \pi_1^{\uparrow H}) \times q(h \triangleright \pi_2^{\uparrow H})\big) \times r(h \triangleright \pi_3^{\uparrow H})\big) \triangleright \tilde{\varepsilon}_{12,3}^{\downarrow G} \quad .$$

The two sides are now equal due to the assumed associativity law of zip_G.

It remains to verify the commutativity law, assuming that zip_G satisfies that law:

$$\text{zip}_P(q \times p) = \Delta ⨾ (\pi_1^{\uparrow H} \boxtimes \pi_2^{\uparrow H}) ⨾ (q \boxtimes p) ⨾ \text{zip}_G \quad ,$$
$$\text{zip}_P(p \times q) \triangleright \text{swap}^{\downarrow P} = \text{swap}^{\uparrow H} ⨾ \Delta ⨾ (\pi_1^{\uparrow H} \boxtimes \pi_2^{\uparrow H}) ⨾ (p \boxtimes q) ⨾ \underline{\text{zip}_G ⨾ \text{swap}^{\downarrow G}}$$
$$= \Delta ⨾ \underline{(\text{swap}^{\uparrow H} \boxtimes \text{swap}^{\uparrow H}) ⨾ (\pi_1^{\uparrow H} \boxtimes \pi_2^{\uparrow H})} ⨾ \underline{(p \boxtimes q) ⨾ \text{swap}} ⨾ \text{zip}_G$$
$$= \underline{\Delta ⨾ (\pi_2^{\uparrow H} \boxtimes \pi_1^{\uparrow H}) ⨾ \text{swap}} ⨾ (q \boxtimes p) ⨾ \text{zip}_G = \Delta ⨾ (\pi_1^{\uparrow H} \boxtimes \pi_2^{\uparrow H}) ⨾ (q \boxtimes p) ⨾ \text{zip}_G \quad .$$

□

The constructions shown in this section will cover all exponential-polynomial contrafunctors built up from type parameters and monoidal fixed types. It follows that *all* exponential-polynomial contrafunctors with monoidal fixed types are applicative. The applicative instance will not be unique if the contrafunctor involves a co-product, because the construction in Statement 11.4.1.4 admits two alternative implementations of the applicative methods.

11.4.2 Applicative profunctors: Laws and constructions

The word "profunctor" is used in two ways in this book (see Section 6.4.1). We call a type constructor $P^{X,Y}$ a profunctor when it is contravariant in X and covariant in Y. Given such a profunctor $P^{X,Y}$, we can set $X = Y$ and obtain a type constructor $Q^X \stackrel{\text{def}}{=} P^{X,X}$, which is neither covariant nor contravariant in X. This new type constructor ($Q^\bullet$) is also called a "profunctor".

When a given type constructor Q^A is fully parametric, we can always separate the covariant and the contravariant occurrences of the type parameter A in Q^A. We can then rename the contravariant occurrences to X and the covariant ones to Y, obtaining a type constructor $P^{X,Y}$ that fits the definition of a profunctor. In this sense, all fully parametric type constructors are profunctors.

An example of a type constructor that *cannot* be a profunctor is an "unfunctor":

```
sealed trait U[A]                    // Unfunctor.
final case class U1(s: Double) extends U[Unit]
final case class U2(b: String) extends U[Int]
final case class U3(b: String) extends U[Long]
```

$$U^A \stackrel{\text{def}}{=} \text{Double}^{:U^{\mathbb{1}}} + \text{String}^{:U^{\text{Int}}} + \text{String}^{:U^{\text{Long}}} \quad .$$

The definition of this type constructor is *not* fully parametric: values of type U^A cannot be created for arbitrary type parameter A (only for $A = \text{Int}$ or $A = \text{Long}$ or $A = \mathbb{1}$). This prevents us from implementing any `map`-like methods for $U^\bullet$.

Profunctors have an `xmap` method instead of a `map` method:

```
def xmap[A, B](f: A => B, g: B => A): P[A] => P[B]
```

For brevity, we will denote Scala code such as `xmap(f, g)(p)` or `p.xmap(f, g)` by:

$$p \triangleright f^{\uparrow P} g^{\downarrow P} \quad , \quad \text{or equivalently}: \quad p \triangleright g^{\downarrow P} f^{\uparrow P} \quad .$$

The applicative laws for profunctors are formulated via `zip` and `wu` methods using `xmap`:

Definition 11.4.2.1 A profunctor $P^\bullet$ is **applicative** if there exist methods `zip` and `wu` such that:

$$\text{zip}_P : P^A \times P^B \to P^{A\times B} \quad , \quad \text{zip}_P(p \times \text{zip}_P(q \times r)) \cong \text{zip}_P(\text{zip}_P(p \times q) \times r) \quad ,$$
$$\text{wu}_P : P^{\mathbb{1}} \quad , \quad \text{zip}_P(\text{wu}_P \times p) \cong p \quad , \quad \text{zip}_P(p \times \text{wu}_P) \cong p \quad .$$

Here we imply tuple-rearranging isomorphisms $P^{(A\times B)\times C} \cong P^{A\times(B\times C)}$ and $P^{\mathbb{1}\times A} \cong P^A \cong P^{A\times\mathbb{1}}$, which are implemented via the functions $\varepsilon_{1,23}$, $\varepsilon_{12,3}$, $\tilde{\varepsilon}_{1,23}$, $\tilde{\varepsilon}_{12,3}$, `ilu`, and `iru`:

$$\text{zip}_P(p \times \text{zip}_P(q \times r)) \triangleright \varepsilon^{\uparrow P}_{1,23} \tilde{\varepsilon}^{\downarrow P}_{1,23} = \text{zip}_P(\text{zip}_P(p \times q) \times r) \triangleright \varepsilon^{\uparrow P}_{12,3} \tilde{\varepsilon}^{\downarrow P}_{12,3} \quad ,$$
$$\text{zip}_P(\text{wu}_P \times p) = p \triangleright \text{ilu}^{\uparrow P} \pi^{\downarrow P}_2 \quad , \quad \text{zip}_P(p \times \text{wu}_P) = p \triangleright \text{iru}^{\uparrow P} \pi^{\downarrow P}_1 \quad .$$

Once the `zip` and `wu` methods are known, we can define `pure` and `xmap2`:

$$\text{pu}_P : A \to P^A \quad , \quad \text{pu}_P(a^{:A}) \stackrel{\text{def}}{=} \text{wu}_P \triangleright (_^{:A} \to 1)^{\downarrow P} (_^{:\mathbb{1}} \to a)^{\uparrow P} \quad ,$$
$$\text{xmap}_2 : (A \times B \to D) \times (D \to A \times B) \to P^A \times P^B \to P^D \quad ,$$
$$\text{xmap}_2(f^{:A\times B\to D} \times g^{:D\to A\times B}) \stackrel{\text{def}}{=} \text{zip}_P \fatsemi (g^{\downarrow C} f^{\uparrow C}) \quad .$$

It is important that the `pure` method (pu_P) for profunctors is defined via the wrapped unit (wu_P). The presence of a value $\text{wu}_P : P^{\mathbb{1}}$ means that the profunctor $P^\bullet$ is pointed (see Section 8.3.5). For functors and contrafunctors, the naturality law of `pure` is enough to enforce the equivalence of `pure` and `wu`. For profunctors, however, the value $\text{wu}_P : P^{\mathbb{1}}$ is *not* equivalent to the type of fully parametric functions with the type signature of `pure`. The following example illustrates this (see also Exercise 11.5.6.18).

Example 11.4.2.2 For the profunctor $P^A \stackrel{\text{def}}{=} (A \to A) \to A$, show that the type $P^{\mathbb{1}}$ is *not* equivalent to the type of fully parametric functions $\text{pu}_P : A \to P^A$.

Solution We can rewrite the type of wu_P equivalently as $P^{\mathbb{1}} = (\mathbb{1} \to \mathbb{1}) \to \mathbb{1} \cong \mathbb{1}$. So, there is only one value of this type (a function that ignores its argument and always returns the unit value 1). The corresponding `pure` method is a function that ignores its argument always returns the given value:

$$\text{pu}_P \stackrel{\text{def}}{=} a^{:A} \to _^{:A\to A} \to a \quad .$$

But there are many more functions with the same type signature as pu_P. To see this, it is convenient swap the curried arguments of pu_P and obtain an equivalent type:

$$A \to P^A = A \to (A \to A) \to A \cong (A \to A) \to A \to A \quad .$$

Examples of functions of this type are

$$f_1 \stackrel{\text{def}}{=} k^{:A\to A} \to k \quad , \quad f_2 \stackrel{\text{def}}{=} k^{:A\to A} \to (k \fatsemi k) \quad ,$$

and so on. For any non-negative integer $n = 0, 1, 2, ...$, we can define the function f_n that applies its argument n times (similar functions were defined in Example 4.3.2.2 and Exercise 4.4.2.6). The function `pure` defined above is the same as f_0. All functions f_n are fully parametric. So, the type of fully parametric functions with type signature $A \to P^A$ contains many more values than the type $P^{\mathbb{1}}$. □

The commutativity law of applicative profunctors is written like this:

$$\text{zip}_P(q \times p) \triangleright \text{swap}^{\downarrow P} \text{swap}^{\uparrow P} = \text{zip}_P(p \times q) \quad .$$

The rest of this section proves some constructions that produce lawful applicative profunctors.

Statement 11.4.2.3 If $F^\bullet$ is an applicative *functor* and $G^\bullet$ is an applicative profunctor then the profunctor $P^A \stackrel{\text{def}}{=} F^{G^A}$ (equivalently written as $P \stackrel{\text{def}}{=} F \circ G$) is applicative. In addition, if G is commutative then so is P.

Proof Exercise 11.5.6.14.

Statement 11.4.2.4 If $G^\bullet$ and $H^\bullet$ are applicative profunctors then so is $P^A \stackrel{\text{def}}{=} G^A \times H^A$. If both G and H are commutative then so is P.

Proof We follow the proof of Statement 11.3.6.2. Begin by implementing the `zip` and `wu` methods for P:

$$\begin{aligned}
&\text{zip}_P : (F^A \times G^A) \times (F^B \times G^B) \to F^{A\times B} \times G^{A\times B} \quad , \\
&\text{zip}_P\big((m^{:F^A} \times n^{:G^A}) \times (p^{:F^B} \times q^{:G^B})\big) \stackrel{\text{def}}{=} \text{zip}_F(m \times p) \times \text{zip}_G(n \times q) \quad , \\
&\text{wu}_P : F^{\mathbb{1}} \times G^{\mathbb{1}} \quad , \qquad \text{wu}_P \stackrel{\text{def}}{=} \text{wu}_F \times \text{wu}_G \quad .
\end{aligned}$$

The lifting to P is defined by $(f^{\uparrow P} g^{\downarrow P}) \stackrel{\text{def}}{=} (f^{\uparrow F} g^{\downarrow F}) \boxtimes (f^{\uparrow G} g^{\downarrow G})$.

To verify the left identity law of zip_P:

$$\begin{aligned}
\text{expect to equal } (p \times q) \triangleright \text{ilu}^{\uparrow P} \pi_2^{\downarrow P} : \quad & \text{zip}_P\big(\underline{\text{wu}_L} \times (p^{:F^A} \times q^{:G^A})\big) = \underline{\text{zip}_P}((\text{wu}_F \times \text{wu}_G) \times (p \times q)) \\
\text{definition of zip}_P : \quad & = \text{zip}_F(\text{wu}_F \times p) \times \text{zip}_G(\text{wu}_G \times q) \\
\text{left identity laws of zip}_F \text{ and zip}_G : \quad & = (p \triangleright \text{ilu}^{\uparrow F} \pi_2^{\downarrow F}) \times (q \triangleright \text{ilu}^{\uparrow G} \pi_2^{\downarrow G}) \\
\text{definition of } \boxtimes : \quad & = (p \times q) \triangleright \big((\text{ilu}^{\uparrow F} \pi_2^{\downarrow F}) \boxtimes (\text{ilu}^{\uparrow G} \pi_2^{\downarrow G})\big) = (p \times q) \triangleright \text{ilu}^{\uparrow P} \pi_2^{\downarrow P} \quad .
\end{aligned}$$

To verify the right identity law of zip_P:

$$\begin{aligned}
\text{expect to equal } (p \times q) \triangleright \text{iru}^{\uparrow P} \pi_1^{\downarrow P} : \quad & \text{zip}_P\big((p^{:F^A} \times q^{:G^A}) \times \underline{\text{wu}_L}\big) = \underline{\text{zip}_P}((p \times q) \times (\text{wu}_F \times \text{wu}_G)) \\
\text{definition of zip}_P : \quad & = \text{zip}_F(p \times \text{wu}_F) \times \text{zip}_G(q \times \text{wu}_G) \\
\text{right identity laws of zip}_F \text{ and zip}_G : \quad & = (p \triangleright \text{iru}^{\uparrow F} \pi_1^{\downarrow F}) \times (q \triangleright \text{iru}^{\uparrow G} \pi_1^{\downarrow G}) \\
\text{definition of } \boxtimes : \quad & = (p \times q) \triangleright \big((\text{iru}^{\uparrow F} \pi_1^{\downarrow F}) \boxtimes (\text{iru}^{\uparrow G} \pi_1^{\downarrow G})\big) = (p \times q) \triangleright \text{iru}^{\uparrow P} \pi_1^{\downarrow P} \quad .
\end{aligned}$$

To verify the associativity law, begin with its left-hand side and use the definition of zip_P:

$$\begin{aligned}
&\text{zip}_P\big((p_1^{:F^A} \times p_2^{:G^A}) \times \text{zip}_P\big((q_1^{:F^B} \times q_2^{:G^B}) \times (r_1^{:F^C} \times r_2^{:G^C})\big)\big) \triangleright \varepsilon_{1,23}^{\uparrow P} \tilde{\varepsilon}_{1,23}^{\downarrow P} \\
&= \text{zip}_P\big((p_1 \times p_2) \times (\text{zip}_F(q_1 \times r_1) \times \text{zip}_G(q_2 \times r_2))\big) \triangleright \big((\varepsilon_{1,23}^{\uparrow F} \tilde{\varepsilon}_{1,23}^{\downarrow F}) \boxtimes (\varepsilon_{1,23}^{\uparrow G} \tilde{\varepsilon}_{1,23}^{\downarrow G})\big) \\
&= \big(\underline{\text{zip}_F(p_1 \times \text{zip}_F(q_1 \times r_1)) \triangleright \varepsilon_{1,23}^{\uparrow F} \tilde{\varepsilon}_{1,23}^{\downarrow F}}\big) \times \big(\underline{\text{zip}_G(p_2 \times \text{zip}_G(q_2 \times r_2)) \triangleright \varepsilon_{1,23}^{\uparrow G} \tilde{\varepsilon}_{1,23}^{\downarrow G}}\big) \quad .
\end{aligned}$$

The right-hand side is rewritten in a similar way:

$$\begin{aligned}
&\text{zip}_P\big(\text{zip}_P\big((p_1^{:F^A} \times p_2^{:G^A}) \times (q_1^{:F^B} \times q_2^{:G^B})\big) \times (r_1^{:F^C} \times r_2^{:G^C})\big) \triangleright \varepsilon_{12,3}^{\uparrow P} \tilde{\varepsilon}_{12,3}^{\downarrow P} \\
&= \text{zip}_P\big(\big(\text{zip}_F(p_1 \times q_1) \times \text{zip}_G(p_2 \times q_2)\big) \times (r_1 \times r_2)\big)\big) \triangleright \big((\varepsilon_{12,3}^{\uparrow F} \tilde{\varepsilon}_{12,3}^{\downarrow F}) \boxtimes (\varepsilon_{12,3}^{\uparrow G} \tilde{\varepsilon}_{12,3}^{\downarrow G})\big) \\
&= \big(\underline{\text{zip}_F(\text{zip}_F(p_1 \times q_1) \times r_1) \triangleright \varepsilon_{12,3}^{\uparrow F} \tilde{\varepsilon}_{12,3}^{\downarrow F}}\big) \times \big(\underline{\text{zip}_G(\text{zip}_G(p_2 \times q_2) \times r_2) \triangleright \varepsilon_{12,3}^{\uparrow G} \tilde{\varepsilon}_{12,3}^{\downarrow G}}\big) \quad .
\end{aligned}$$

The underlined expressions in both sides are equal due to associativity laws of zip_F and zip_G.

To verify the commutativity law of P assuming it holds for F and G:

$$\begin{aligned}
\text{expect zip}_P\big((p \times q) \times (m \times n)\big) : \quad & \text{zip}_P\big((m \times n) \times (p \times q)\big) \triangleright \text{swap}^{\uparrow P} \text{swap}^{\downarrow P} \\
\text{definitions of zip}_P \text{ and } {}^{\uparrow P} : \quad & = \big(\text{zip}_F(m \times p) \times \text{zip}_G(n \times q)\big) \triangleright \big((\text{swap}^{\uparrow F} \text{swap}^{\downarrow F}) \boxtimes (\text{swap}^{\uparrow G} \text{swap}^{\downarrow G})\big) \\
\text{definition of } \boxtimes : \quad & = \big(\text{zip}_F(m \times p) \triangleright \text{swap}^{\uparrow F} \text{swap}^{\downarrow F}\big) \times \big(\text{zip}_G(n \times q) \triangleright \text{swap}^{\uparrow G} \text{swap}^{\downarrow G}\big) \\
\text{commutativity of } F \text{ and } G : \quad & = \text{zip}_F(p \times m) \times \text{zip}_G(q \times n) = \text{zip}_P\big((p \times q) \times (m \times n)\big) \quad .
\end{aligned}$$

□

Statement 11.4.2.5 If $F^\bullet$ is an applicative profunctor and Z is a fixed monoid type then the profunctor $P^A \stackrel{\text{def}}{=} Z + F^A$ is also applicative:

$$\text{zip}_P : (Z + F^A) \times (Z + F^B) \to Z + F^{A\times B} \quad , \qquad \text{zip}_P \stackrel{\text{def}}{=} \begin{array}{|c||cc|} & Z & F^{A\times B} \\ \hline Z\times Z & z_1 \times z_2 \to z_1 \oplus z_2 & \mathbb{0} \\ F^A \times Z & _^{:F^A} \times z \to z & \mathbb{0} \\ Z \times F^B & z \times _^{:F^B} \to z & \mathbb{0} \\ F^A \times F^B & \mathbb{0} & \text{zip}_F \end{array} \quad .$$

The method $\text{wu}_P : Z + F^{\mathbb{1}}$ is defined by $\text{wu}_P \stackrel{\text{def}}{=} \mathbb{0}^{:Z} + \text{wu}_F$. If Z is a commutative monoid and $F^\bullet$ is commutative then $P^\bullet$ is also commutative.

Proof We follow the proof of Statement 11.3.6.3 *mutatis mutandis*.

The lifting to P is defined in the standard way:

$$(f^{:A\to B})^{\uparrow P}(g^{:B\to A})^{\downarrow P} \stackrel{\text{def}}{=} \begin{array}{|c||cc|} & Z & F^B \\ \hline Z & \text{id} & \mathbb{0} \\ F^A & \mathbb{0} & f^{\uparrow F} g^{\downarrow F} \end{array} \quad .$$

To verify the left identity law, we use the left identity law of zip_F:

$$\text{zip}_P(\text{wu}_P \times p^{:Z+F^B}) = \text{zip}_P((\mathbb{0} + \text{wu}_F) \times p) = (\text{wu}_F \times p) \triangleright \begin{array}{|c||cc|} & Z & F^{\mathbb{1}\times B} \\ \hline F^{\mathbb{1}} \times Z & _^{:F^{\mathbb{1}}} \times z \to z & \mathbb{0} \\ F^{\mathbb{1}} \times F^B & \mathbb{0} & \text{zip}_F \end{array}$$

$$= p \triangleright \begin{array}{|c||cc|} & Z & F^{\mathbb{1}\times B} \\ \hline Z & \text{id} & \mathbb{0} \\ F^B & \mathbb{0} & k^{:F^B} \to \underline{\text{zip}_F(\text{wu}_F \times k)} \end{array} = p \triangleright \begin{array}{|c||cc|} & Z & F^{\mathbb{1}\times B} \\ \hline Z & \text{id} & \mathbb{0} \\ F^B & \mathbb{0} & k \to k \triangleright \text{ilu}^{\uparrow F} \pi_2^{\downarrow F} \end{array} = p \triangleright \text{ilu}^{\uparrow P} \pi_2^{\downarrow P} \quad .$$

To verify the right identity law, we write a similar calculation:

$$\text{zip}_P(p^{:Z+F^A} \times \text{wu}_P) = \text{zip}_P(p \times (\mathbb{0} + \text{wu}_F)) = (p \times \text{wu}_F) \triangleright \begin{array}{|c||cc|} & Z & F^{A\times\mathbb{1}} \\ \hline Z \times F^{\mathbb{1}} & z \times _^{:F^{\mathbb{1}}} \to z & \mathbb{0} \\ F^A \times F^{\mathbb{1}} & \mathbb{0} & \text{zip}_F \end{array}$$

$$= p \triangleright \begin{array}{|c||cc|} & Z & F^{A\times\mathbb{1}} \\ \hline Z & \text{id} & \mathbb{0} \\ F^A & \mathbb{0} & k^{:F^A} \to \underline{\text{zip}_F(k \times \text{wu}_F)} \end{array} = p \triangleright \begin{array}{|c||cc|} & Z & F^{A\times\mathbb{1}} \\ \hline Z & \text{id} & \mathbb{0} \\ F^A & \mathbb{0} & k \to k \triangleright \text{iru}^{\uparrow F} \pi_1^{\downarrow F} \end{array} = p \triangleright \text{iru}^{\uparrow P} \pi_1^{\downarrow P} \quad .$$

To verify the associativity law, we use a trick to avoid long derivations. The two sides of the associativity law are expressions of type $Z + F^{A\times B\times C}$:

$$\text{left-hand side}: \quad \text{zip}_P\big(p^{:Z+F^A} \times \text{zip}_P(q^{:Z+F^B} \times r^{:Z+F^C})\big) \triangleright \varepsilon_{1,23}^{\uparrow P} \tilde{\varepsilon}_{1,23}^{\downarrow P} \quad ,$$

$$\text{right-hand side}: \quad \text{zip}_P\big(\text{zip}_P(p^{:Z+F^A} \times q^{:Z+F^B}) \times r^{:Z+F^C}\big) \triangleright \varepsilon_{12,3}^{\uparrow P} \tilde{\varepsilon}_{12,3}^{\downarrow P} \quad .$$

Since each of the arguments p, q, r may be in one of the two parts of the disjunction type $Z + F^\bullet$, we have 8 cases. We note, however, that the code of $\text{zip}_P(p \times q)$ will return a value of type $Z + \mathbb{0}$

whenever at least one of the arguments (p, q) is of type $Z + \mathbb{0}$. So, a composition of two `zip` operations will also return a value of type $Z + \mathbb{0}$ whenever at least one of the arguments (p, q, r) is of type $Z + \mathbb{0}$. It remains to consider only two cases:

(1) At least one of p, q, r is of type $Z + \mathbb{0}$. In this case, any arguments of type $\mathbb{0} + F^\bullet$ are ignored by zip_P, while the arguments of type $Z + \mathbb{0}$ are combined using the monoid Z's binary operation $(\oplus)$. So, the result of the `zip` operation is the same if we replace any arguments (p, q, r) of type $\mathbb{0} + F^\bullet$ by the empty value e_Z. For example:

$$\text{zip}_P\big((z + \mathbb{0}) \times (\mathbb{0} + k^{:F^A})\big) = z + \mathbb{0} = \text{zip}_P\big((z + \mathbb{0}) \times (e_Z + \mathbb{0})\big) \quad .$$

After this replacement, we have three arguments $(z_1 + \mathbb{0},\ z_2 + \mathbb{0},\ z_3 + \mathbb{0})$ instead of p, q, r, and the function zip_P reduces to the operation $\oplus$, for which the associativity law holds by assumption.

(2) All of p, q, r are of type $\mathbb{0} + F^\bullet$. In this case, zip_P reduces to zip_F, which satisfies the associativity law by assumption.

To verify the commutativity law of P, use the code matrix for `swap` with the relevant types:

$$\text{swap} ⨾ \text{zip}_P = \begin{array}{|c||cccc|} & Z \times Z & F^B \times Z & Z \times F^A & F^B \times F^A \\ \hline Z \times Z & \text{swap} & \mathbb{0} & \mathbb{0} & \mathbb{0} \\ F^A \times Z & \mathbb{0} & \mathbb{0} & \text{swap} & \mathbb{0} \\ Z \times F^B & \mathbb{0} & \text{swap} & \mathbb{0} & \mathbb{0} \\ F^A \times F^B & \mathbb{0} & \mathbb{0} & \mathbb{0} & \text{swap} \end{array} ⨾ \begin{array}{|c||cc|} & Z & F^{B\times A} \\ \hline Z \times Z & z_1 \times z_2 \to z_1 \oplus z_2 & \mathbb{0} \\ F^B \times Z & _ \times z \to z & \mathbb{0} \\ Z \times F^A & z \times _ \to z & \mathbb{0} \\ F^B \times F^A & \mathbb{0} & \text{zip}_F \end{array}$$

$$= \begin{array}{|c||cc|} & Z & F^{B\times A} \\ \hline Z \times Z & z_1 \times z_2 \to z_2 \oplus z_1 & \mathbb{0} \\ F^A \times Z & _ \times z \to z & \mathbb{0} \\ Z \times F^B & z \times _ \to z & \mathbb{0} \\ F^A \times F^B & \mathbb{0} & \text{swap} ⨾ \text{zip}_F \end{array} \quad .$$

By assumption, $\text{swap} ⨾ \text{zip}_F = \text{zip}_F ⨾ \text{swap}^{\uparrow F} \text{swap}^{\downarrow F}$. The code for $\text{swap}^{\uparrow P} \text{swap}^{\downarrow P}$ is:

$$\text{swap}^{\uparrow P} \text{swap}^{\downarrow P} = \begin{array}{|c||cc|} & Z & F^{B\times A} \\ \hline Z & \text{id} & \mathbb{0} \\ F^{A\times B} & & \text{swap}^{\uparrow F} \text{swap}^{\downarrow F} \end{array} \quad .$$

We can now transform the right-hand side of the commutativity law:

$$\text{zip}_L ⨾ \text{swap}^{\uparrow P} \text{swap}^{\downarrow P} = \begin{array}{|c||cc|} & Z & F^{B\times A} \\ \hline Z \times Z & z_1 \times z_2 \to z_1 \oplus z_2 & \mathbb{0} \\ F^A \times Z & _ \times z \to z & \mathbb{0} \\ Z \times F^B & z \times _ \to z & \mathbb{0} \\ F^A \times F^B & \mathbb{0} & \text{zip}_F ⨾ \text{swap}^{\uparrow F} \text{swap}^{\downarrow F} \end{array} \quad .$$

The difference between the sides disappears if Z is a commutative monoid $(z_1 \oplus z_2 = z_2 \oplus z_1)$. □

Statement 11.4.2.6 If $F^\bullet$ and $H^\bullet$ are applicative profunctors and $H^\bullet$ is co-pointed with the method $\text{ex}_F : H^A \to A$ such that the compatibility law (11.10) holds, then $P^A \stackrel{\text{def}}{=} H^A + F^A$ is also applicative:

$$\text{zip}_P : (H^A + F^A) \times (H^B + F^B) \to H^{A\times B} + F^{A\times B} \quad ,$$

$$\text{zip}_P \stackrel{\text{def}}{=} \left| \begin{array}{c|cc} & H^{A\times B} & F^{A\times B} \\ \hline H^A \times H^B & \text{zip}_H & \mathbb{0} \\ H^A \times F^B & \mathbb{0} & ((\text{ex}_H \fatsemi \text{pu}_F) \boxtimes \text{id}) \fatsemi \text{zip}_F \\ F^A \times H^B & \mathbb{0} & (\text{id} \boxtimes (\text{ex}_H \fatsemi \text{pu}_F)) \fatsemi \text{zip}_F \\ F^A \times F^B & \mathbb{0} & \text{zip}_F \end{array} \right| \quad ;$$

The method $\text{wu}_P : H^{\mathbb{1}} + F^{\mathbb{1}}$ is defined by $\text{wu}_P \stackrel{\text{def}}{=} \text{wu}_H + \mathbb{0}^{:F^{\mathbb{1}}}$. If $F^\bullet$ and $H^\bullet$ are commutative then $P^\bullet$ is also commutative.

Proof We follow the proof of Statement 11.3.6.7. The lifting to $P^\bullet$ is defined by:

$$(f^{:A\to B})^{\uparrow P} (g^{:B\to A})^{\downarrow P} \stackrel{\text{def}}{=} \left| \begin{array}{c|cc} & H^B & F^B \\ \hline H^A & f^{\uparrow H} g^{\downarrow H} & \mathbb{0} \\ F^A & \mathbb{0} & f^{\uparrow F} g^{\downarrow F} \end{array} \right| \quad .$$

To verify the left identity law, we begin with its left-hand side:

$$\text{zip}_P(\text{wu}_P \times p) = \left((\text{wu}_H^{:H^{\mathbb{1}}} + \mathbb{0}^{:F^{\mathbb{1}}}) \times p^{:H^B+F^B}\right) \triangleright \left| \begin{array}{c|cc} & H^{\mathbb{1}\times B} & F^{\mathbb{1}\times B} \\ \hline H^{\mathbb{1}} \times H^B & \text{zip}_H & \mathbb{0} \\ H^{\mathbb{1}} \times F^B & \mathbb{0} & ((\text{ex}_H \fatsemi \text{pu}_F) \boxtimes \text{id}) \fatsemi \text{zip}_F \\ F^{\mathbb{1}} \times H^B & \mathbb{0} & (\text{id} \boxtimes (\text{ex}_H \fatsemi \text{pu}_F)) \fatsemi \text{zip}_F \\ F^{\mathbb{1}} \times F^B & \mathbb{0} & \text{zip}_F \end{array} \right|$$

$$= p \triangleright \left| \begin{array}{c|cc} & H^{\mathbb{1}\times B} & F^{\mathbb{1}\times B} \\ \hline H^B & h \to \text{zip}_H(\text{wu}_H \times h) & \mathbb{0} \\ F^B & \mathbb{0} & f \to \text{zip}_F((\text{wu}_H \triangleright \text{ex}_H \triangleright \text{pu}_F) \times f) \end{array} \right| \quad .$$

Using Eq. (8.12) and the definition of `wu` through `pure`, we find:

$$\text{wu}_H \triangleright \text{ex}_H \triangleright \text{pu}_F = 1 \triangleright \text{pu}_F = \text{wu}_F \quad .$$

Since the identity laws of F and H are assumed to hold, we can transform the last matrix as:

$$\left| \begin{array}{c|cc} & H^{\mathbb{1}\times B} & F^{\mathbb{1}\times B} \\ \hline H^B & h \to \text{zip}_H(\text{wu}_H \times h) & \mathbb{0} \\ F^B & \mathbb{0} & f \to \text{zip}_F((\text{wu}_H \triangleright \text{ex}_H \triangleright \text{pu}_F) \times f) \end{array} \right| = \left| \begin{array}{c|cc} & H^{\mathbb{1}\times B} & F^{\mathbb{1}\times B} \\ \hline H^B & \text{ilu}^{\uparrow H} \pi_2^{\downarrow H} & \mathbb{0} \\ F^B & \mathbb{0} & \text{ilu}^{\uparrow F} \pi_2^{\downarrow F} \end{array} \right|$$

$$= \text{ilu}^{\uparrow P} \pi_2^{\downarrow P} \quad .$$

After this simplification, the left-hand side equals $p \triangleright \text{ilu}^{\uparrow P} \pi_2^{\downarrow P}$ (the right-hand side of the law).

The right identity law is verified in a similar way:

$$\text{zip}_P(p \times \text{wu}_L) = \text{zip}_P\left(p^{:H^A+F^A} \times (\text{wu}_H^{:H^{\mathbb{1}}} + \mathbb{0}^{:F^{\mathbb{1}}})\right)$$

$$= p \triangleright \left| \begin{array}{c|cc} & H^{A\times \mathbb{1}} & F^{A\times \mathbb{1}} \\ \hline H^A & h \to \text{zip}_H(h \times \text{wu}_H) & \mathbb{0} \\ F^A & \mathbb{0} & f \to \text{zip}_F(f \times (\text{wu}_H \triangleright \text{ex}_H \triangleright \text{pu}_F)) \end{array} \right|$$

$$= p \triangleright \begin{array}{c|c|cc|} & & H^{A\times\mathbb{1}} & F^{A\times\mathbb{1}} \\ \hline & H^A & \text{iru}^{\uparrow H}\pi_1^{\downarrow H} & \mathbb{0} \\ & F^A & \mathbb{0} & \text{iru}^{\uparrow F}\pi_1^{\downarrow F} \end{array} = p \triangleright \text{iru}^{\uparrow P}\pi_1^{\downarrow P} \quad .$$

The associativity law is an equation between values of type $H^{A\times B\times C} + F^{A\times B\times C}$:

$$\text{zip}_P(p^{:H^A+F^A} \times \text{zip}_P(q^{:H^B+F^B} \times r^{:H^C+F^C})) \triangleright \varepsilon^{\uparrow P}_{1,23}\tilde{\varepsilon}^{\downarrow P}_{1,23} = \text{zip}_P(\text{zip}_P(p\times q)\times r) \triangleright \varepsilon^{\uparrow P}_{12,3}\tilde{\varepsilon}^{\downarrow P}_{12,3} \quad .$$

The operation $\text{zip}_P(p\times q)$ is defined in such a way that it returns a value of type $H^{A\times B}+\mathbb{0}$ only when both p and q are in the left part of the disjunction:

$$\text{zip}_P\big((a^{:H^A}+\mathbb{0}^{:F^A})\times(b^{:H^B}+\mathbb{0}^{:F^B})\big) = \text{zip}_H(a\times b)+\mathbb{0}^{:F^{A\times B}} \quad .$$

Otherwise, $\text{zip}_P(p\times q)$ returns a value of type $\mathbb{0}^{:H^{A\times B}}+F^{A\times B}$. So, let us consider three cases:

(1) The arguments are $p = a^{:H^A}+\mathbb{0}$, $q = b^{:H^B}+\mathbb{0}$, $r = c^{:H^C}+\mathbb{0}$. In this case, zip_P reduces to zip_H:

$$\text{left-hand side}: \quad \text{zip}_P(p\times\text{zip}_P(q\times r)) \triangleright \varepsilon^{\uparrow P}_{1,23}\tilde{\varepsilon}^{\downarrow P}_{1,23} = \text{zip}_H\big(a\times\text{zip}_H(b\times c)\big) \triangleright \varepsilon^{\uparrow H}_{1,23}\tilde{\varepsilon}^{\downarrow H}_{1,23}+\mathbb{0} \quad ,$$

$$\text{right-hand side}: \quad \text{zip}_P(\text{zip}_P(p\times q)\times r) \triangleright \varepsilon^{\uparrow P}_{12,3}\tilde{\varepsilon}^{\downarrow P}_{12,3} = \text{zip}_H\big(\text{zip}_H(a\times b)\times c\big) \triangleright \varepsilon^{\uparrow H}_{12,3}\tilde{\varepsilon}^{\downarrow H}_{12,3}+\mathbb{0} \quad .$$

The two sides are equal due to the associativity law of zip_H.

(2) The argument q has type $\mathbb{0}+F^B$. In this case, zip_P reduces to zip_F after converting arguments of type $H^\bullet+0$ to type $F^\bullet$ when needed. We may define this conversion as a helper function `toF` in the same way as in the proof of Statement 11.3.6.7. The associativity law of zip_P is then reduced to the same law of zip_F:

$$\text{zip}_P(p\times\text{zip}_P(q\times r)) \triangleright \varepsilon^{\uparrow P}_{1,23}\tilde{\varepsilon}^{\downarrow P}_{1,23} = \mathbb{0}+\text{zip}_F\big(\text{toF}(p)\times\text{zip}_F(\text{toF}(q)\times\text{toF}(r))\big) \triangleright \varepsilon^{\uparrow F}_{1,23}\tilde{\varepsilon}^{\downarrow F}_{1,23} \quad ,$$

$$\text{zip}_P(\text{zip}_P(p\times q)\times r) \triangleright \varepsilon^{\uparrow P}_{12,3}\tilde{\varepsilon}^{\downarrow P}_{12,3} = \mathbb{0}+\text{zip}_F(\text{zip}_F(\text{toF}(p)\times\text{toF}(q))\times\text{toF}(r)) \triangleright \varepsilon^{\uparrow F}_{12,3}\tilde{\varepsilon}^{\downarrow F}_{12,3} \quad .$$

The two sides are equal due to the associativity law of zip_F.

(3) Either $p = \mathbb{0}+a^{:F^A}$ while $q = b^{:H^B}+\mathbb{0}$ and $r = c^{:H^C}+\mathbb{0}$; or $r = \mathbb{0}+c^{:F^C}$ while $p = a^{:H^A}+\mathbb{0}$ and $q = b^{:H^B}+\mathbb{0}$. The two situations are symmetric, so let us consider the first one:

$$\begin{aligned} \text{left-hand side}: \quad & \text{zip}_P(p\times\text{zip}_P(q\times r)) \triangleright \varepsilon^{\uparrow P}_{1,23}\tilde{\varepsilon}^{\downarrow P}_{1,23} \\ & = \mathbb{0}+\text{zip}_F\big(\text{toF}(p)\times\text{toF}(\text{zip}_H(b\times c)+\mathbb{0})\big) \triangleright \varepsilon^{\uparrow F}_{1,23}\tilde{\varepsilon}^{\downarrow F}_{1,23} \quad . \end{aligned}$$

Simplify the sub-expressions involving `toF` separately:

$$\begin{aligned} & \text{toF}(p) = \text{toF}(\mathbb{0}+a) = a \quad , \\ & \text{toF}(\text{zip}_H(b\times c)+\mathbb{0}) = \text{pu}_F(\underline{\text{ex}_H(\text{zip}_H}(b\times c)) \\ \text{use Eq. (11.10)}: \quad & = \text{pu}_F(\text{ex}_H(b)\times\text{ex}_H(c)) \quad . \end{aligned}$$

For profunctors, the right identity law of `zip` and `pure` has the form:

$$\text{zip}_F(a^{:F^A}\times\text{pu}_F(b^{:B})) = a \triangleright (k^{:A}\to k\times b)^{\uparrow F}\pi_1^{\downarrow F} \quad .$$

So, we can rewrite the left-hand side of the associativity law like this:

$$\begin{aligned} \text{left-hand side}: \quad & \text{zip}_P(p\times\text{zip}_P(q\times r)) \triangleright \varepsilon^{\uparrow P}_{1,23}\tilde{\varepsilon}^{\downarrow P}_{1,23} = \mathbb{0}+\underline{\text{zip}_F}\big(a\times\underline{\text{pu}_F}(\text{ex}_H(b)\times\text{ex}_H(c))\big) \triangleright \varepsilon^{\uparrow F}_{1,23}\tilde{\varepsilon}^{\downarrow F}_{1,23} \\ \text{identity law of } \text{zip}_F: \quad & = \mathbb{0}+a \triangleright \big(k^{:A}\to k\times(\text{ex}_H(b)\times\text{ex}_H(c))\big)^{\uparrow F}\pi_1^{\downarrow F} \triangleright \varepsilon^{\uparrow F}_{1,23}\tilde{\varepsilon}^{\downarrow F}_{1,23} \\ & = \mathbb{0}+a \triangleright \big(k^{:A}\to k\times\text{ex}_H(b)\times\text{ex}_H(c)\big)^{\uparrow F}\pi_1^{\downarrow F} \quad . \end{aligned}$$

The right-hand side can be transformed by using `toF` on all arguments:

$$\begin{aligned}
\text{right-hand side}: \quad & \mathrm{zip}_P(\mathrm{zip}_P(p\times q)\times r)\triangleright\varepsilon^{\uparrow P}_{12,3}\tilde{\varepsilon}^{\downarrow P}_{12,3}\\
&=\mathbb{0}+\mathrm{zip}_F(\mathrm{zip}_F(\mathrm{toF}\,(p)\times\mathrm{toF}\,(q))\times\mathrm{toF}\,(r))\triangleright\varepsilon^{\uparrow F}_{12,3}\tilde{\varepsilon}^{\downarrow F}_{12,3}\\
&=\mathbb{0}+\mathrm{zip}_F(\mathrm{zip}_F(a\times\mathrm{toF}\,(b+\mathbb{0}))\times\mathrm{toF}\,(r+\mathbb{0}))\triangleright\varepsilon^{\uparrow F}_{12,3}\tilde{\varepsilon}^{\downarrow F}_{12,3}\quad .
\end{aligned}$$

Simplify the sub-expressions of the form $\mathrm{zip}_F(a\times\mathrm{toF}\,(b+\mathbb{0}))$:

$$\mathrm{zip}_F\left(a^{:F^A}\times\mathrm{toF}\,(b^{:H^B}+\mathbb{0})\right)=\mathrm{zip}_F(a\times\mathrm{pu}_F(\mathrm{ex}_H(b))=a\triangleright(k^{:A}\to k\times\mathrm{ex}_H(b))^{\uparrow F}\pi_1^{\downarrow F}\quad . \tag{11.14}$$

Using this formula, we continue to transform the right-hand side:

$$\begin{aligned}
\text{right-hand side}: \quad & \mathbb{0}+\underline{\mathrm{zip}_F}(\mathrm{zip}_F(a\times\mathrm{toF}\,(b+\mathbb{0}))\times\underline{\mathrm{toF}\,(c+\mathbb{0})})\triangleright\varepsilon^{\uparrow F}_{12,3}\tilde{\varepsilon}^{\downarrow F}_{12,3}\\
\text{use Eq. (11.14)}: \quad & =\mathbb{0}+\underline{\mathrm{zip}_F}(a\times\underline{\mathrm{toF}\,(b+\mathbb{0})})\triangleright\left(k\to k\times\mathrm{ex}_H(c)\right)^{\uparrow F}\pi_1^{\downarrow F}\triangleright\varepsilon^{\uparrow F}_{12,3}\tilde{\varepsilon}^{\downarrow F}_{12,3}\\
\text{use Eq. (11.14)}: \quad & =\mathbb{0}+a\triangleright\left(k\to k\times\mathrm{ex}_H(b)\right)^{\uparrow F}\pi_1^{\downarrow F}\triangleright\left(k\to k\times\mathrm{ex}_H(c)\right)^{\uparrow F}\pi_1^{\downarrow F}\triangleright\varepsilon^{\uparrow F}_{12,3}\tilde{\varepsilon}^{\downarrow F}_{12,3}\\
\text{compute composition}: \quad & =\mathbb{0}+a\triangleright\left(k\to k\times\mathrm{ex}_H(b)\times\mathrm{ex}_H(c)\right)^{\uparrow F}\pi_1^{\downarrow F}\quad .
\end{aligned}$$

The two sides are now equal.

It remains to verify the commutativity law in case that law holds for F and H:

$$\mathrm{swap}\,;\mathrm{zip}_F\overset{!}{=}\mathrm{zip}_F\,;\mathrm{swap}^{\uparrow F}\mathrm{swap}^{\downarrow F}\quad,\quad \mathrm{swap}\,;\mathrm{zip}_H\overset{!}{=}\mathrm{zip}_H\,;\mathrm{swap}^{\uparrow H}\mathrm{swap}^{\downarrow H}\quad .$$

Begin with the left-hand side of the commutativity law for zip_P:

$$\mathrm{swap}\,;\mathrm{zip}_P=\begin{array}{|c||cc|}
 & H^{B\times A} & F^{B\times A}\\
\hline
H^A\times H^B & \mathrm{swap}\,;\mathrm{zip}_H & \mathbb{0}\\
F^A\times H^B & \mathbb{0} & \mathrm{swap}\,;((\mathrm{ex}_H\,;\mathrm{pu}_F)\boxtimes\mathrm{id})\,;\mathrm{zip}_F\\
H^A\times F^B & \mathbb{0} & \mathrm{swap}\,;(\mathrm{id}\boxtimes(\mathrm{ex}_H\,;\mathrm{pu}_F))\,;\mathrm{zip}_F\\
F^A\times F^B & \mathbb{0} & \mathrm{swap}\,;\mathrm{zip}_F
\end{array}\quad .$$

Writing out the compositions of `swap` and the pair product functions, we get:

$$\begin{aligned}
&\mathrm{swap}\,;((\mathrm{ex}_H\,;\mathrm{pu}_F)\boxtimes\mathrm{id})=(\mathrm{id}\boxtimes(\mathrm{ex}_H\,;\mathrm{pu}_F))\,;\mathrm{swap}\quad ,\\
&\mathrm{swap}\,;(\mathrm{id}\boxtimes(\mathrm{ex}_H\,;\mathrm{pu}_F))=(\mathrm{pu}_F\boxtimes\mathrm{id})\,;\mathrm{swap}\quad .
\end{aligned}$$

Using these simplifications, we rewrite the left-hand side as:

$$\mathrm{swap}\,;\mathrm{zip}_P=\begin{array}{|c||cc|}
 & H^{B\times A} & F^{B\times A}\\
\hline
H^A\times H^B & \mathrm{swap}\,;\mathrm{zip}_H & \mathbb{0}\\
F^A\times H^B & \mathbb{0} & (\mathrm{id}\boxtimes(\mathrm{ex}_H\,;\mathrm{pu}_F))\,;\mathrm{swap}\,;\mathrm{zip}_F\\
H^A\times F^B & \mathbb{0} & ((\mathrm{ex}_H\,;\mathrm{pu}_F)\boxtimes\mathrm{id})\,;\mathrm{swap}\,;\mathrm{zip}_F\\
F^A\times F^B & \mathbb{0} & \mathrm{swap}\,;\mathrm{zip}_F
\end{array}\quad .$$

The right-hand side is rewritten to the same code after using the laws of zip_F and zip_H:

$$\text{zip}_P \fatsemi \text{swap}^{\uparrow P} \text{swap}^{\downarrow P}$$

$$= \begin{array}{c||cc|} & H^{A\times B} & F^{A\times B} \\ \hline H^A \times H^B & \text{zip}_H & \mathbb{0} \\ F^A \times H^B & \mathbb{0} & (\text{id} \boxtimes (\text{ex}_H \fatsemi \text{pu}_F)) \fatsemi \text{zip}_F \\ H^A \times F^B & \mathbb{0} & ((\text{ex}_H \fatsemi \text{pu}_F) \boxtimes \text{id}) \fatsemi \text{zip}_F \\ F^A \times F^B & \mathbb{0} & \text{zip}_F \end{array} \fatsemi \begin{array}{c||cc|} & H^{B\times A} & F^{B\times A} \\ \hline H^{A\times B} & \text{swap}^{\uparrow H} \text{swap}^{\downarrow H} & \mathbb{0} \\ F^{A\times B} & \mathbb{0} & \text{swap}^{\uparrow F} \text{swap}^{\downarrow F} \end{array}$$

$$= \begin{array}{c||cc|} & H^{B\times A} & F^{B\times A} \\ \hline A \times B & \text{swap} \fatsemi \text{zip}_H & \mathbb{0} \\ F^A \times B & \mathbb{0} & (\text{id} \boxtimes (\text{ex}_H \fatsemi \text{pu}_F)) \fatsemi \text{swap} \fatsemi \text{zip}_F \\ A \times F^B & \mathbb{0} & ((\text{ex}_H \fatsemi \text{pu}_F) \boxtimes \text{id}) \fatsemi \text{swap} \fatsemi \text{zip}_F \\ F^A \times F^B & \mathbb{0} & \text{swap} \fatsemi \text{zip}_F \end{array} \quad .$$

The two sides are now equal. □

Statement 11.4.2.7 If $G^\bullet$ is an applicative profunctor and $H^\bullet$ is *any functor* then the profunctor $P^A \stackrel{\text{def}}{=} H^A \to G^A$ is applicative.

Proof We follow the proof of Statement 11.4.1.5. Implement the `zip` and `wu` methods for P:

$$\begin{aligned} &\text{zip}_P : (H^A \to G^A) \times (H^B \to G^B) \to H^{A\times B} \to G^{A\times B} \quad , \\ &\text{zip}_P(p^{:H^A\to G^A} \times q^{:H^B\to G^B}) \stackrel{\text{def}}{=} h^{:H^{A\times B}} \to \text{zip}_G(p(h \triangleright \pi_1^{\uparrow H}) \times q(h \triangleright \pi_2^{\uparrow H})) \quad , \\ &\text{wu}_P \stackrel{\text{def}}{=} _^{:H^{\mathbb{1}}} \to \text{wu}_G \quad . \end{aligned}$$

We will also use the definition of zip_P in a point-free form, which omits the argument h:

$$\text{zip}_P(p \times q) \stackrel{\text{def}}{=} \Delta \fatsemi (\pi_1^{\uparrow H} \boxtimes \pi_2^{\uparrow H}) \fatsemi (p \boxtimes q) \fatsemi \text{zip}_G \quad .$$

The code for lifting to P is standard:

$$(p^{:H^A\to G^A}) \triangleright f^{\downarrow P} g^{\uparrow P} = f^{\uparrow H} \fatsemi p \fatsemi (f^{\downarrow G} g^{\uparrow G}) \quad .$$

To verify the left identity law of P, we use the left identity law of G:

$$\begin{aligned} & h^{:H^{\mathbb{1}\times A}} \triangleright \text{zip}_P(\text{wu}_P \times p^{:H^A\to G^A}) = \text{zip}_G(\underline{\text{wu}_P(h \triangleright \pi_1^{\uparrow H})} \times p(h \triangleright \pi_2^{\uparrow H})) \\ \text{definition of } \text{wu}_P : \quad & = \text{zip}_G(\text{wu}_G \times p(h \triangleright \pi_2^{\uparrow H})) \\ \text{left identity law of } G : \quad & = p(h \triangleright \pi_2^{\uparrow H}) \triangleright \text{ilu}^{\uparrow G} \pi_2^{\downarrow G} = h \triangleright \underline{\pi_2^{\uparrow H} \fatsemi p \fatsemi \text{ilu}^{\uparrow G} \pi_2^{\downarrow G}} = h \triangleright (p \fatsemi \text{ilu}^{\uparrow P} \pi_2^{\downarrow P}) \quad . \end{aligned}$$

To verify the right identity law of P:

$$\begin{aligned} & h^{:H^{A\times\mathbb{1}}} \triangleright \text{zip}_P(p^{:H^A\to G^A} \times \text{wu}_P) = \text{zip}_G(p(h \triangleright \pi_1^{\uparrow H}) \times \text{wu}_P(h \triangleright \pi_2^{\uparrow H})) = \text{zip}_G(p(h \triangleright \pi_1^{\uparrow H}) \times \text{wu}_G) \\ & = p(h \triangleright \pi_1^{\uparrow H}) \triangleright \text{iru}^{\uparrow G} \pi_1^{\downarrow G} = h \triangleright \underline{\pi_1^{\uparrow H} \fatsemi p \fatsemi \text{iru}^{\uparrow G} \pi_1^{\downarrow G}} = h \triangleright (p \fatsemi \text{iru}^{\uparrow P} \pi_1^{\downarrow P}) \quad . \end{aligned}$$

To verify the associativity law, we use the definition of $^{\uparrow P}$:

$$\begin{aligned}
\text{left-hand side}: \quad & h^{:H^{A\times B\times C}} \triangleright \left(\text{zip}_P(p\times \text{zip}_P(q\times r)) \triangleright \underline{\varepsilon^{\uparrow P}_{1,23}\tilde{\varepsilon}^{\downarrow P}_{1,23}}\right) \\
&= h^{:H^{A\times B\times C}} \triangleright \tilde{\varepsilon}^{\uparrow H}_{1,23} \triangleright \text{zip}_P(p\times \text{zip}_P(q\times r)) \triangleright \varepsilon^{\uparrow G}_{1,23}\tilde{\varepsilon}^{\downarrow G}_{1,23} \\
&= \text{zip}_G\left(p(h\triangleright \underline{\tilde{\varepsilon}^{\uparrow H}_{1,23} ⨾ \pi^{\uparrow H}_1}) \times \text{zip}_P(q\times r)(h\triangleright \tilde{\varepsilon}^{\uparrow H}_{1,23} ⨾ \pi^{\uparrow H}_2)\right) \triangleright \varepsilon^{\uparrow G}_{1,23}\tilde{\varepsilon}^{\downarrow G}_{1,23} \\
&= \text{zip}_G\left(p(h\triangleright \pi^{\uparrow H}_1) \times \text{zip}_G\left(q(h\triangleright \pi^{\uparrow H}_2) \times r(h\triangleright \pi^{\uparrow H}_3)\right)\right) \triangleright \varepsilon^{\uparrow G}_{1,23}\tilde{\varepsilon}^{\downarrow G}_{1,23} \quad , \\
\text{right-hand side}: \quad & h^{:H^{A\times B\times C}} \triangleright \tilde{\varepsilon}^{\uparrow H}_{12,3} \triangleright \text{zip}_P(\text{zip}_P(p\times q)\times r) \triangleright \varepsilon^{\uparrow G}_{12,3}\tilde{\varepsilon}^{\downarrow G}_{12,3} \\
&= \text{zip}_G\left(\text{zip}_P(p\times q)(h\triangleright \tilde{\varepsilon}^{\uparrow H}_{12,3} ⨾ \pi^{\uparrow H}_1) \times r(h\triangleright \tilde{\varepsilon}^{\uparrow H}_{12,3} ⨾ \pi^{\uparrow H}_2)\right) \triangleright \varepsilon^{\uparrow G}_{12,3}\tilde{\varepsilon}^{\downarrow G}_{12,3} \\
&= \text{zip}_G\left(\text{zip}_G\left(p(h\triangleright \pi^{\uparrow H}_1) \times q(h\triangleright \pi^{\uparrow H}_2)\right) \times r(h\triangleright \pi^{\uparrow H}_3)\right) \triangleright \varepsilon^{\uparrow G}_{12,3}\tilde{\varepsilon}^{\downarrow G}_{12,3} \quad .
\end{aligned}$$

The two sides are now equal due to the associativity law of zip_G.

It remains to verify the commutativity law, assuming that zip_G satisfies that law:

$$\begin{aligned}
&\text{zip}_P(q\times p) = \Delta ⨾ (\pi^{\uparrow H}_1 \boxtimes \pi^{\uparrow H}_2) ⨾ (q\boxtimes p) ⨾ \text{zip}_G \quad , \\
&\text{zip}_P(p\times q) \triangleright \text{swap}^{\uparrow P}\text{swap}^{\downarrow P} = \text{swap}^{\uparrow H} ⨾ \Delta ⨾ (\pi^{\uparrow H}_1 \boxtimes \pi^{\uparrow H}_2) ⨾ (p\boxtimes q) ⨾ \underline{\text{zip}_G ⨾ \text{swap}^{\uparrow G}\text{swap}^{\downarrow G}} \\
&\quad = \Delta ⨾ \underline{(\text{swap}^{\uparrow H} \boxtimes \text{swap}^{\uparrow H}) ⨾ (\pi^{\uparrow H}_1 \boxtimes \pi^{\uparrow H}_2)} ⨾ \underline{(p\boxtimes q) ⨾ \text{swap}} ⨾ \text{zip}_G \\
&\quad = \underline{\Delta ⨾ (\pi^{\uparrow H}_2 \boxtimes \pi^{\uparrow H}_1) ⨾ \text{swap}} ⨾ (q\boxtimes p) ⨾ \text{zip}_G = \Delta ⨾ (\pi^{\uparrow H}_1 \boxtimes \pi^{\uparrow H}_2) ⨾ (q\boxtimes p) ⨾ \text{zip}_G \quad .
\end{aligned}$$

□

Note that the functor H must be *covariant* in the last construction ($P^A \stackrel{\text{def}}{=} H^A \to G^A$). Otherwise, the profunctor P will not be applicative even the simple cases such as $P^A \stackrel{\text{def}}{=} H^A \to A$ where H is an arbitrary profunctor. (We omit the proof of that statement.)

11.5 Discussion and further developments

11.5.1 Equivalence of typeclass methods with laws

In this and the previous chapters, we have seen that certain typeclass methods are equivalent. The equivalence of `map2`, `zip`, and `ap` was proved in Section 11.3.1. For filterable functors, we have shown that the methods `filter`, `deflate`, and `liftOpt` are equivalent if some naturality laws hold (Sections 9.2.2 and 9.2.3). For monads, we proved that `flatMap` is equivalent to `flatten` (Statement 10.2.2.1). Another case of equivalence between certain function types was proved in Section 10.3.2.

Perhaps the simplest example of this sort of equivalence is that between the `pure` method and the "wrapped unit" (`wu`), as we saw in Section 8.3.5. A function $\text{pu}_F : A \to F^A$ satisfying a naturality law is equivalent to a value $\text{wu}_F : F^{\mathbb{1}}$.

After seeing those detailed proofs, we can now clarify the meaning of "equivalence under laws". The goal of this subsection is to find a rigorous formulation of that equivalence.

In each case seen so far, we have two functions with two different type signatures (usually with type parameters), and we assume that certain naturality laws hold. It is important to keep in mind that the naturality laws are equations whose form *automatically* follows from type signatures. If a function has more than one type parameter, there is one naturality law per type parameter (the full details are in Section 9.4.1 and Appendix D). For example, the type signature `pure[A]: A => F[A]` (in the type notation, $\text{pu}_F : \forall A.\, A \to F^A$) gives rise to the naturality law $f ⨾ \text{pu}_F = \text{pu}_F ⨾ f^{\uparrow F}$. In Scala code, this law is written as `pure(f(x)) == pure(x).map(f)`.

What we have proved in Section 8.3.5 is a one-to-one correspondence between two *sets*: the set of all values `wu` of type `F[Unit]`, and the set of all functions with type `pure[A]: A => F[A]` satisfying

the naturality law `pure(f(x)) == pure(x).map(f)`. The proof defines explicit mappings between `wu` and `pure` in both directions and verifies that their compositions are identity maps. We also made sure to exclude all functions of type `A => F[A]` that fail to satisfy the naturality law. For instance, we proved that if `pure` is defined via `wu` then the naturality law will hold for `pure`.

So, if `wu: F[Unit]` is defined in some way for a functor `F`, we will always have a corresponding definition of `pure` that satisfies the naturality law. Conversely, for any given lawful definition of `pure` we will have a corresponding value `wu`.

The same pattern was used in the proofs of equivalence in other chapters. Let us formulate this pattern in a more abstract way. We would like to prove that some function p (of type $P^{A,B,\dots}$) is equivalent to some q (of type $Q^{A,B,\dots}$) when suitable naturality laws hold. These functions may have any number of type parameters (A, B, etc.). More explicitly, the types of p and q are:

$$p : \forall(A, B, \dots).\, P^{A,B,\dots} \quad , \qquad q : \forall(A, B, \dots).\, Q^{A,B,\dots} \quad .$$

We view the type $\forall(A, B, \dots).\, P^{A,B,\dots}$ as a set of all functions p (implementable in Scala) with this code:

```
def p[A, B, ...]: P[A, B, ...] = { ... }
```

and similarly for q. To establish the equivalence, we first write some code that expresses p through q and back. This requires two functions, $f : P^{A,B,\dots} \to Q^{A,B,\dots}$ and $g : Q^{A,B,\dots} \to P^{A,B,\dots}$. These functions are mappings (in both directions) between the set of all functions p and the set of all functions q of the corresponding types. Then we prove that the mappings are one-to-one: for any p and q we have $g(f(p)) = p$ and $f(g(q)) = q$. Finally, we check that the required naturality laws still hold after applying the mappings f and g. In other words, for any p satisfying its naturality law, we prove that the function $f(p)$ of type $Q^{A,B,\dots}$ will satisfy *its* naturality law; and similarly for q and $g(q)$.

Without the naturality laws, the types $\forall(A, B, \dots).\, P^{A,B,\dots}$ and $\forall(A, B, \dots).\, Q^{A,B,\dots}$ are usually *not* equivalent: there is no one-to-one correspondence between functions p and functions q of those types. Imposing a naturality law means that we exclude functions p of type $\forall(A, B, \dots).\, P^{A,B,\dots}$ that do not satisfy the naturality law. This gives us a smaller set of all lawful functions p (and similarly for q). The equivalence holds only for those smaller sets of lawful functions.

To see this in a simple example, consider the equivalence of p=`pure` and q=`wu` for the identity functor ($L^A \stackrel{\text{def}}{=} A$). The method `pure` has type $\forall A.\, A \to A$, while `wu` has the `Unit` type ($L^{\mathbb{1}} = \mathbb{1}$). There is only one value `wu`, but there are many possible functions p of type $\forall A.\, A \to A$. For instance, we could define a function p like this:

```
def p[A]: A => A = {
  case a: Int   => (a + 123)
              .asInstanceOf[A]
  case a        => a
}
```

This function is defined for all types A, so it fits the type $\forall A.\, A \to A$. The set of all functions of type $\forall A.\, A \to A$ contains a large number of similarly defined functions. For example, we may replace `123` by another number or select another type instead of `Int`. However, all those functions fail to satisfy the naturality law, which has the form $f^{:A\to B} ⨾ p^B = p^A ⨾ f$. The only function p that satisfies this naturality law is the identity function, $p^A = \text{id}^A \stackrel{\text{def}}{=} a^{:A} \to a$ (see Exercise 4.4.2.15). We find that imposing the naturality law removes all functions p except one ($p = \text{id}$) from the set of functions $p : \forall A.\, A \to A$. There remains a set containing a single element (id), which is in a one-to-one correspondence with the set of values of type $\mathbb{1}$.

So, the rigorous meaning of an "equivalence between functions p and q assuming some laws" is a one-to-one correspondence between the set of all functions $p : \forall(A, B, \dots).\, P^{A,B,\dots}$ that satisfy the given laws of p, and the set of all functions $q : \forall(A, B, \dots).\, Q^{A,B,\dots}$ that satisfy the given laws of q.

Note that the Scala compiler is unable to check automatically that the naturality laws hold for a function p with type parameters. Indeed, those laws are equations for p that involve arbitrary functions $f^{:A\to B}$ with arbitrary types A and B. Such equations cannot be enforced by the type signature of p. Neither can we prove naturality laws by running tests: each test will have to use specific types as the type parameters, while the purpose of naturality laws is to verify that the function works in the same way for *all* types. Naturality laws can be verified only through a proof that uses symbolic reasoning. (The proof can be omitted if the code is fully parametric, as shown in Appendix D.)

The Scala compiler also cannot verify the laws specific to a given typeclass (e.g., the identity and composition laws of filterable functors, or the identity and associativity laws of monads and applicative functors). These laws must be also verified by symbolic reasoning. It is often easier to verify laws if the type signature of the method has fewer type parameters. For instance, the `zip` method has two type parameters while `map2` has three; `flatten` has one type parameter while `flatMap` has two. For this reason, we have systematically derived the laws of all the equivalent typeclass methods. In many cases, we found a formulation of the laws that was either conceptually simpler or more straightforward to verify.

11.5.2 Relationship between monads and applicative functors

Any lawful monad gives its type constructor at least one applicative functor definition: as we showed in Section 11.3.5, we may define the `map2` method via `map` and `flatMap` in two ways (which will give the same code if the monad is commutative). The `map2` methods defined in this way will have the right type signature and will satisfy the laws of applicative functors. This is due to the fact that the laws of `map2` are derived (as shown in Section 11.3.3) from the monad laws precisely by considering the `map2` method defined via the monad's `flatMap`.

However, in many cases we need to define the `map2` method in a different way because expressing `map2` via `flatMap` does not give us the required functionality. We have seen several examples of this. For instance, the standard behavior of `map2` for sequences, trees, and `Either` is not compatible with the standard `flatMap` methods of those functors.

So, it is rarely useful to define an `Applicative` typeclass instance automatically for all monads. In most cases, we need to define the applicative instance separately from the monad instance. (Automatic derivation of `Applicative` instances is made difficult also by the fact that many type constructors will admit more than one lawful implementation of `map2`.)

In addition, some applicative functors are not monads. We have shown in Section 11.3.6 that a lawful `Applicative` instance exists for all polynomial functors with monoidal fixed types. (Accordingly, all our examples of non-applicative functors involve non-polynomial functors.) This does not hold for monads; not all polynomial functors are monadic.[10] Example 11.1.1.1 shows a simple applicative functor ($L^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$) that *cannot* have a lawful monad implementation (Exercise 10.2.9.9).

We have also seen that the `zip` and `wu` operations exist for some type constructors that are not covariant. We conclude that the `Applicative` typeclass is larger than the `Monad` typeclass.

11.5.3 Applicative morphisms

One of the applicative constructions (Statement 11.3.6.7) needs a compatibility law (11.10) between the methods zip_H and ex_H. This law is understood better if we rewrite the type signature $\text{ex}_H : H^A \to A$ as $\text{ex}_H : H^A \to \text{Id}^A$, where $\text{Id}^A \stackrel{\text{def}}{=} A$ is the identity functor (which is also applicative). Viewed in this way, the `extract` method is an example of a mapping between two applicative functors (H and Id). So, the compatibility law requires that the operation zip_H be mapped to the tupling operation, which is the same as the `zip` operation of the identity functor: $\text{zip}_{\text{Id}}(a \times b) = a \times b$.

This suggests considering a more general mapping $\phi : H^A \to K^A$ where H and K are two applicative functors, and formulating a **composition law** of ϕ by analogy with Eq. (11.10):

$$\phi\big(\text{zip}_H(p \times q)\big) = \text{zip}_K(\phi(p) \times \phi(q)) \quad , \quad \text{or equivalently}: \quad \text{zip}_H \,⨾\, \phi = (\phi \boxtimes \phi) \,⨾\, \text{zip}_K \quad .$$

This law reproduces Eq. (11.10) when $\phi = \text{ex}_H$ and $K = \text{Id}$ because we will then have $\text{zip}_K = \text{id}$.

It seems reasonable to require also that ϕ should map the method pu_H to pu_K. This gives the **identity law**: $\text{pu}_H \,⨾\, \phi = \text{pu}_K$. Functions ϕ satisfying these two laws are called **applicative morphisms** between two applicative functors H and K. The two laws make applicative morphisms fully analogous to monad morphisms (Section 10.3.4).

[10] It is unknown how to characterize or enumerate all polynomial functors that are monads (see Problems 15.3.2.2–15.3.2.1).

11.5.4 Deriving the laws of `ap` using category theory

Statement 11.3.1.2 shows that the additional capabilities of `map2` compared with `map` can be expressed via the function `ap`:

$$\text{map}_2\,(p^{:L^A} \times q^{:L^B})(f^{:A\times B\to C}) = \text{ap}\,(p \triangleright (a^{:A} \to b^{:B} \to f(a\times b))^{\uparrow L})(q) \quad .$$

How can we derive the laws of `ap` that correspond to the laws of `map2`? If we substitute the equation above into the laws of `map2`, we will obtain a relationship involving arbitrary values $p^{:L^A}$, $q^{:L^B}$, and $f^{:A\times B\to C}$. However, the type signature of `ap` is $L^{A\to B} \to L^A \to L^B$, and so we expect the laws of `ap` to have arbitrary arguments of types $L^{A\to B}$ and L^B. It is not clear how to choose the types in the laws of `map2` in order to derive a complete set of laws for `ap`.

To resolve this issue, we turn to category theory for guidance. The type signature of `ap` looks like a "lifting" from values of type $L^{A\to B}$ to functions of type $L^A \to L^B$. Category theory suggests to describe this lifting as part of a suitable categorical functor. A categorical functor can exist only between two categories. So, we need to show that values of type $L^{A\to B}$ ("wrapped functions") can play the role of morphisms in a suitably defined category. To define that category, we need to produce objects, morphisms, the identity morphism, and the composition operation, and prove their laws. Finally, we will need to prove that `ap` satisfies the identity and composition laws appropriate for a (categorical) functor.

This section will follow these considerations in order to derive and verify the laws of `ap`.

The first step is to define two categories between which we will establish a categorical functor. We would like the morphisms of the first category to be values of type $L^{A\to B}$, and the morphisms of the second category to be functions of type $L^A \to L^B$. The second category was called "L-lifted" in Section 9.4.3. Functions of type $L^A \to L^B$ already satisfy the required properties of morphisms: the identity morphism is the function $\text{id}^{:L^A\to L^A}$, and the ordinary function composition is associative and respects the identity. The objects of the first category may be chosen as ordinary types (A, B, etc.). To establish that values of type $L^{A\to B}$ also satisfy the properties of morphisms, we need to define a composition operation for those values and prove its properties.

It turns out that there are two ways of defining the composition operation, as shown in Statement 11.5.4.1 (a) and (b) below. The difference between the two definitions disappears if L is a *commutative* applicative functor. We will use the definition (b); the choice is dictated by compatibility with the `ap` operation, as we will demonstrate in Statement 11.5.4.2.

Statement 11.5.4.1 Given a functor L with lawful `zip` and `pure` methods, we define an **L-applicative composition** operation (denoted by $\odot$). For any $p : L^{A\to B}$ and $q : L^{B\to C}$, the value $p \odot q$ will be of type $L^{A\to C}$. The operation $\odot$ may be defined in two different ways:

$$\text{definition (a) of } \odot: \quad p^{:L^{A\to B}} \odot q^{:L^{B\to C}} \stackrel{\text{def}}{=} (p\times q) \triangleright \text{zip} \triangleright (f^{:A\to B} \times g^{:B\to C} \to f \fatsemi g)^{\uparrow L} \quad ,$$

$$\text{definition (b) of } \odot: \quad p^{:L^{A\to B}} \odot q^{:L^{B\to C}} \stackrel{\text{def}}{=} (q\times p) \triangleright \text{zip} \triangleright (g^{:B\to C} \times f^{:A\to B} \to f \fatsemi g)^{\uparrow L} \quad .$$

Moreover, the special "wrapped identity" value, `wid`, of type $L^{A\to A}$, is defined by $\text{wid}_L^A \stackrel{\text{def}}{=} \text{pu}_L(\text{id}^A)$. The following properties then hold for each of the two definitions:

$$\text{left identity law of } \odot: \quad \text{wid}_L^A \odot p^{:L^{A\to B}} = p \quad ,$$

$$\text{right identity law of } \odot: \quad p^{:L^{A\to B}} \odot \text{wid}_L^B = p \quad ,$$

$$\text{associativity law of } \odot: \quad (p^{:L^{A\to B}} \odot q^{:L^{B\to C}}) \odot r^{:L^{C\to D}} = p \odot (q \odot r) \quad .$$

Proof We will prove the laws separately for the definitions (a) and (b).

(a) To verify the left identity law, substitute the definition (a) of $\odot$:

$$\text{expect to equal } p: \quad \text{wid}_L \odot p = \text{pu}_L(\text{id}) \odot p = (\text{pu}_L(\text{id}) \times p) \triangleright \text{zip} \fatsemi (f\times g \to f\fatsemi g)^{\uparrow L}$$

$$\text{left identity law of zip}: \quad = p \triangleright (s^{:A\to B} \to \text{id}\times s)^{\uparrow L} \fatsemi (f\times g \to f\fatsemi g)^{\uparrow L}$$

$$\text{function composition under } {}^{\uparrow L}: \quad = p \triangleright (\underline{s \to \text{id}\fatsemi s})^{\uparrow L} = p \triangleright \text{id}^{\uparrow L} = p \quad .$$

To verify the right identity law:

$$\begin{aligned}
\text{expect to equal } p: \quad & p \odot \text{wid}_L = p \odot \text{pu}_L(\text{id}) = (p \times \text{pu}_L(\text{id})) \triangleright \text{zip} ⨾ (f \times g \to f ⨾ g)^{\uparrow L} \\
\text{right identity law of zip}: \quad & = p \triangleright (s^{:A \to B} \to s \times \text{id})^{\uparrow L} ⨾ (f \times g \to f ⨾ g)^{\uparrow L} \\
\text{function composition under } {}^{\uparrow L}: \quad & = p \triangleright \underline{(s \to s ⨾ \text{id})}^{\uparrow L} = p \triangleright \text{id}^{\uparrow L} = p \quad .
\end{aligned}$$

To verify the composition law, begin rewriting its left-hand side:

$$\begin{aligned}
\text{left-hand side}: \quad & (p \odot q) \odot r = \big((p \times q) \triangleright \text{zip} ⨾ (f \times g \to f ⨾ g)^{\uparrow L}\big) \odot r \\
& = \big((\text{zip}\,(p \times q) \triangleright \underline{(f \times g \to f ⨾ g)^{\uparrow L}}) \times r\big) \triangleright \underline{\text{zip}} ⨾ (k \times h \to k ⨾ h)^{\uparrow L} \\
\text{naturality law of zip}: \quad & = (\text{zip}\,(p \times q) \times r) \triangleright \text{zip} ⨾ \underline{((f \times g) \times h \to (f ⨾ g) \times h)^{\uparrow L} ⨾ (k \times h \to k ⨾ h)^{\uparrow L}} \\
\text{composition under } {}^{\uparrow L}: \quad & = \text{zip}\,(\text{zip}\,(p \times q) \times r) \triangleright ((f \times g) \times h \to f ⨾ g ⨾ h)^{\uparrow L} \quad .
\end{aligned}$$

The right-hand side is rewritten similarly:

$$\begin{aligned}
\text{right-hand side}: \quad & p \odot (q \odot r) = p \odot \big(\text{zip}\,(q \times r) \triangleright (f \times g \to f ⨾ g)^{\uparrow L}\big) \\
& = \big(p \times (\text{zip}\,(q \times r) \triangleright \underline{(g \times h \to g ⨾ h)^{\uparrow L}})\big) \triangleright \underline{\text{zip}} ⨾ (f \times k \to f ⨾ k)^{\uparrow L} \\
\text{naturality law of zip}: \quad & = (p \times \text{zip}\,(q \times r)) \triangleright \text{zip} ⨾ \underline{(f \times (g \times h) \to f \times (g ⨾ h))^{\uparrow L} ⨾ (f \times k \to f ⨾ k)^{\uparrow L}} \\
\text{composition under } {}^{\uparrow L}: \quad & = \text{zip}\,\big(p \times \text{zip}\,(q \times r)\big) \triangleright (f \times (g \times h) \to f ⨾ g ⨾ h)^{\uparrow L} \quad .
\end{aligned}$$

It is clear that we need to use the associativity law of `zip`. To be able to use that law, we express the functions $(f \times g) \times h \to f ⨾ g ⨾ h$ and $f \times (g \times h) \to f ⨾ g ⨾ h$ through the conversion functions $\varepsilon_{1,23}$ and $\varepsilon_{12,3}$ (defined in Section 11.3.4):

$$\begin{aligned}
\big((f \times g) \times h \to f ⨾ g ⨾ h\big) &= \varepsilon_{12,3} ⨾ (f \times g \times h \to f ⨾ g ⨾ h) \quad , \\
\big(f \times (g \times h) \to f ⨾ g ⨾ h\big) &= \varepsilon_{1,23} ⨾ (f \times g \times h \to f ⨾ g ⨾ h) \quad .
\end{aligned}$$

Using these equations, we show that the two sides of the associativity law of $\odot$ are equal:

$$\begin{aligned}
\text{left-hand side}: \quad & \underline{\text{zip}\,\big(\text{zip}\,(p \times q) \times r\big) \triangleright \varepsilon_{12,3}} ⨾ (f \times g \times h \to f ⨾ g ⨾ h) \\
\text{right-hand side}: \quad & = \underline{\text{zip}\,\big(p \times \text{zip}\,(q \times r)\big) \triangleright \varepsilon_{1,23}} ⨾ (f \times g \times h \to f ⨾ g ⨾ h) \quad .
\end{aligned}$$

The underlined expessions are equal due to the associativity law (11.3) of `zip`.

(b) To verify the left identity law, substitute the definition (b) of $\odot$:

$$\begin{aligned}
\text{expect to equal } p: \quad & \text{wid}_L \odot p = \text{pu}_L(\text{id}) \odot p = (p \times \text{pu}_L(\text{id})) \triangleright \text{zip} ⨾ (g \times f \to f ⨾ g)^{\uparrow L} \\
\text{right identity law of zip}: \quad & = p \triangleright (s^{:A \to B} \to s \times \text{id})^{\uparrow L} ⨾ (g \times f \to f ⨾ g)^{\uparrow L} \\
\text{function composition under } {}^{\uparrow L}: \quad & = p \triangleright \underline{(s \to \text{id} ⨾ s)}^{\uparrow L} = p \triangleright \text{id}^{\uparrow L} = p \quad .
\end{aligned}$$

To verify the right identity law:

$$\begin{aligned}
\text{expect to equal } p: \quad & p \odot \text{wid}_L = p \odot \text{pu}_L(\text{id}) = (\text{pu}_L(\text{id}) \times p) \triangleright \text{zip} ⨾ (g \times f \to f ⨾ g)^{\uparrow L} \\
\text{left identity law of zip}: \quad & = p \triangleright (s^{:A \to B} \to \text{id} \times s)^{\uparrow L} ⨾ (g \times f \to f ⨾ g)^{\uparrow L} \\
\text{function composition under } {}^{\uparrow L}: \quad & = p \triangleright \underline{(s \to s ⨾ \text{id})}^{\uparrow L} = p \triangleright \text{id}^{\uparrow L} = p \quad .
\end{aligned}$$

To verify the composition law, begin rewriting its left-hand side:

$$\begin{aligned}
\text{left-hand side}: \quad & (p \odot q) \odot r = \big((q \times p) \triangleright \text{zip} ⨾ (g \times f \to f ⨾ g)^{\uparrow L}\big) \odot r \\
& = \big(r \times (\text{zip}\,(q \times p) \triangleright \underline{(g \times f \to f ⨾ g)^{\uparrow L}})\big) \triangleright \underline{\text{zip}} ⨾ (h \times k \to k ⨾ h)^{\uparrow L} \\
\text{naturality law of zip}: \quad & = (r \times \text{zip}\,(q \times p)) \triangleright \text{zip} ⨾ \underline{(h \times (g \times f) \to h \times (f ⨾ g))^{\uparrow L} ⨾ (h \times k \to k ⨾ h)^{\uparrow L}} \\
\text{composition under } {}^{\uparrow L}: \quad & = \text{zip}\,\big(r \times \text{zip}\,(q \times p)\big) \triangleright (h \times (g \times f) \to f ⨾ g ⨾ h)^{\uparrow L} \quad .
\end{aligned}$$

The right-hand side is rewritten similarly:

$$\begin{aligned}
\text{right-hand side}: \quad & p \odot (q \odot r) = p \odot \left(\text{zip}\,(q \times r) \triangleright (g \times f \to f \,⨾\, g)^{\uparrow L}\right) \\
& = \left(\left(\text{zip}\,(q \times r) \triangleright \underline{(h \times g \to g \,⨾\, h)^{\uparrow L}}\right) \times p\right) \triangleright \underline{\text{zip}} \,⨾\, (k \times f \to f \,⨾\, k)^{\uparrow L} \\
\text{naturality law of zip}: \quad & = (\text{zip}\,(q \times r) \times p) \triangleright \text{zip} \,⨾\, \underline{((h \times g) \times f \to (g \,⨾\, h) \times f)^{\uparrow L} \,⨾\, (k \times f \to f \,⨾\, k)^{\uparrow L}} \\
\text{composition under } {}^{\uparrow L}: \quad & = \text{zip}\left(\text{zip}\,(q \times r) \times p\right) \triangleright ((h \times g) \times f \to f \,⨾\, g \,⨾\, h)^{\uparrow L} \quad .
\end{aligned}$$

To apply the associativity law of `zip`, we use the conversion functions $\varepsilon_{1,23}$ and $\varepsilon_{12,3}$ and write:

$$\begin{aligned}
\left(h \times (g \times f) \to f \,⨾\, g \,⨾\, h\right) &= \varepsilon_{1,23} \,⨾\, (h \times g \times f \to f \,⨾\, g \,⨾\, h) \quad , \\
\left((h \times g) \times f \to f \,⨾\, g \,⨾\, h\right) &= \varepsilon_{12,3} \,⨾\, (h \times g \times f \to f \,⨾\, g \,⨾\, h) \quad .
\end{aligned}$$

Using these equations, we show that the two sides of the associativity law of $\odot$ are equal:

$$\begin{aligned}
\text{left-hand side}: \quad & \underline{\text{zip}\left(r \times \text{zip}\,(q \times p)\right) \triangleright \varepsilon_{1,23}} \,⨾\, (h \times g \times f \to f \,⨾\, g \,⨾\, h)^{\uparrow L} \\
\text{right-hand side}: \quad & = \underline{\text{zip}\left(\text{zip}\,(q \times r) \times p\right) \triangleright \varepsilon_{12,3}} \,⨾\, (h \times g \times f \to f \,⨾\, g \,⨾\, h) \quad .
\end{aligned}$$

The underlined expessions are equal due to the associativity law of `zip`. □

The laws of the composition operation ($\odot$) show that values of type $L^{A \to B}$ are indeed morphisms. Let us call "L-applicative" the category with those morphisms. We may now write the laws of a (categorical) functor between the L-applicative and the L-lifted categories. Such a functor consists of two mappings: a mapping between objects: each type A is mapped to the type L^A; and a mapping between morphisms: each L-applicative morphism $f : L^{A \to B}$ is mapped to the morphism $\text{ap}\,(f) : L^A \to L^B$ in the L-lifted category. In other words, we expect the function `ap` to play the role of the morphism mapping of that functor. This mapping must obey the laws of identity and composition.

The identity law says that the function `ap` must map the identity morphism of the L-applicative category into the identity morphism of the L-lifted category:

$$\text{left identity law of ap}: \quad \text{ap}\,(\text{pu}_L(\text{id}^{:A \to A})) = \text{id}^{:L^A \to L^A} \quad . \tag{11.15}$$

The composition law says that the composition $p \odot q$ of any two L-applicative morphisms $p^{:L^{A \to B}}$ and $q^{:L^{B \to C}}$ must be mapped by `ap` into the composition $\text{ap}\,(p) \,⨾\, \text{ap}\,(q)$:

$$\text{composition law of ap}: \quad \text{ap}\left(p^{:L^{A \to B}} \odot q^{:L^{B \to C}}\right) = \text{ap}\,(p) \,⨾\, \text{ap}\,(q) \quad . \tag{11.16}$$

In this law, the operation $\odot$ needs to be defined as in Statement 11.5.4.1(b). Expressing that operation via `ap`, we obtain the following formulation of the composition law of `ap`:

$$\text{ap}\left(\text{ap}\left(q^{:L^{B \to C}} \triangleright (g^{:B \to C} \to f^{:A \to B} \to f \,⨾\, g)^{\uparrow L}\right)(p^{:L^{A \to B}})\right) = \text{ap}\,(p) \,⨾\, \text{ap}\,(q) \quad .$$

Instead of proving this complicated law, we will prove the functor laws of `ap`:

Statement 11.5.4.2 Given a functor L with lawful `zip` and `pure` methods, we define the `ap` method as in Statement 11.3.1.3:

$$\text{ap}\,(r^{:L^{A \to B}})(p^{:L^A}) \stackrel{\text{def}}{=} \text{zip}\,(r \times p) \triangleright \text{eval}^{\uparrow L} = (r \times p) \triangleright \text{zip} \,⨾\, (f^{:A \to B} \times a^{:A} \to f(a))^{\uparrow L} \quad .$$

The operation $\odot$ is defined by Statement 11.5.4.1(b). Then the `ap` method satisfies Eqs. (11.15)–(11.16).

Proof To verify the identity law (11.15), apply both sides to some $p^{:L^A}$:

$$\begin{aligned}
\text{expect to equal } p: \quad & \text{ap}\,(\text{pu}_L(\text{id}))(p^{:L^A}) = (\underline{\text{pu}_L(\text{id})} \times p) \triangleright \underline{\text{zip}} \,⨾\, \text{eval}^{\uparrow L}(f^{:A \to A} \times a^{:A} \to f(a))^{\uparrow L} \\
\text{left identity law of zip}: \quad & = p \triangleright \underline{(a^{:A} \to \text{id} \times a)^{\uparrow L} \,⨾\, (f^{:A \to A} \times a^{:A} \to f(a))^{\uparrow L}} \\
\text{composition under } {}^{\uparrow L}: \quad & = p \triangleright (a^{:A} \to \text{id}\,(a))^{\uparrow L} = p \triangleright (a \to a)^{\uparrow L} = p \triangleright \text{id} = p \quad .
\end{aligned}$$

To verify the composition law (11.16), apply both sides to an arbitrary value $r^{:L^A}$:

$$\begin{aligned}
\text{left-hand side}: \quad & r \triangleright \text{ap}\left(p^{:L^{A\to B}} \odot q^{:L^{B\to C}}\right) = \text{ap}\,(p \odot q)(r) = ((p \odot q) \times r) \triangleright \text{zip} ⨾ \text{eval}^{\uparrow L} \\
\text{definition (b) of } \odot: \quad & = \left(\left(\text{zip}\,(q \times p) \triangleright \underline{(h \times g \to g ⨾ h)^{\uparrow L}}\right) \times r\right) \triangleright \underline{\text{zip}} ⨾ \text{eval}^{\uparrow L} \\
\text{naturality law of zip}: \quad & = \left(\text{zip}\,(q \times p) \times r\right) \triangleright \text{zip} \triangleright \underline{((h \times g) \times a \to (g ⨾ h) \times a)^{\uparrow L} ⨾ \text{eval}^{\uparrow L}} \\
\text{composition under } {}^{\uparrow L}: \quad & = \text{zip}\left(\text{zip}\,(q \times p) \times r\right) \triangleright ((h \times g) \times a \to a \triangleright g ⨾ h)^{\uparrow L} \quad .
\end{aligned}$$

Now rewrite the right-hand side of Eq. (11.16) applied to r:

$$\begin{aligned}
\text{right-hand side}: \quad & r \triangleright \text{ap}\,(p) ⨾ \text{ap}\,(q) = \text{ap}\,(q)\left(\text{ap}\,(p)(r)\right) = \text{zip}\left(q \times \text{ap}\,(p)(r)\right) \triangleright \text{eval}^{\uparrow L} \\
& = \underline{\text{zip}}\left(q \times \left(\text{zip}\,(p \times r) \triangleright \underline{\text{eval}^{\uparrow L}}\right)\right) \triangleright \text{eval}^{\uparrow L} \\
\text{naturality law of zip}: \quad & = \text{zip}\left(q \times \text{zip}\,(p \times r)\right) \triangleright \underline{(h \times (g \times a) \to h \times g(a))^{\uparrow L} ⨾ \text{eval}^{\uparrow L}} \\
\text{composition under } {}^{\uparrow L}: \quad & = \text{zip}\left(q \times \text{zip}\,(p \times r)\right) \triangleright \left(h \times (g \times a) \to h(g(a))\right)^{\uparrow L} \quad .
\end{aligned}$$

The remaining difference is reduced to the associativity law of `zip`:

$$\begin{aligned}
\text{left-hand side}: \quad & \underline{\text{zip}\left(\text{zip}\,(q \times p) \times r\right) \triangleright \varepsilon_{12,3}} ⨾ (h \times g \times a \to a \triangleright g \triangleright h)^{\uparrow L} \\
\text{right-hand side}: \quad & = \underline{\text{zip}\left(q \times \text{zip}\,(p \times r)\right) \triangleright \varepsilon_{1,23}} ⨾ \left(h \times g \times a \to a \triangleright g \triangleright h\right)^{\uparrow L} \quad .
\end{aligned}$$

The underlined expessions are equal due to the associativity law (11.3) of `zip`. □

It is important to use the definition (b) of the operation $\odot$ from Statement 11.5.4.1. The definition (a) does create a valid category but describes a reversed order of effects and cannot be used to write the law of `ap` in the form of a (categorical) functor composition law. So, we will use the definition (b) for the category we call "L-applicative". Statements 11.5.4.1–11.5.4.2 show that the laws of `ap`, viewed as the laws of the L-applicative category together with the laws of a functor from the L-applicative to the L-lifted category, are a consequence of the laws of `zip`.

In this way, we have used the guidance of category theory to formulate the laws of `ap`.

The proofs of Statements 11.5.4.1–11.5.4.2 use the left identity law of `zip` but not the right identity law. That law is equivalent to an additional law of `ap` (see Exercise 11.5.6.8).

11.5.5 The pattern of "functorial" typeclasses in view of category theory

In the previous chapters, we have derived several equivalent formulations of the laws of various typeclasses (such as functor, contrafunctor, filterable, monad, applicative). We found that some of these formulations are easier to use when verifying the laws by hand. However, in every case we have found a certain formulation of the typeclass laws in terms of the laws of a "lifting" (namely, the identity and composition laws). This formulation is important because it makes contact with category theory, which provides assurance that the laws are chosen reasonably and correctly. Let us now summarize what we have learned about the laws of the typeclasses.

In many cases considered so far, we were able to find a certain typeclass method whose type signature looks like a "lifting" between functions of one sort and functions of another sort. For instance, the `Filterable` typeclass has the `liftOpt` method (Section 9.2.3) with type signature

$$\text{liftOpt} : (A \to \mathbb{1} + B) \to (F^A \to F^B) \quad .$$

Here, we wrote the optional parentheses around $(F^A \to F^B)$ to emphasize that `liftOpt` maps from one sort of functions to another. Category theory generalizes functions (with type $A \to B$) to "morphisms", which in most cases are functions with a modified type signature (such as $A \to \mathbb{1} + B$, or $F^A \to F^B$, or something else). Table 11.1 lists the type signatures of the "lifting" functions and the

Typeclass	Type signature of the "lifting"	Morphism type
functor	$\text{fmap} : (A \to B) \to (F^A \to F^B)$	$A \to B$
filterable	$\text{liftOpt} : (A \to \mathbb{1} + B) \to (F^A \to F^B)$	$A \to \mathbb{1} + B$
monad	$\text{flm} : \left(A \to F^B\right) \to (F^A \to F^B)$	$A \to F^B$
applicative	$\text{ap} : F^{A \to B} \to (F^A \to F^B)$	$F^{A \to B}$
contrafunctor	$\text{cmap} : (B \to A) \to (F^A \to F^B)$	$B \to A$
profunctor	$\text{xmap} : (A \to B) \times (B \to A) \to (F^A \to F^B)$	$(A \to B) \times (B \to A)$
contrafilterable	$\text{liftOpt} : (B \to \mathbb{1} + A) \to (F^A \to F^B)$	$B \to \mathbb{1} + A$
comonad	$\text{coflm} : \left(F^A \to B\right) \to (F^A \to F^B)$	$F^A \to B$

Table 11.1: Some typeclasses that follow the functorial pattern.

corresponding morphism types for some typeclasses. We see that in each case the "lifting" method maps functions of a selected morphism type to functions of type $F^A \to F^B$.

We call **functorial** all typeclasses that follow this pattern. The word "functorial" will remind us that there exists a certain (categorical) functor at the core of the typeclass. Having defined that functor, we can derive the methods and the laws of the typeclass via the following steps:

1. Define each category's composition operation and identity morphism.
2. Verify that the category laws hold.
3. Write the type signature of the "lifting" method corresponding to the functor.
4. Impose the naturality laws and the functor laws on the "lifting" method.
5. Derive other typeclass methods (and their laws) from the "lifting" method and its laws.

Let us go through this pattern for the applicative functor typeclass that we studied in this chapter. Following the functorial pattern, we say that a covariant type constructor $F^\bullet$ is applicative if there exists a (categorical) functor between the F-applicative and F-lifted categories. The morphisms in the F-applicative category are values of type $F^{A \to B}$; the morphisms in the F-lifted category are values of type $F^A \to F^B$. This is all we need to start with. All other properties of applicative functors are then derived in a systematic way. With this approach, we do not need to memorize the complicated type signatures and laws of the methods `map2` and `ap`.

The first step is to define the the F-applicative category. We define identity morphisms (`wid[A]` of type $F^{A \to A}$) and the composition operation ($\odot$) that composes $F^{A \to B}$ with $F^{B \to C}$ to obtain $F^{A \to C}$. The second step is to verify that the category laws hold with these definitions; this is done in Statement 11.5.4.1. The F-lifted category with morphisms $F^A \to F^B$ is shared by all functorial typeclasses we have seen, and we already checked that it satisfies the category laws (Section 9.4.3). The third step is to write the type signature of the "lifting" function of the functor:

$$\text{ap} : F^{A \to B} \to F^A \to F^B \quad .$$

The fourth step is to require that `ap` obey the naturality and the functor laws. The last step is to derive other methods that are equivalent to `ap` but more convenient to use. We have shown that the `map2` and `zip` methods with suitable laws are equivalent to `ap` (see Section 11.3.1 and Statement 11.5.4.2).

In this way, we demonstrate that the applicative typeclass follows the functorial typeclass pattern.

Many typeclasses can be derived from this pattern, but some cannot. For example, applicative contrafunctors and profunctors (Section 11.4) or traversable functors (Chapter 12) are not functorial typeclasses. Their laws must be motivated by other considerations.

11.5.6 Exercises

Exercise 11.5.6.1 Implement an applicative instance for $F^A = \mathbb{1} + \text{Int} \times A + A \times A \times A$.

Exercise 11.5.6.2 Show that the following functors $F^\bullet$ *cannot* be applicative:

(a) $F^A \stackrel{\text{def}}{=} (A \to P) \to Q$.

(b) $F^A \stackrel{\text{def}}{=} (A \to P) \to \mathbb{1} + A$.

Here P and Q are some fixed monoidal types that are not related in any way to each other.

Exercise 11.5.6.3 Show that the ternary tree functor from Exercise 11.2.9.2 is a lawful commutative applicative functor. Use applicative functor constructions to avoid long proofs.

Exercise 11.5.6.4 Show that $F^A \stackrel{\text{def}}{=} G^A + H^{G^A}$ is a lawful applicative functor if G and H are. Show that F is commutative if G and H are. Use applicative functor constructions to avoid long proofs.

Exercise 11.5.6.5 For an arbitrary applicative functor F with a lawful `zip` method, prove this law:

$$\text{zip}_F\left(\text{pu}_F(a^{:A}) \times \text{pu}_F(b^{:B})\right) = \text{pu}_F(a \times b) \quad .$$

Exercise 11.5.6.6 It is given that two functions $u : A \to F^B$ and $v : A \to F^B$ (where F is some functor) satisfy the law $u \fatsemi f^{\uparrow F} = v \fatsemi f^{\uparrow F}$ with an *arbitrary* function $f : B \to C$. Show that the function f can be omitted from the law: an equivalent law is simply $u = v$.

Exercise 11.5.6.7 Show that $\text{pu}_L(f) \odot \text{pu}_L(g) = \text{pu}_L\,(f \fatsemi g)$ for an applicative functor L. The operation $\odot$ is defined in Statement 11.5.4.1(b).

Exercise 11.5.6.8 Show that if `ap` is defined via a lawful `zip` then `ap` satisfies the following law:

$$\text{right identity law of ap}_L : \quad \text{ap}_L\,(r^{:L^{A\to B}})(\text{pu}_L(a^{:A})) = r \triangleright (f^{:A\to B} \to f(a))^{\uparrow L} \quad .$$

Exercise 11.5.6.9 Show that P^S is a monoid if S is a fixed monoidal type and $P^\bullet$ is any applicative functor, contrafunctor, or profunctor.

Exercise 11.5.6.10 Prove the following statements (which complement Statement 11.3.6.1):

(a) If $F^\bullet$ is an applicative functor and $G^\bullet$ is an applicative contrafunctor then the contrafunctor $L^A \stackrel{\text{def}}{=} G^{F^A}$ is applicative.

(b) If $F^\bullet$ and $G^\bullet$ are both applicative contrafunctors then $L^A \stackrel{\text{def}}{=} F^{G^A}$ is an applicative *functor*.

(c) In both parts **(a)** and **(b)**, if $F^\bullet$ and $G^\bullet$ are commutative then $L^\bullet$ is also commutative.

Exercise 11.5.6.11 Show that $F^A \stackrel{\text{def}}{=} H^A \times G^A$ is applicative and co-pointed if G and H are applicative functors and H is co-pointed. Show that the compatibility law (11.10) holds for zip_F and ex_F if it holds for zip_H and ex_H.

Exercise 11.5.6.12 Explicitly prove the laws in contrafunctor product construction (Statement 11.4.1.3).

Exercise 11.5.6.13 Show that the recursive functor $F^A \stackrel{\text{def}}{=} \mathbb{1} + G^{A\times F^A}$ is applicative if G^A is applicative and wu_F is defined recursively as $\text{wu}_F \stackrel{\text{def}}{=} \mathbb{0} + \text{pu}_G\,(1 \times \text{wu}_F)$. Use applicative functor constructions.

Exercise 11.5.6.14 Prove Statement 11.4.2.3.

Exercise 11.5.6.15 The type constructor $Q^\bullet$ is defined by

$$Q^A \stackrel{\text{def}}{=} (A \to \text{Int}) \times A \times (A \to A) \quad .$$

Show that Q^A is neither covariant nor contravariant in A, and express Q^A via a profunctor.

Exercise 11.5.6.16 **(a)** For any given profunctor P^A, implement a function of type $A \times P^B \to P^{A\times B}$.

(b) Show that one *cannot* implement a function of type $A \times P^B \to P^A$ for some profunctors P^A.

Exercise 11.5.6.17 Implement profunctor and applicative instances for $P^A \stackrel{\text{def}}{=} A + Z \times G^A$ where G^A is a given applicative profunctor and Z is a monoid.

Exercise 11.5.6.18 For the profunctor $P^A \stackrel{\text{def}}{=} A \to A$:

(a) Show that P^A is pointed: there exist a value `wu` of type $P^\mathbb{1}$ and a method `pure` of type $A \to P^A$.

(b) Show that the type of fully parametric functions $\text{pu}_P : A \to P^A$ is *not* equivalent to the type $P^\mathbb{1}$.

12 Traversable functors

12.1 Motivation

The `map`/`reduce` style is a major industrial use case of functional programming. Previous chapters examined systematically the properties of functions used in that style (`map`, `filter`, `flatMap`, and `zip`) and generalized them to many different type constructors. This chapter adopts the same approach to study and generalize the `reduce` method. In this way, we will obtain a complete understanding of the `map`/`reduce` programming style.

12.1.1 From `reduce` and `foldLeft` to `foldMap` and `traverse`

The type signatures of `reduce` and `foldLeft` for a sequence of type `Seq[A]` can be written as:

```
def reduce[A](xs: Seq[A])(update: (A, A) => A): A
def foldLeft[A, B](xs: Seq[A])(b0: B)(update: (B, A) => B): B
```

$$\text{reduce} : \text{Seq}^A \to (A \times A \to A) \to A \quad , \qquad \text{foldLeft} : \text{Seq}^A \to B \to (B \times A \to B) \to B \quad .$$

In `foldLeft`, the accumulated result is of a different type (B) than the type of sequence elements (A). So, `foldLeft` is a more general version of `reduce`. We can implement `reduce` through `foldLeft`:

```
def reduce[A](xs: Seq[A])(update: (A, A) => A): A = xs.tail.foldLeft(xs.head)(update)
```

The `foldLeft` operation goes over each element of a sequence and accumulates the updates, returning the final updated value. We have seen many examples of using `foldLeft` in Chapter 2. Now we would like to generalize `foldLeft` to data structures other than sequences. Such a generalization must be based on imposing suitable laws, but it is not obvious what laws `foldLeft` must satisfy (other than naturality laws that will hold automatically for any fully parametric code). To make progress, we will generalize `foldLeft` further to obtain a new function called `traverse`, for which we can motivate non-trivial laws. We will achieve this generalization in two steps, first transforming `foldLeft` to a function called `foldMap` and then to `traverse`.

The way from `foldLeft` to `foldMap` starts by changing the type of `update: (B, A) => B` to an equivalent curried form with a flipped order of arguments:

$$\text{upd} : A \to B \to B \quad .$$

So, upd $(x^{:A})$ has type $B \to B$. The key observation is that the set of all functions of type $B \to B$ is a monoid (denoted by `monoidFunc` in Example 8.2.5.6 and Section 8.3.4). For brevity, we will here denote that monoid by `MF`. The empty element of that monoid is the identity function ($e_{\text{MF}} \stackrel{\text{def}}{=} \text{id}^{:B\to B}$), and the `compose` operation is the function composition ($f^{:B\to B} \oplus_{\text{MF}} g^{:B\to B} \stackrel{\text{def}}{=} f \fatsemi g$). Now the evaluation of a `foldLeft` operation can be visualized like this:

$$\begin{aligned}&\text{foldLeft}\left([x_1, x_2, x_3]\right)(b_0)\left(\text{upd}\right) = b_0 \triangleright \text{upd}\,(x_1) \triangleright \text{upd}\,(x_2) \triangleright \text{upd}\,(x_3)\\ &= b_0 \triangleright \text{upd}\,(x_1) \fatsemi \text{upd}\,(x_2) \fatsemi \text{upd}\,(x_3) = b_0 \triangleright \left(\text{upd}\,(x_1) \oplus_{\text{MF}} \text{upd}\,(x_2) \oplus_{\text{MF}} \text{upd}\,(x_3)\right) \quad .\end{aligned}$$

This formulation suggests that we could replace the specific monoid (`MF`) used in `foldLeft` by an arbitrary monoid (M). The new, more general function is called `foldMap`. To obtain that function, we first change the order of the curried arguments for convenience:

$$\text{foldMap}\,(\text{upd})(\text{xs})\,(b_0) \stackrel{\text{def}}{=} \text{foldLeft}\,(\text{xs})\,(b_0)\,(\text{upd}) \quad .$$

Omitting the argument b_0, we may visualize the computation performed by `foldMap` as:

$$\text{foldMap}\,(\text{upd})\,([x_1, x_2, x_3]) = \text{upd}\,(x_1) \oplus_M \text{upd}\,(x_2) \oplus_M \text{upd}\,(x_3) \quad .$$

The type of `upd` is now $A \to M$ instead of $A \to B \to B$. So, the type signature of `foldMap` must be:

$$\text{foldMap} : (A \to M) \to \text{Seq}^A \to M \quad .$$

This type signature assumes that M is a monoid type. We can now implement `foldMap` via `foldLeft`:

```
def foldMap[M: Monoid, A](f: A => M): Seq[A] => M =
  _.foldLeft(Monoid[M].empty) { (m, a) => m |+| f(a) }
```

We can also implement `foldLeft` via `foldMap` by using the monoid instance of `MF` (see Example 8.2.5.6):

```
implicit def monoidFunc[A]: Monoid[A => A] = ...
def foldLeft[A, B](xs: Seq[A])(b0: B)(update: (B, A) => B): B =
  foldMap[B => B, A](a => b => update(b, a))(xs)(b0)
```

We will prove later that `reduce`, `foldLeft`, and `foldMap` are in fact equivalent.

The type signature of `foldMap` suggests that we could replace `Seq[_]` by an arbitrary type constructor `L[_]`. We call **foldable** a type constructor `L[_]` for which the `foldMap` method is defined:

```
def foldMap_L[M: Monoid, A](f: A => M): L[A] => M
```

$$\text{foldMap}_L : (A \to M) \to L^A \to M \quad , \quad \text{assuming that } M \text{ is a monoid} \quad .$$

The function foldMap_L works in the same way for all monoids M (but differently for each L).

The final step towards the `traverse` method is to replace an arbitrary monoid M by an arbitrary *applicative* functor F^B with a new type parameter B. This step is not straightforward: values of applicative functor types are not monoids (we do not necessarily have an operation of type $F^B \times F^B \to F^B$). So, we will give an independent motivation for the `traverse` method in the next subsection.

12.1.2 The `traverse` operation

Consider the task of waiting for several concurrent operations to finish. As an example, suppose we have a list of type `List[A]` and a processing function of type `A => Future[B]`. We need to apply the processing function in parallel to all items from the list and wait until the entire data set is processed.

The Scala standard library has a special method (`Future.traverse`) to use in this situation:

```
val data: List[A] = ???
val processing: A => Future[B] = ???
implicit val ec: ExecutionContext = ???
val results: Future[List[B]] = Future.traverse(data)(processing)
```

Since the processing of each element is asynchronous, the result is wrapped in a `Future` type constructor. The `traverse` method will mark that `Future` value as finished when all processing is done.

If we ignore the implicit argument of type `ExecutionContext` (which is specific to the `Future` class) and also flip the first two curried arguments, we will arrive at the following type signature of `traverse`:

$$\text{traverse} : (A \to \text{Future}^B) \to \text{List}^A \to \text{Future}^{\text{List}^B} \quad .$$

Generalizing to other type constructors F, L instead of `Future` and `List`, we obtain:

```
def traverse[A, B, F[_]: Applicative: Functor](f: A => F[B]): L[A] => F[L[B]]
```

$$\text{trav}_L^{F,A} : (A \to F^B) \to L^A \to F^{L^B} \quad .$$

In this type signature, we have chosen the parameter names F and L specifically to help us avoid errors (e.g., writing L^{F^B} instead of F^{L^B}) by recalling the example with $F = \texttt{Future}$ and $L = \texttt{List}$.

It turns out that the `traverse` method can work in the same way for any *applicative* functor F. A functor L having a `traverse` method of this type is called **traversable**.

The type signatures of `traverse` and `foldMap` are similar except that M is replaced by F^B or F^{L^B} as needed. To see that `foldMap` is a special case of `traverse`, take a constant functor $F^A \stackrel{\text{def}}{=} M$, which is applicative if M is a monoid (as shown in Section 11.3.6). With this choice of F, the type signature of `traverse` reduces to that of `foldMap`.

Before studying the properties of `traverse`, we will look at some examples of its practical use.

12.2 Practical use of folding and traversing operations

12.2.1 Implementing `traverse` for various data types

To understand how `traverse` works, let us implement it for some data types, including lists and trees.

Example 12.2.1.1 Implement `traverse` for the type constructor L defined by:

```
type L[A] = Option[(A, A)]
```

$$L^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A \quad .$$

Solution For the given type constructor L, write the type signature of `trav` as:

```
def trav[A, B, F[_]: Applicative : Functor](f: A => F[B]): Option[(A, A)] => F[Option[(B, B)]]
```

In the short type notation, this type signature is:

$$\text{trav}_L : (A \to F^B) \to (\mathbb{1} + A \times A) \to F^{\mathbb{1}+B\times B} \quad .$$

Let us implement `trav`, aiming to preserve information. We need to compute a value of type $F^{\mathbb{1}+B\times B}$. Begin by pattern matching on the argument of type $\mathbb{1} + A \times A$:

```
def trav[A, B, F[_]: Applicative : Functor](f: A => F[B]): Option[(A, A)] => F[Option[(B, B)]] = {
  case None             => Applicative[F].pure(None)  // No other choice here.
  case Some((a1, a2))   => ???                        // Need a value of type F[Some[(B, B)]] here.
}
```

The only way of getting values of type B is by applying `f` to the values `a1` and `a2`. We will then get two values of type F^B. We may merge them into a single value of type $F^{B\times B}$ via F's `zip` method. It remains to use F's `map` in order to transform $F^{B\times B}$ to the required type $F^{\mathbb{1}+B\times B}$:

```
def trav[A, B, F[_]: Applicative : Functor](f: A => F[B]): Option[(A, A)] => F[Option[(B, B)]] = {
  case None             => Applicative[F].pure(None)          // No other choice here.
  case Some((a1, a2))   => Applicative[F].zip(f(a1), f(a2)).map { case (b1, b2)  => Some((b1, b2)) }
}
```

In the code notation, this is written as:

$$\text{trav}_L(f^{:A\to F^B}) \stackrel{\text{def}}{=} \left\| \begin{array}{c|c} & F^{\mathbb{1}+B\times B} \\ \hline \mathbb{1} & _ \to \text{pu}_F(1 + \mathbb{0}^{:B\times B}) \\ A \times A & (f \boxtimes f) ; \text{zip}_F ; (b_1^{:B} \times b_2^{:B} \to \mathbb{0}^{:\mathbb{1}} + b_1 \times b_2)^{\uparrow F} \end{array} \right| \quad .$$

The code must use F's methods `map`, `pure`, and `zip`. This makes it clear why `traverse` requires F to be an applicative functor. Apart from that requirement, the code works in the same way for all F.

Example 12.2.1.2 Implement `traverse` for Scala's `List` type constructor.

Solution We need to implement the following type signature:

```
def trav[A, B, F[_]: Applicative : Functor](f: A => F[B]): List[A] => F[List[B]]
```

A list may be empty, or may have a head and a tail. Similarly to the code in Example 12.2.1.1, we use F's methods `pure`, `zip`, and `map` to compute a suitable value of type `F[Seq[B]]` in these cases.

```
def trav[A, B, F[_]: Applicative : Functor](f: A => F[B])(la: List[A]): F[List[B]] = la match {
  case Nil            => Applicative[F].pure(Nil)
  case head :: tail   => (f(head) zip trav(f)(tail)).map { case (headB, tailB)  => headB :: tailB }
}
```

This code applies `f` to each element of the list and accumulates the resulting F-effects of type F^B. The F-effects are merged using F's `zip` function (implemented as an extension method).

In the code notation, the function `trav` looks like this:

$$\text{trav}_{\text{List}}(f^{:A\to F^B}) \stackrel{\text{def}}{=} \left|\begin{array}{c|c} & F^{\mathbb{1}+B\times\text{List}^B} \\ \hline \mathbb{1} & _ \to \text{pu}_F(1+\mathbb{0}^{:B\times\text{List}^B}) \\ A\times\text{List}^A & (f\boxtimes\overline{\text{trav}}_{\text{List}}(f))\,;\text{zip}_F\,;(h^{:B}\times t^{:\text{List}^B}\to\mathbb{0}^{:\mathbb{1}}+h\times t)^{\uparrow F} \end{array}\right| .$$

Example 12.2.1.3 Implement a `traverse` operation for a binary tree.

Solution We will use the following binary tree data type denoted by `T2`:

```
sealed trait T2[A]
final case class Leaf[A](a: A)                 extends T2[A]
final case class Branch[A](l: T2[A], r: T2[A]) extends T2[A]
```

We implement `trav` similarly to the implementation of `foldLeft` in Section 3.3.3:

```
def trav[A, B, F[_]: Applicative : Functor](f: A => F[B])(t: T2[A]): F[T2[B]] = t match {
  case Leaf(a)          => f(a).map(b => Leaf(b))            // Reproduce the Leaf structure under F.
  case Branch(t1, t2)   =>
    val (r1, r2) = (trav(f)(t1), trav(f)(t2))    // Traverse the two branches and obtain two results.
    (r1 zip r2).map { case (b1, b2)  => Branch(b1, b2) }   // Reproduce the Branch structure under F.
}
```

In the code notation, this function looks like this:

$$\text{trav}\,(f^{:A\to F^B}) \stackrel{\text{def}}{=} \left|\begin{array}{c|c} & F^{B+\text{T2}^B\times\text{T2}^B} \\ \hline A & f\,;(b^{:B}\to b+\mathbb{0}^{:\text{T2}^B\times\text{T2}^B})^{\uparrow F} \\ \text{T2}^A\times\text{T2}^A & (\overline{\text{trav}}\,(f)\boxtimes\overline{\text{trav}}\,(f))\,;\text{zip}_F\,;(l^{:\text{T2}^B}\times r^{:\text{T2}^B}\to\mathbb{0}^{:B}+l\times r)^{\uparrow F} \end{array}\right| .$$

The code first traverses the two sub-branches (using recursive calls to `trav`) and then combines the resulting values. So, this implementation represents a depth-first, left-to-right traversal of the tree.

Example 12.2.1.4 Implement a `traverse` operation for a perfect-shaped tree (`PTree`, Section 3.3.5).

Solution We begin writing code as in Example 12.2.1.3 by pattern matching:

```
1 def trav[A, B, F[_]: Applicative : Functor](f: A => F[B])(t: PTree[A]): F[PTree[B]] = t match {
2   case Leaf(a)     => f(a).map(b => Leaf(b))          // Reproduce the Leaf structure under F.
3   case Branch(t)   => ???  // Here t has type PTree[(A, A)].
4 }
```

In line 3, we have a pattern variable `t` of type `PTree[(A, A)]`, but we need to obtain a value of type `F[Branch[B]]` in that scope. Since a `Branch` contains a recursive instance of the type `PTree`, it seems we need to use a recursive call of `trav` here. However, we cannot apply `trav(f)` to `t` because the types do not match. We will be able to apply `trav` if instead of `f: A => F[B]` we had a function of type `(A, A) => F[(B, B)]`. We can create such a function out of `f` using F's `zip` method:

```
{ case (a1, a2) => f(a1) zip f(a2) }
```

So, we write:

```
  case Branch(t)   => trav { case (a1, a2) => f(a1) zip f(a2) }(t); ??? } // Have F[PTree[(B, B)]].
```

We can now use F's `map` method to restore the `Branch` structure under F. The complete code is:

```
def trav[A, B, F[_]: Applicative : Functor](f: A => F[B])(t: PTree[A]): F[PTree[B]] = t match {
  case Leaf(a)     => f(a).map(b => Leaf(b))          // Reproduce the Leaf structure under F.
  case Branch(t)   => (trav[(A, A), (B, B), F] { case (a1, a2) => f(a1) zip f(a2) }(t)
                      ).map(x => Branch(x))          // Reproduce the Branch structure under F.
}
```

Here we assumed that the `zip` function is defined on `F` as an extension method. □

These examples show how to implement the `traverse` operation for a container-like data structure L^A that stores some values of type A. Each stored value of type A is processed using the given function $f : A \to F^B$. All of the resulting F-effects need to be collected and merged (using F's `zip` and `map`) into a single F-effect that wraps a value of type L^B. The wrapped value should reproduce the shape of the data structure that was present in the input value of type L^A.

In all examples above, we merged all F-effects together by collecting each F-effect exactly once. It would be strange if the `traverse` operation repeated some F-effects, as in the following code:

```
def badtrav[A, B, F[_]: Applicative : Functor](f: A => F[B])(la: List[A]): F[List[B]] = la match {
  case Nil            => Applicative[F].pure(Nil)
  case head :: tail   => (f(head) zip f(head) zip badtrav(f)(tail))     // Use the F-effect twice.
                           .map { case (headB, (_, tailB))  => headB :: tailB }
}
```

Although this implementation fits the type signature of `traverse`, it contradicts the intuition of traversing the sequence only once. When F = `Future`, the code `badtrav(f)(xs)` will start *two* parallel computations for each element of the sequence `xs` and ignore one of the results. When F's effect describes parsing (see Section 11.2.7) or other computations that maintain an internal state, invoking such an F-effect twice will likely lead to incorrect results. Below we will see that the laws of `traverse` prohibit implementations that either repeat some of the F-effects or ignore some of the values.

12.2.2 Aggregating tree-like data by folding. Breadth-first traversal

Although `foldMap` (and `foldLeft`) are special cases of `traverse`, it is often easier to implement `foldMap` directly. Let us look at some examples of `foldMap` that implements different kinds of tree traversals.

One implementation of `foldMap` corresponds to the `traverse` operation defined in Example 12.2.1.3. The code of `foldMap` is:

```
def foldMap[A, M: Monoid](f: A => M)(t: T2[A]): M = t match {
  case Leaf(a)          => f(a)
  case Branch(t1, t2)   => foldMap(f)(t1) |+| foldMap(f)(t2)
}
```

A special case of a monoid is `List[A]`; its `combine` operation is the concatenation of lists. If we use `foldMap` with the monoid `M = List[A]` and the function `f: A => List[A]` that creates a single-element list, we obtain a function `toList` that converts a tree `T2[A]` into `List[A]`:

```
def toList[A]: T2[A] => List[A] = foldMap[A, List[A]](List(_)) // Need a Monoid instance for List[A].
```

In the same way, we may implement `toList: L[A] => List[A]` for any foldable functor `L[_]`.

The function `toList` captures the requirement that a foldable functor `L` must have a well-defined way of iterating over the data items stored inside it. So, `L` is foldable if there is a known way of extracting its data items in the form of a list. We may then use standard aggregation operations on lists (`foldLeft`, `reduce`, and so on). Below, we will prove that `toList` is equivalent to `foldMap`. So, a foldable functor allows us to apply any aggregation operation to the data stored in the functor.

An aggregation's results may depend on the order of data traversal. It is important that the function `toList` can be implemented in different ways by choosing a specific order in which the data from `L` is copied to a list. Some data structures have a "natural" traversal order that is easiest to implement. Let us see some examples of implementing different traversal orders for lists and trees.

The above code of `foldMap` for the binary tree implements a depth-first traversal of the tree. We can see that by applying `toList` to an example tree (`t2`):

```
val t2: T2[Int] = Branch(Leaf(8), Branch(Branch(Leaf(3), Leaf(5)), Leaf(4)))

scala> toList(t2)
res0: List[Int] = List(8, 3, 5, 4)
```

$= [8, 3, 5, 4]$

Let us now implement another version of `toList`, called `toList2`, that performs the *breadth-first* traversal of a binary tree. The idea is to prepare a list of all leaf values at level 0 in the tree, then a list of all values at level 1, and so on. At each level, values must be collected left to right. The result will be a list of lists. For example, the tree `t2` shown above has no values at level 0, value 8 at level 1, value 4 at level 2, and values 3 and 5 at level 3. So, applying `toList2` to the tree `t2` will give the nested list $[[],[8],[4],[3,5]]$. Then we can `flatten` that list and obtain the list $[8,4,3,5]$, which is the breadth-first traversal order of the tree `t2`.

Begin writing code of `toList2` by pattern matching on the tree structure:

```
def toListBFS[A]: T2[A] => List[A] = toList2 andThen (_.flatten)

def toList2[A]: T2[A] => List[List[A]] = {
  case Leaf(a)        => List(List(a))    // Only one value at level 0.
  case Branch(l, r)   => ???         // Have toList2(l) and toList2(r).
}
```

After using recursive calls of `toList2` on the left and right sub-trees (`l` and `r`), we need somehow to combine the resulting lists. For the tree `t2`, applying `toList2` to the sub-trees will return the lists $[[8]]$ and $[[],[4],[3,5]]$. We need to concatenate the corresponding nested lists and obtain $[[],[8],[4],[3,5]]$. (An initial empty list needs to be added because the sub-trees begin one level deeper.) The standard `zip` operation would not work correctly here: `zip` would truncate the longer list. Instead, we need to keep the longer list's elements. So, let us implement a special helper function (`listMerge`) for merging the nested lists in this way:

```
def listMerge[A](l: List[List[A]], r: List[List[A]]): List[List[A]] = (l, r) match {
  case (Nil, r)               => r        // Keep the elements from the longer list.
  case (l, Nil)               => l
  case (lh :: lt, rh :: rt)   => (lh ++ rh) :: listMerge(lt, rt)
}
```

Now we can complete the code of `toList2` and run some tests:

```
def toList2[A]: T2[A] => List[List[A]] = {
  case Leaf(a)        => List(List(a)) // Only one value at level 0.
  case Branch(l, r)   => listMerge(Nil :: toList2(l), Nil :: toList2(r))
}

scala> toList2(t2)
res1: List[List[Int]] = List(List(), List(8), List(4), List(3, 5))

scala> toListBFS(t2)
res2: List[Int] = List(8, 4, 3, 5)
```

For lists and other sequences, the easiest traversal orders are either direct or reverse. A direct traversal is implemented by the standard `foldLeft` operation on `Seq`, while a reverse-order traversal is done simply by applying the `reverse` method, such as `la.reverse.foldLeft(...)(...)`.

Similarly to `foldLeft` and `foldMap`, the `traverse` method may be implemented with different traversal orders. For comparison, we show the code for the direct-ordered and reverse-ordered `List` traversals:

```
// This code assumes that F has typeclass instances for Applicative and Functor.
def travList[A, B](f: A => F[B])(la: List[A]): F[List[B]] = la match {
  case Nil            => Applicative[F].pure(Nil)
  case head :: tail   => f(head).map2(travList(f)(tail)) { (x, y) => x +: y }
}

def travRevList[A, B](f: A => F[B])(la: List[A]): F[List[B]] = la.reverse match {
  case Nil            => Applicative[F].pure(Nil)
  case head :: tail   => f(head).map2(travRevList(f)(tail)) { (h, t) => t :+ h }
}
```

The function `travRevList` first reverses the given list, so that F's effects are merged in the reverse order. Then it swaps `h` and `t` using `map2`, which restores the original list structure wrapped under F.

Implementing a breadth-first `traverse` for a tree is more difficult. It is not sufficient to convert the tree to a list in the breadth-first order and collect F's effects. We also need to restore the original tree structure wrapped under the functor F. Section 12.2.4 will show an example of implementing a breadth-first `traverse` for binary trees.

12.2.3 Decorating a tree. I. Depth-first traversal

A `traverse` method for a given tree-like data structure may be used to perform various operations on trees, as long as those operations are described by the effect of some applicative functor F.

An example of a tree operation is "decorating", i.e., adding labels of some sort to each leaf. Since tree-like data types are functors that have a `map` method, a simple decorating operation replaces each leaf with a function of the data at that leaf. For instance, we may add the value 20 to each leaf:

```
val t2 = Branch(Leaf(8), Branch(Branch(Leaf(3), Leaf(5)), Leaf(4)))

scala> t2.map(x => x + 20)  // Assuming a Functor instance for T2[_].
res0: T2[Int] = Branch(Leaf(28), Branch(Branch(Leaf(23), Leaf(25)), Leaf(24)))
```

This transforms the tree

into

. However, the `map` method works separately with each leaf and cannot use any previously computed values. E.g., we cannot use `map` to implement a `zipWithIndex` function for trees, transforming the tree

into

with added indices (0, 1, 2, 3). For such tasks, we need the additional functionality of the `traverse` operation.

Labeling the leaves of a tree with their traversal index requires us to update an internal state (the number of leaves seen so far) while traversing the tree. Recall that updating an internal state is the effect of a `State` monad. Since all monads can also implement the applicative methods (`pure` and `zip`), we may use the `State` monad as the applicative functor F for the `traverse` function.

In this section, we will implement `zipWithIndex` for a *depth-first* tree traversal. Let us call that function `zipWithIndexDF`. The code of `zipWithIndexDF` will have the form:

```
final case class St[A](run: Int => (A, Int))    // A State monad with internal state of type Int.
                        // Assume that we have defined Applicative and Functor instances for St.
def computeIndex[A]: A => St[(A, Int)] = ???                // Define the "decoration" function.
def zipWithIndexDF[A](tree: T2[A]): T2[(A, Int)] = {
  val afterTraverse: St[T2[(A, Int)]] = trav[A, (A, Int), St](computeIndex)(tree)
  afterTraverse.run(0)._1                        // Run the State monad and get the result value.
}
```

This will be a depth-first traversal if `trav` is the function shown in Example 12.2.1.3. It remains to define the function `computeIndex` describing the "decoration logic". Begin writing the code as:

```
def computeIndex[A]: A => St[(A, Int)] = a => St { i => ???: ((A, Int), Int) }
```

Given a leaf value (`a: A`) and a previous internal state (`i: Int`), we must compute the "decorated" leaf value of type $A \times \text{Int}$ as well as the new internal state. Since our goal is to label the leaves by the current leaf count, we use the internal state to store the number of leaves seen so far. So, we just need to increment the internal state after copying it to a leaf:

```
def computeIndex[A]: A => St[(A, Int)] = a => St { i => ((a, i), i + 1) }
```

This completes the implementation of `zipWithIndexDFS`, which we can now test:

```
scala> zipWithIndexDF(t2)
res0: T2[(Int, Int)] = Branch(Leaf((8,0)), Branch(Branch(Leaf((3,1)), Leaf((5,2))), Leaf((4,3))))
```

12.2.4 Decorating a tree. II. Breadth-first traversal

In Section 12.2.2, we have implemented a breadth-first folding for the tree data structure `T2[A]`. What additional code does `traverse` need? Folding over `T2[A]` merely needs to arrange the leaf values of type `A` in the breadth-first order, but a `traverse` function must return a value of type `F[T2[B]]`. This requires us to sequence the effects of an arbitrary applicative functor `F[_]` in the breadth-first order, while gathering the values of type `B` into a tree structure (`T2[B]`) wrapped under `F`. The function `toListBFS` shown in the previous section is not sufficient for that purpose, because the tree structure cannot be reproduced if we only have a list of leaf values. Even the nested list computed by `toList2` is not sufficient. We need additional information about each leaf's position in the tree.

So, let us begin by adding the required position information to the nested list of leaf values. For each leaf, we need to store the path from the root of the tree. We can use the `Either` type constructor and its subtypes `Left` and `Right` for marking that path. For example, the position information for the tree `t2 =` could be represented by the following nested list:


```
List(
  List(),                                                  // No leaves at level 0.
  List( Left(8) ),                                         // One leaf at level 1.
  List( Right(Right(4)) ),                                 // One leaf at level 2.
  List( Right(Left(Left(3))), Right(Left(Right(5))) ),     // Two leaves at level 3.
)
```

This data structure makes it easy to iterate over the leaf values in the breadth-first order. At the same time, it keeps enough information for restoring the original tree since each leaf value comes with the full path from the root of the tree.

An immediate problem with this data structure is a type mismatch in the outer `List`: its elements are not of the same type because each subsequent sub-list contains data wrapped under `Either` more deeply. To make the types match, we need to implement a special list-like data structure whose first element has type `List[A]`, the second element has type `List[Either[A, A]]`, the third element has type `List[Either[Either[A, A], Either[A, A]]]`, and so on. (Exercise 3.4.2.9 shows a similar data type.) Let us call this data type `TD` ("tree descriptor"):

```
sealed trait TD[A]
final case class Last[A](a: List[A]) extends TD[A]
final case class More[A](a: List[A], tail: TD[Either[A, A]]) extends TD[A]
```

The tree descriptor for the tree `t2` shown above will then look like this:

```
val td2: TD[Int] = More(List(), More(                          // No leaves at level 0.
    List( Left(8) ), More(                                     // The leaf at level 1.
      List( Right(Right(4)) ), Last(                           // The leaf at level 2.
        List( Right(Left(Left(3))), Right(Left(Right(5))) ),   // Two leaves at level 3.
))))
```

Now the types match, while a wrong number of `Left` or `Right` wrappers will be a type error.

This representation of the tree descriptor is certainly not efficient in terms of memory consumption and processing speed. An "industry-strength" breadth-first tree traversal will use quite different data structures while implementing the same logic. For instance, one could use a bit vector $[1, 0, 1]$ instead of the wrapper `Right(Left(Right()))`, avoiding the allocation of many objects in memory. However, our present goal is not to achieve high efficiency but to obtain correct code quickly.

The next step is to write functions transforming between trees `T2[A]` and descriptors `TD[A]`:

```
def t2ToTD[A]: T2[A] => TD[A] = ???
def tdToT2[A]: TD[A] => T2[A] = ???
```

Begin implementing the first function by pattern matching:

```
def t2ToTD[A]: T2[A] => TD[A] = {
  case Leaf(a)        => Last(List(a))                       // A leaf at level 0.
```

```
3   case Branch(l, r)   => ( t2ToTD(l),  t2ToTD(r) ); ???     // How to combine the subtrees?
4 }
```

It seems reasonable that we need to apply `t2ToTD` recursively to the subtrees `l` and `r` in line 3, obtaining the tree descriptors of type `TD[A]` for those subtrees. We still have to merge these tree descriptors in the correct way. Inspired by the `listMerge` function from Section 12.2.2, we implement the descriptor-merging code by concatenating the lists separately at each depth level:

```
def tdMerge[A](l: TD[A], r: TD[A]): TD[A] = (l, r) match {
  case (Last(la), Last(lb))                 => Last(la ++ lb)
  case (Last(la), More(lb, tail))           => More(la ++ lb, tail)
  case (More(la, tail), Last(lb))           => More(la ++ lb, tail)
  case (More(la, tailA), More(lb, tailB))   => More(la ++ lb, tdMerge(tailA, tailB))
}
```

However, writing just `tdMerge(t2ToTD(l), t2ToTD(r))` would not be correct in line 3 of `t2ToTD`. We need somehow to store the information that the subtrees `l` and `r` are located one level deeper at the left and at the right respectively. To see how that information needs to be stored, let us look at the tree `t2` whose left and right subtrees are `l =` and `r =` . The descriptors of these subtrees are computed by recursive calls of `t2ToTD`, which (if implemented correctly) should yield the following:

```
t2ToTD(l) == Last(List(8))                    // One leaf at level 0.

t2ToTD(r) == More(List(), More(               // No leaves at level 0.
  List( Right(4) ), Last(                     // One leaf at level 1.
    List( Left(Left(3)), Left(Right(5)) )     // Two leaves at level 2.
)))
```

The correct merging of these tree descriptors requires wrapping all data in the subtree `l` in an additional `Left()` layer and all data in the subtree `r` in an additional `Right()` layer. Let us implement two helper functions (`addLeft` and `addRight`) for that transformation:

```
def addLeft[A]: TD[A] => TD[Either[A, A]] = {
  case Last(a)         => Last(a.map(Left(_)))
  case More(a, tail)   => More(a.map(Left(_)), addLeft[Either[A, A]](tail))
}

def addRight[A]: TD[A] => TD[Either[A, A]] = {
  case Last(a)         => Last(a.map(Right(_)))
  case More(a, tail)   => More(a.map(Right(_)), addRight[Either[A, A]](tail))
}
```

We expect that `addLeft` and `addRight` will transform the left and right subtree descriptors like this:

```
addLeft(t2ToTD(l)) == Last(List(Left(8)))

addRight(t2ToTD(r)) == More( List(), More(
  List( Right(Right(4)) ), Last(
    List( Right(Left(Left(3))), Right(Left(Right(5))) )
)))
```

It remains to mark the subtrees as being one layer deeper. We do this by inserting an empty list at the beginning of a `TD` structure. The types will then require that the subsequent lists must have one more layer of `Either` wrappers, which they will indeed have after applying `addLeft` and `addRight`. So, the correct merging of the two subtree descriptors is:

```
More(List(), tdMerge( addLeft(t2ToTD(l)), addRight(t2ToTD(r)) ))
```

The complete code of `t2ToTD` becomes:

```
def t2ToTD[A]: T2[A] => TD[A] = {
  case Leaf(a)         => Last(List(a))
  case Branch(l, r)    => More(List(), tdMerge( addLeft(t2ToTD(l)), addRight(t2ToTD(r)) ))
}
```

We can test this code on an example tree:

```
val t2 = Branch(Leaf(8), Branch(Branch(Leaf(3), Leaf(5)), Leaf(4)))

scala> t2ToTD(t2)
res0: TD[Int] = More(List(), More(List(Left(8)), More(List(Right(Right(4))),
     Last(List(Right(Left(Left(3))), Right(Left(Right(5))))))))
```

To implement the inverse function (`tdToT2`), we need to keep in mind that `t2ToTD` will only create a certain subset of possible values of type `TD[A]`. Looking at the code of `t2ToTD`, we can see, for instance, that the first list is always empty when the tree has a branch. The first list is either empty or contains only one element (the single leaf of the tree). We will never create values of type `TD[A]` where the first list has more than one element, or where the first list is nonempty and there are also nonempty subsequent lists. Such values of type `TD[A]` do not correspond to any trees of type `T2[A]`. So, the code of `tdToT2` can be a *partial* function of type `TD[A] => T2[A]` that only works with valid tree descriptors, that is, values of type `TD[A]` that were obtained by applying `t2ToTd` to some trees of type `T2[A]`. The composition `t2ToTD andThen tdToT2` should be the identity function.

Begin writing the code of `tdToT2` by pattern matching:

```
def tdToT2[A]: TD[A] => T2[A] = {
  case Last(List(a))   => Leaf(a)    // Tree has a single leaf at level 0.
  case _               => ???
}
```

Valid tree descriptors can only have a non-empty first list if the tree has a single leaf at level 0. So, we may ignore all other cases and continue with patterns that assume an empty first list:

```
1 def tdToT2[A]: TD[A] => T2[A] = {
2   case Last(List(a))        => Leaf(a)    // Tree has a single leaf at level 0.
3   case More(List(a), _)     => Leaf(a)    // Tree has a single leaf at level 0.
4   case More(List(), tail)   => ???        // Tree has two branches.
5 }
```

In line 4, we need to restore two branches of the tree from the given value `tail` of type `TD[Either[A, A]]`. How can we do that? The left branch contains all the leaf data from `tail` where the outermost wrapper is a `Left()`. We can select just that subset of leaf data and remove the outermost `Left()` wrapper by implementing the helper functions `removeLeft` and `filterLeft` as shown here:

```
def removeLeft[A]: List[Either[A, A]] => List[A] = _.collect { case Left(x) => x }

def filterLeft[A]: TD[Either[A, A]] => TD[A] = {
  case Last(la)         => Last(removeLeft(la))
  case More(la, tail)   => More(removeLeft(la), filterLeft(tail))
}
```

For example, if we apply `filterLeft` to the descriptor `t2ToTD(t2).tail` then we will get:

```
filterLeft(t2ToTD(t2).tail) == More(List(8), More(List(), Last(List())))
```

This tree descriptor corresponds to the left subtree of `t2`.

In a similar way, we implement `filterRight` that extracts the descriptor data for the right subtree:

```
def removeRight[A]: List[Either[A, A]] => List[A] = _.collect { case Right(x) => x }
def filterRight[A]: TD[Either[A, A]] => TD[A] = {
  case Last(la)         => Last(removeRight(la))
  case More(la, tail)   => More(removeRight(la), filterRight(tail))
}

scala> filterRight(t2ToTD(t2).tail)
res1: TD[Int] = More(List(), More(List(Right(4)), Last(List(Left(Left(3)), Left(Right(5))))))
```

Now we are ready to complete the implementation of `tdToT2`. We use `filterLeft` and `filterRight` to obtain the descriptors for the left and the right subtrees. Applying `tdToT2` recursively to those descriptors will restore the two subtrees:

```
def tdToT2[A]: TD[A] => T2[A] = {
  case Last(List(a))        => Leaf(a)
  case More(List(a), _)     => Leaf(a)
  case More(List(), tail)   => Branch(tdToT2(filterLeft(tail)), tdToT2(filterRight(tail)))
}

scala> tdToT2(t2ToTD(t2)) == t2
res2: Boolean = true
```

The next step is to implement `traverse` for the type constructor `TD[_]`, which makes `TD[_]` into a traversable functor. We will call its `traverse` operation "`travTD`" for clarity. Since `TD[A]` is essentially a decorated list of lists, we will need `List`'s `traverse` (Example 12.2.1.2), which we here denote by `travList`. We will also need a `traverse` function for the functor `Either[A, A]`:

```
def travEither[A, B, F[_]: Functor](f: A => F[B])(e: Either[A, A]): F[Either[B, B]] = e match {
  case Left(a)    => f(a).map(Left(_))
  case Right(a)   => f(a).map(Right(_))
}
```

Using `travList` and `travEither`, we can now implement `travTD` like this:

```
def travTD[A, B, F[_]: Applicative : Functor](f: A => F[B])(td: TD[A]): F[TD[B]] = td match {
  case Last(a)         => travList(f)(a).map(Last(_))
  case More(a, tail)   =>
    val headFB: F[List[B]] = travList(f)(a)
    val tailFB: F[TD[Either[B, B]]] = travTD { x: Either[A, A] => travEither(f)(x) }(tail)
    (headFB zip tailFB).map { case (headB, tailB) => More(headB, tailB) }
}
```

The final step is to convert `TD`'s `traverse` into `T2`'s `traverse` by using `t2ToTD` and `tdToT2`. We first compute the tree descriptor of a given tree and use `travTD` to perform a traversal. The result is a value of type `F[TD[B]]`, which we convert into a value of type `F[T2[B]]` using `tdToT2` lifted to `F`:

```
def travBF[A, B, F[_]: Applicative : Functor](f: A => F[B])(tree: T2[A]): F[T2[B]] =
  travTD(f)(t2ToTD(tree)).map(tdToT2)
```

This completes our implementation of a breadth-first traversal for binary trees. To test the code, we run the example of decorating a tree with the breadth-first traversal index:

```
def zipWithIndexBF[A](tree: T2[A]): T2[(A, Int)] = {
  val afterTraverse: St[T2[(A, Int)]] = travBF[A, (A, Int), St](computeIndex)(tree)
  afterTraverse.run(0)._1                        // Run the State monad and get the result value.
}

scala> zipWithIndexBF(t2)
res3: T2[(Int, Int)] = Branch(Leaf((8,0)), Branch(Branch(Leaf((3,2)), Leaf((5,3))), Leaf((4,1))))
```

The only difference between `zipWithIndexDF` and `zipWithIndexBF` is the choice of the `traverse` operation (`trav` or `travBF`). The "decoration logic" is described by the function `computeIndex` and does not depend on the traversal order. As a result, we gain flexibility: an arbitrary traversal order may be used with an arbitrary "decoration logic".

12.2.5 The `Traversable` typeclass. Implementing `scanLeft` via `traverse`

We define a "traversable functor" typeclass (called `Traversable`) by specifying a `traverse` operation, which is convenient to provide as an extension method:

```
trait Traversable[L[_]] {
  def trav[A, B, F[_]: Applicative: Functor](f: A => F[B])(la: L[A]): F[L[B]]
}
implicit class TraversableOps[L[_], A](la: L[A])(implicit tr: Traversable[L]) {
  def traverse[B, F[_]: Applicative : Functor](f: A => F[B]): F[L[B]] = tr.trav(f)(la)
}
```

As we have mentioned in Section 12.1.2, the `foldLeft` operation can be defined via `traverse`. The `foldLeft` operation transforms an initial sequence into a result value by updating some internal state. Another function similar to `foldLeft` is `scanLeft`; the difference is that in `scanLeft`, the result value is the *sequence* of all intermediate state values rather than just the last state value. Let us see how the functionality of `scanLeft` can be expressed using `traverse` with a `State` monad that handles the state updates. This will allow us to implement `scanLeft` automatically for every traversable functor.

Assume that `L[_]` is a traversable functor whose `trav` function is available:

```
def trav[A, B, F[_]: Applicative: Functor](f: A => F[B])(la: L[A]): F[L[B]] = ...
```

We would like to implement `scanLeft` with the following type signature:

```
def scanLeft[A, Z](la: L[A])(init: Z)(f: (A, Z) => Z): L[Z] = ???
```

The standard `scanLeft` method for lists produces a list with an extra initial element, which is always equal to the initial value `init`. This initial element is not essential and may be omitted from `scanLeft`. In fact, we *need* to omit that initial element when we generalize `scanLeft` to data types `L[A]` other than `List[A]`, since those data types will not necessarily support adding one more data item.

In order to update the state of type `Z` and also store the state's value as the result of the traversal, we will use the `State` monad and compute a value of type `State[Z, Z]`. The code of `scanLeft` calls `traverse` with a suitable function `accum` that creates the required value of type `State[Z, Z]`:

```
case class State[Z, A](run: Z => (A, Z)) // Assume Applicative and Functor instances for State[Z, *].
def accum[A, Z](a: A, f: (A, Z) => Z): State[Z, Z] = State { z =>
  val newZ = f(a, z)  // Update the internal state using ‘f‘.
  (newZ, newZ)        // Store the internal state, and also return it as a result value.
}
implicit class TraversableScanOps[L[_] : Traversable, A](la: L[A]) {
  def scanLeft[Z](init: Z)(f: (A, Z) => Z): L[Z] = la.traverse(a => accum(a, f)).run(init)._1
}
```

In this way, `scanLeft` is made available for all traversable functors.

To test this code, we implement `zipWithIndex` via depth-first traverse for a tree of type `T2`:

```
def zipWithIndexDFS[A]: T2[A] => T2[(A, Int)] =
  _.scanLeft[(A, Int)]((null.asInstanceOf[A], -1)) { case (a, (_, i)) => (a, i + 1) }
val t2 = Branch(Leaf(8), Branch(Branch(Leaf(3), Leaf(5)), Leaf(4)))

scala> zipWithIndexDFS(t2)
res0: T2[(Int, Int)] = Branch(Leaf((8,0)), Branch(Branch(Leaf((3,1)), Leaf((5,2))), Leaf((4,3))))
```

12.2.6 Tasks that cannot be performed via `traverse`

The `traverse` function is powerful since it can use an arbitrary applicative functor `F[_]`. However, some computations are still not expressible via `traverse` because they require information that `traverse` cannot have. We will now look at two examples of this.

The first example is the depth labeling of a tree: each leaf gets a value equal to its depth. For instance, the tree

becomes

after depth labeling. This cannot be implemented via `traverse` because it cannot detect nodes that have the same depth in the tree. To see this in more detail, recall that the code of `traverse` always collects and merges all the effects of a given functor `F`. If the effect of `F` describes incrementing a counter (as in our examples involving the `State` monad), the code of `traverse` will increment that counter at *each* leaf. We may traverse the tree so that leaves at the same depth are traversed next to each other (as in the breadth-first traversal). But the code of `traverse` cannot skip the incrementing when a leaf is at the same depth: `traverse` does not receive any information about the position of values in the tree. So, we cannot label certain nodes with the same depth value but other nodes with a different depth value.

Depth labeling can be implemented as a special operation such as `zipWithDepth` for the tree type `T2`:

```
def zipWithDepth[A](initial: Int = 0): T2[A] => T2[(A, Int)] = {
  case Leaf(a)        => Leaf((a, initial))
  case Branch(l, r)   => Branch(zipWithDepth(initial + 1)(l), zipWithDepth(initial + 1)(r))
}
val t2: T2[Int] = Branch(Leaf(8), Branch(Branch(Leaf(3), Leaf(5)), Leaf(4)))

scala> zipWithDepth()(t2)
res0: T2[(Int, Int)] = Branch(Leaf((8,1)), Branch(Branch(Leaf((3,3)), Leaf((5,3))), Leaf((4,2))))
```

More generally, traversals cannot perform computations that depend on the *position* of data in the tree (e.g., whether the data is in a left or in a right branch and at what depth). An example of such a computation is "pretty-printing" that converts trees into a text form. In the LaTeX format used to typeset this book, the tree

is represented by the text string `"\Tree[ 8 [ [ 3 5 ] 4 ] ]"`. We may write the following code for converting trees to this format:

```
def printLaTeX[A](t: T2[A])(toString: A => String): String = {
  def printLaTeXSubtree: T2[A] => String = {
    case Leaf(a)        => toString(a)
    case Branch(l, r)   => "[ " + printLaTeXSubtree(l) + " " + printLaTeXSubtree(r) + " ]"
  }
  "\\Tree" + printLaTeXSubtree(t)
}

scala> printLaTeX(t2)(_.toString)
res1: String = \Tree[ 8 [ [ 3 5 ] 4 ] ]
```

12.2.7 Recursion schemes. I. Folding operations

The previous section showed two examples of folding and traversing operations that cannot be expressed through the standard `foldMap` or `traverse` methods and are instead implemented via custom recursive code. If we needed to implement the same operations for trees of different shapes or for other recursive data types, we would need to write new custom code for each new data type. That code will contain a certain common pattern: it will use recursive calls at the points where the recursive data type refers to itself. It turns out that we can separate this common pattern from the custom code, reducing the effort required for implementing the folding and traversing operations for different recursive data types.

To begin, write the recursive definition of the tree `T2` as:

$$\text{T2}^A \stackrel{\text{def}}{=} A + \text{T2}^A \times \text{T2}^A \quad .$$

This type refers recursively to itself in two places. To express that, define a bifunctor $S^{A,R}$:

$$S^{A,R} \stackrel{\text{def}}{=} A + R \times R \quad .$$

We can now rewrite the definition of `T2` as a recursive type equation: $\text{T2}^A \stackrel{\text{def}}{=} S^{A,\text{T2}^A}$. The corresponding Scala code is:

```
type S[A, R] = Either[A, (R, R)]
final case class T2[A](run: S[A, T2[A]])
```

The bifunctor $S^{\bullet,\bullet}$ is called the **recursion scheme** of the type `T2`. The recursion scheme describes the places where the recursive type refers to itself in its definition. All the recursive uses correspond to occurrences of the type parameter R in $S^{A,R}$.

The folding operation `foldMap` takes a function f of type $A \to Z$ as a parameter:

$$\text{foldMap}_L(f^{:A\to Z}) : L^A \to Z \quad .$$

The function f will be applied to all values of type A stored inside L^A. As we have seen in the previous section, the function $f^{:A\to Z}$ cannot receive any information about the location of values of type A inside L^A. In order to access that information and make the folding operation "location-aware", we need to change the type signature of f. To figure out the new type signature, let us look at the code of `printLaTeXSubtree` (short notation "`pls`") from the previous section:

$$\text{pls} \stackrel{\text{def}}{=} \left|\begin{array}{c|c} & \text{String} \\ \hline A & \text{toString} \\ \text{T2}^A \times \text{T2}^A & (\overline{\text{pls}} \boxtimes \overline{\text{pls}}) \fatsemi (l \times r \to \text{"["} + l + \text{" "} + r + \text{"]"}) \end{array}\right| \quad .$$

We can rewrite this code using the bifunctor $S^{A,R} \stackrel{\text{def}}{=} A + R \times R$. In this example, the result type Z is `String`. By calling `printLaTeXSubtree` recursively on the left and the right subtrees, we obtain two values of type Z. We can separate the recursive calls to `printLaTeXSubtree` from the custom string processing and express `printLaTeXSubtree` as the following function composition:

$$\text{pls} \stackrel{\text{def}}{=} \left|\begin{array}{c|cc} & A & Z \times Z \\ \hline A & \text{id} & \mathbb{0} \\ \text{T2}^A \times \text{T2}^A & \mathbb{0} & \overline{\text{pls}} \boxtimes \overline{\text{pls}} \end{array}\right| \fatsemi \left|\begin{array}{c|c} & Z \\ \hline A & \text{toString} \\ Z \times Z & l \times r \to \text{"["} + l + \text{" "} + r + \text{"]"} \end{array}\right| \quad .$$

The intermediate result is a data structure of type $A + Z \times Z$, to which we need to apply some string manipulations. We note that the type $A + Z \times Z$ is the same as $S^{A,Z}$, while the first matrix in the code above is the lifting $\overline{\text{pls}}^{\uparrow S^{A,\bullet}}$ with respect to the type parameter R of $S^{A,R}$. So, we may express the custom string manipulations specific to `printLaTeXSubtree` via a function `toLaTeX` of type $S^{A,Z} \to Z$:

$$\text{toLaTeX} : S^{A,Z} \to Z \quad , \qquad \text{toLaTeX} \stackrel{\text{def}}{=} \left|\begin{array}{c|c} & Z \\ \hline A & \text{toString} \\ Z \times Z & l \times r \to \text{"["} + l + \text{" "} + r + \text{"]"} \end{array}\right| \quad .$$

Using this function, the implementation of `printLaTeXSubtree` can be rewritten as:

$$\text{pls} \stackrel{\text{def}}{=} \overline{\text{pls}}^{\uparrow S^{A,\bullet}} \fatsemi \text{toLaTeX} \quad .$$

All custom "location-aware" logic is now encapsulated in the (non-recursive!) function `toLaTeX`. So, we can now generalize the calculation by defining a new `fold` function (denoted by fold_S):

$$\text{fold}_S : (S^{A,Z} \to Z) \to L^A \to Z \quad , \qquad \text{fold}_S(f) \stackrel{\text{def}}{=} \overline{\text{fold}_S(f)}^{\uparrow S^{A,\bullet}} \fatsemi f \quad . \tag{12.1}$$

We then obtain $\text{pls} = \text{fold}_S(\text{toLaTeX})$. The corresponding Scala code is:

```
def fmapR[A, R, T](f: R => T): S[A, R] => S[A, T] = _.map { case (r1, r2) => (f(r1), f(r2)) }

def foldS[A, Z](f: S[A, Z] => Z)(tree: T2[A]): Z = f(fmapR(foldS(f))(tree.run))

def toLaTeX[A]: S[A, String] => String = {
  case Left(a)          => a.toString
  case Right((l, r))    => s"[ $l $r ]"
}
def printLaTeX[A](tree: T2[A]): String = "\\Tree" + foldS[A, String](toLaTeX)(tree)

val t2: T2[Int] = T2(Right((T2(Left(8)), T2(Right((T2(Right((T2(Left(3)), T2(Left(5))))),
    T2(Left(4))))))))

scala> printLaTeX(t2)
res0: String = \Tree[ [ 8 [ 3 5 ] ] 4 ]
```

It is important that the function fold_S is parametric in the recursion scheme $S^{\bullet,\bullet}$ and the result type Z (which is not required to be a monoid). Different recursion schemes $S^{\bullet,\bullet}$ may be used to define lists, trees, and other recursive data types. The same code of fold_S will work for all those data types, as long as we have the recursion scheme $S^{\bullet,\bullet}$ and the corresponding lifting function (`fmapR` in the Scala code shown above, or ${}^{\uparrow S^{A,\bullet}}$ in the code notation).

To illustrate the general applicability of fold_S to different data types, let us implement a `printLaTeX` function for ordinary lists, for non-empty lists, and for rose trees (Section 3.3.4).

Scala's standard `List` type and the corresponding recursion scheme $S^{A,R}$ may be defined by:

$$\text{List}^A \stackrel{\text{def}}{=} \mathbb{1} + A \times \text{List}^A \quad , \qquad S^{A,R} \stackrel{\text{def}}{=} \mathbb{1} + A \times R \quad , \qquad \text{List}^A \stackrel{\text{def}}{=} S^{A,\text{List}^A} \quad .$$

A non-empty list type (`NEL`) and the corresponding recursion scheme $S^{A,R}$ may be defined by:

$$\text{NEL}^A \stackrel{\text{def}}{=} A + A \times \text{NEL}^A \quad , \qquad S^{A,R} \stackrel{\text{def}}{=} A + A \times R \quad , \qquad \text{NEL}^A \stackrel{\text{def}}{=} S^{A,\text{NEL}^A} \quad .$$

A rose tree (`TreeN`) and the corresponding recursion scheme $S^{A,R}$ may be defined by:

$$\text{TreeN}^A \stackrel{\text{def}}{=} A + \text{NEL}^{\text{TreeN}^A} \quad , \qquad S^{A,R} \stackrel{\text{def}}{=} A + \text{NEL}^R \quad , \qquad \text{TreeN}^A \stackrel{\text{def}}{=} S^{A,\text{TreeN}^A} \quad .$$

For rose trees, the recursion scheme $S^{A,R}$ is itself a recursively defined type because it uses the non-empty list (NEL). This is not a problem: $S^{A,R}$ is still polynomial, which guarantees that any value of type $S^{A,R}$ contains a finite number of elements of types A and R. So, any value of type `TreeN[A]` will contain a finite number of values of type A, assuring that the folding operation will terminate.

To avoid repetitive code, let us define all three data types (`List`, `NEL`, and `TreeN`) at once through a universal recursive class `Fix` that takes the recursion scheme $S^{\bullet,\bullet}$ as a type parameter:

```
type S1[A, R] = Option[(A, R)]              // For List.
type S2[A, R] = Either[A, (A, R)]           // For NEL.
type S3[A, R] = Either[A, NEL[R]]           // For TreeN.
final case class Fix[S[_, _], A](unfix: S[A, Fix[S, A]])
```

Using the class `Fix`, we may write the equivalent definitions `List[A]` ≅ `Fix[S1, A]`, `NEL[A]` ≅ `Fix[S2, A]`, and `TreeN[A]` ≅ `Fix[S3, A]`. Some example values of these types are:

```
val x1 = Fix[S1, Int](Some((1, Fix[S1, Int](Some((2, Fix[S1, Int](Some((3, Fix[S1,
     Int](None))))))))))  // Equivalent to List(1, 2, 3) .
val x2 = Fix[S2, Int](Right((1, Fix[S2, Int](Right((2, Fix[S2, Int](Left(3))))))))  // NEL(1, 2, 3) .
val x3 = Fix[S3, Int](Right(NEL(Fix[S3, Int](Right(NEL(Fix[S3, Int](Left(10)), Fix[S3,
     Int](Right(NEL(Fix[S3, Int](Left(20)), Fix[S3, Int](Left(30))))))))), Fix[S3, Int](Left(40)))))
```

In practice, defining data structures via `Fix` is both inconvenient and inefficient. We show these definitions only to clarify how one can generalize "location-aware" folding operations to arbitrary recursive data types. (For simplicity, the definition of `S3` uses `NEL` rather than `Fix[S2, A]`.)

To make the code of fold_S general, we add a functor typeclass constraint to $S^{A,\bullet}$:

```
def fold[A, Z, S[_,_]](f: S[A, Z] => Z)(t: Fix[S, A])(implicit fs: Functor[S[A, *]]): Z =
  f(t.unfix.map(fold(f)))
```

Below we will assume that the appropriate functor instances are defined for `S1`, `S2`, and `S3`.

It remains to implement functions with type signatures `S[A, String] => String`:

```
def toLaTeX1[A]: S1[A, String] => String = {
  case None                  => "Nil"
  case Some((head, tail))    => head.toString + ", " + tail
}
def toLaTeX2[A]: S2[A, String] => String = {
  case Left(a)                => a.toString
  case Right((head, tail))    => head.toString + ", " + tail
}
def toLaTeX3[A]: S3[A, String] => String = {
  case Left(a)       => a.toString
```

```
  case Right(nel)   => "[ " + nel.mkString(" ") + " ]" // Assume mkString() is defined for NEL.
}
```

We can now use fold_S to convert some data structures to a LaTeX form:

```
def listToLaTeX[A](t: Fix[S1, A]): String = "[ " + fold[A, String, S1](toLaTeX1)(t) + " ]"
def nelToLaTeX[A](t: Fix[S2, A]): String = "[ " + fold[A, String, S2](toLaTeX2)(t) + " ]"
def treeNToLaTeX[A](t: Fix[S3, A]): String = "\\Tree" + fold[A, String, S3](toLaTeX3)(t)

scala> listToLaTeX(x1)
res1: String = [ 1, 2, 3, Nil ]

scala> nelToLaTeX(x2)
res2: String = [ 1, 2, 3 ]

scala> treeNToLaTeX(x3)
res3: String = \Tree[ [ 10 [ 20 30 ] ] 40 ]
```

The definition of fold_S calls itself recursively under the lifting $(\overline{\text{fold}_S(f)}^{\uparrow S^{A,\bullet}})$. Is it guaranteed that this recursion terminates? The only way it can terminate is when the lifting $f^{\uparrow S^{A,\bullet}}$ does not *always* call the function f. This will happen if $S^{A,R}$ is a disjunctive type with some parts that do not depend on R. This is indeed the case for lists ($S^{A,R} \stackrel{\text{def}}{=} \mathbb{1} + A \times R$ or $S^{A,R} \stackrel{\text{def}}{=} A + A \times R$) and trees ($S^{A,R} \stackrel{\text{def}}{=} A + R \times R$). Applying $f^{\uparrow S^{A,\bullet}}$ to values of R-independent types will be an identity function; it will not actually call f. This will be the base case of the recursion.

It remains to assure that the recursion reaches the base case with every value of type L^A. That is, no values of type L^A should cause an infinite loop in fold_S. A simple example where fold_S enters an infinite loop is the recursion scheme $S^{A,R} \stackrel{\text{def}}{=} A + (\mathbb{1} \to R)$. Note that $S^{A,R}$ is non-polynomial due to the function type $\mathbb{1} \to R$, which delays the evaluation of a value of type R. This allows us to implement a well-defined, finite value `x: L[A]` which refers to itself under the delayed evaluation:

```
type S[A, R] = Either[A, Unit => R]
final case class Looping[A](run: S[A, Looping[A]])
lazy val x: Looping[Int] = Looping(Right(_ => x))

scala> x.run.right.get(())                        // No stack overflows.
val res0: Looping[Int] = Looping(Right($Lambda$1123/1058984040@753aca85))

scala> x.run.right.get(()).run.right.get(())     // This is again the same value:
val res1: Looping[Int] = Looping(Right($Lambda$1123/1058984040@753aca85))
```

Trying to compute `fold(f)(x)` with any `f` will result in an infinite loop.

It seems that we need to restrict recursion schemes $S^{A,R}$ to *polynomial* bifunctors. Such $S^{A,R}$ will define recursive polynomial functors L^A that support no delayed evaluation of stored values of type A. So, any value `x` of type L^A will have to contain a finite number of values of type A, and `fold(f)(x)` is guaranteed to terminate for any terminating function $f : S^{A,Z} \to Z$.

Rather than working with the general fold_S function and redefine all recursive types via `Fix`, it is more convenient to implement and use specialized versions of fold_S for already defined recursive types. The general implementation of fold_S in Eq. (12.1) can be translated mechanically (e.g., using macros or code generators) into code specialized for a given data type and recursion scheme.

For instance, while the type `TreeN[A]` is equivalent to `Fix[S3[A, *]]` shown above, it is easier to work with `TreeN[A]`. The specialized version of fold_S for `TreeN[A]` has the type signature:

```
def foldTreeN[A, Z](f: S3[A, Z] => Z): TreeN[A] => Z = ???
```

The general definition of fold_S in Eq. (12.1) shows us how to write the code of `foldTreeN`:

```
def foldTreeN[A, Z](f: Either[A, NEL[Z]] => Z): TreeN[A] => Z = {
  case Leaf(a)      => f(Left(a))
  case Branch(ts)   => f(Right(ts.map(foldTreeN(f))))
}
```

Then we can implement `printLaTeX` for `TreeN[A]` like this:

```
def printLaTeX[A](tree: TreeN[A]): String = "\\Tree" + foldTreeN[A, String](toLaTeX3)(tree)
```

Another simple example of an aggregation operation that cannot be expressed as a traversal is the task of determining the maximum branching number of a given rose tree. The function `foldTreeN` now allows us to implement that computation:

```
def maxBranching[A]: TreeN[A] => Int = foldTreeN[A, Int] {
  case Left(_)      => 0
  case Right(nel)   => math.max(nel.max, nel.length)    // NEL must have ‘max‘ and ‘length‘ methods.
}

scala> maxBranching(x3)
res2: Int = 2
```

12.2.8 Recursion schemes. II. Unfolding operations

A folding operation converts a collection to a single value. The opposite operation is "unfolding": converting a single value into a collection. By reversing the direction of certain function arrows in the type signature of fold_S, we can define a general "unfolding" method that uses an arbitrary recursion scheme $S^{A,R}$ and an arbitrary function of type $Z \to S^{A,Z}$:

$$\text{unfold}_S : (Z \to S^{A,Z}) \to Z \to L^A \quad , \quad \text{unfold}_S(f) \stackrel{\text{def}}{=} f \,\fatsemi\, \overline{\text{unfold}_S(f)}^{\uparrow S^{A,\bullet}} \quad . \tag{12.2}$$

Section 2.3 showed an unfolding operation for sequences: starting from an initial value, a function is applied repeatedly to compute further elements of the sequence. The `unfold` operation generalizes that computation to an arbitrary recursive type L^A whose recursion scheme $S^{A,R}$ is given.

To get more intuition, we look at some examples using `unfold` with lists and binary trees.

Example 12.2.8.1 Use `unfold` to create a `List` of consecutive powers of 2 up to a given value n:

```
type S[A, R] = ???
type Z = ???

def unfoldList[A, Z](f: Z => S[A, Z])(init: Z): List[A] = ???

def powersOf2UpTo(n: Long): List[Long] = unfoldList(???)(???)

scala> powersOf2UpTo(1000)
res0: List[Long] = List(1, 2, 4, 8, 16, 32, 64, 128, 256, 512)
```

Solution The recursion scheme for `List[A]` is $S^{A,R} \stackrel{\text{def}}{=} \mathbb{1} + A \times R$. Let us specialize the code of `unfold` from Eq. (12.2) to the type `List[A]`:

```
type S[A, R] = Option[(A, R)]
def unfoldList[A, Z](f: Z => S[A, Z])(init: Z): List[A] = f(init) match {
  case None          => Nil
  case Some((a, z))  => a :: unfoldList(f)(z)
}
```

We now need to determine a suitable type Z and a suitable function $f : Z \to \mathbb{1} + A \times Z$ so that the unfolding would produce the required sequence. If we are in the middle of unfolding, we need to produce the remaining portion of the list, say, $[128, 256, 512]$, given only a current value of type Z. We can do that if the current value of type Z is 128 (the smallest remaining power of 2). So, let us choose $Z \stackrel{\text{def}}{=}$ `Long` to represent the smallest remaining power of 2. The type A will be also `Long`.

How can we implement the function f? It should return `None` (denoted by $1 + \mathbb{0}$) when the list is finished. When we are in the middle of generating the list, the call $f(z)$ should return $\mathbb{0} + a \times z'$ with some values a and z'. The value a must be the new element of the list. The value z' will be passed to the next call of f. Since the next element must be twice the previous one, we must have $a = z$ and $z' = 2 * z$. So, the code of f is:

```
def f(n: Long): Long => Option[(Long, Long)] = { z => if (z >= n) None else Some((z, z * 2)) }
```

Note that the code of f is *not* recursive, and the value n is captured inside the nameless function returned by $f(n)$. We can now complete the solution:

```
def powersOf2UpTo(n: Long): List[Long] = unfoldList(f(n))(1)
```

Example 12.2.8.2 Implement `unfoldT2` for the data type `T2[A]` from Section 12.2.7. Use it to create "full" binary trees of given depth, e.g.,

(depth 2) and

(depth 3).

Solution We adapt the general code in Eq. (12.2) to obtain the code of `unfoldT2`:

```
type S[A, R] = Either[A, (R, R)]
def unfoldT2[A, Z](f: Z => S[A, Z])(init: Z): T2[A] = f(init) match {
  case Left(a)            => Leaf(a)
  case Right((z1, z2))    => Branch(unfoldT2(f)(z1), unfoldT2(f)(z2))
}
```

We plan to implement the function `fullBinaryTree` like this:

```
def fullBinaryTree(n: Int): T2[Int] = {
  type Z = ???
  val init: Z = ???
  val f: Z => Either[Int, (Z, Z)] = ???
  unfoldT2[Int, Z](f)(init)
}
```

The next step is to choose a suitable type `Z` such that the unfolding procedure can generate trees of the required shape. To figure out what `Z` must be, we need to consider an intermediate step that generates a subtree at some point in the middle of "unfolding". For the tree of depth 3 as shown above, an example of a subtree in the middle is

. This subtree must be computed as `unfoldT2(f)(z)` with some `z` of type `Z`. The information needed for this computation is the initial value (4) and the total number (2) of the subtree's leaves. So, the type `Z` needs to contain two integers: the value of the first leaf and the size of the remaining subtree (which will always be a power of 2).

```
type Z = (Int, Int)        // (k, m) where k is the first leaf's value and m is the subtree size.
val init: Z = (0, 1 << n)  // The size of the entire tree is 2 to the power n.
```

The initial value for the entire tree shown above will be `(0, 8)`.

Next, we figure out the function `f`. When `f` is applied to a value `(k, m)` of type `Z`, it should generate the corresponding subtree. If the subtree size `m` is 1, the return value is `Left(k)`. Otherwise, the return value should give two new values of type `Z` corresponding to the two subtrees one level deeper. Those two subtrees have sizes `m / 2` and initial values `k` and `m / 2 + k`. So, the code of `f` is:

```
val f: Z => Either[Int, (Z, Z)] = {
  case (k, m) if m == 1   => Left(k)
  case (k, m)             => Right(((k, m / 2), (m / 2 + k, m / 2)))
}
```

This completes the implementation of `fullBinaryTree`. To test the resulting code, compute a full tree of depth 2:


```
scala> fullBinaryTree(2)
res0: T2[Int] = Branch(Branch(Leaf(0), Leaf(1)), Branch(Leaf(2), Leaf(3)))
```

Example 12.2.8.3 Use `unfoldT2` from Example 12.2.8.2 to implement the function `evenOdd(n)` that generates binary trees of type `T2[Int]` where the leaves have descending numbers from `n` to `0`, but all odd numbers are on the left and all even numbers on the right. For example, `evenOdd(3)` should generate the tree

, while `evenOdd(4)` should give the tree

.

Solution We plan to implement `evenOdd` like this:

```
def evenOdd(n: Int): T2[Int] = {
  type Z = ???
  val init: Z = ???
  val f: Z => Either[Int, (Z, Z)] = ???
  unfoldT2[Int, Z](f)(init)
}
```

The examples with `evenOdd(3)` and `evenOdd(4)` suggest that `evenOdd(n)` is a tree containing a leaf with value `n` and a subtree `evenOdd(n - 1)`. Can we use the integer `n` as the initial value for unfolding?

To figure this out, consider an intermediate stage of the unfolding process where `unfoldT2` will apply the function `f` to some value `z` of type `Z`. If `f(z) == Left(k)`, we will get a `Leaf` with value `k`. The other possibility is `f(z) == Right((z1, z2))`, where `z1` and `z2` are some values of type `Z`. This will generate a `Branch` with two subtrees. The function `unfoldT2` will then apply `f` to `z1` and `z2` in order to create the left and the right subtrees. The difference between those subtrees must come entirely from the difference between the values `z1` and `z2`.

In our case, we need to make a `Leaf` either at the left or at the right depending on whether the initial leaf value is odd or even. The function `f` must know whether `f(z)` should return a subtree or a `Leaf`. This information can only come from the value `z`. So, the type `Z` must contain a `Boolean` flag saying whether we need a `Leaf` at the current place. For clarity, let us define the type `Z` as a case class:

```
final case class Z(startAt: Int, makeLeaf: Boolean)
val init = Z(startAt = n, makeLeaf = false)
```

When `makeLeaf` is `true`, we must create a `Leaf`, so `f` should return a `Left()`. Otherwise `f` should return a `Right((z1, z2))`, where `z1` and `z2` should set `makeLeaf` depending on `startAt` being odd or even. Looking at the examples, we find that `f` must also return a `Left()` when `startAt == 0`. So, the code of `f` is:

```
val f: Z => Either[Int, (Z, Z)] = {
  case Z(n, false) if n > 0 && n % 2 == 0  => Right((Z(n - 1, false), Z(n, true)))
  case Z(n, false) if n > 0 && n % 2 == 1  => Right((Z(n, true), Z(n - 1, false)))
  case Z(n, _)                             => Left(n) // Make a leaf when n == 0 or makeLeaf == true.
}
```

The implementation of `evenOdd` is now complete. To test:

```
scala> evenOdd(3)
res0: T2[Int] = Branch(Leaf(3), Branch(Branch(Leaf(1), Leaf(0)), Leaf(2)))
```

□

The kind of reasoning shown in Examples 12.2.8.1–12.2.8.3 is called **co-induction**. It is related to mathematical induction but is significantly different from the reasoning required to write the code for a folding operation (which is directly modeled on induction). In co-induction, the base cases are not at the beginning of the computation but "in the future". Note that `unfold(f)(z)` will call itself whenever the value $f(z)$ of type $S^{A,Z}$ contains additional values of type Z. The programmer must carefully choose a suitable type Z and a suitable function f such that `unfold(f)(z)` stops the recursion at the required places. For instance, if $S^{A,Z} \stackrel{\text{def}}{=} A + Z \times Z$, the function f must sometimes return a value of type $A + \mathbb{0}$ to stop the unfolding.

Is the recursion guaranteed to stop while evaluating `unfold`? The type signature $f : Z \to S^{A,Z}$ itself does not guarantee that $f(z)$ returns values that will stop the unfolding at the right places. If $S^{A,Z} \stackrel{\text{def}}{=} A + Z \times Z$ and $f(z)$ always returns values of type $\mathbb{0} + Z \times Z$ (for example, $f(z) \stackrel{\text{def}}{=} \mathbb{0} + z \times z$), the unfolding operation `unfold(f)` will enter an infinite loop trying to constructing a tree of infinite size. This will, of course, fail since data structures in a computer cannot have infinite size.

Unfolding will always terminate if we use a data type that *delays* the evaluation of its recursively defined parts. Those parts will be computed only on demand. To obtain further data, the code needs to call certain functions. Data types of this kind are sometimes called "infinite", which is misleading since no infinite amount of data is involved. A function may be called many times and produce any number of result values, but it does not mean that a function stores an infinite amount of data.

As an example, consider the recursion scheme $S^{A,R} \stackrel{\text{def}}{=} A + (\mathbb{1} \to R \times R)$. The corresponding data type $L^A \stackrel{\text{def}}{=} S^{A,L^A}$ is a binary tree whose branches are evaluated on demand (but leaves are evaluated eagerly). This data structure supports unfolding with *any* function $f : Z \to S^{A,Z}$ because the recursive evaluation of f at the branches is always delayed. For instance, `unfold` can generate a value of type L^A whose tree structure has the form with unbounded size:

```
type S[A, R] = Either[A, Unit => (R, R)]

sealed trait UT[A]      // A tree with on-call evaluation of branches and eager evaluation of leaves.
case class ULeaf[A](a: A)                          extends UT[A]
case class UBranch[A](run: Unit => (UT[A], UT[A])) extends UT[A] // Call run(()) to get the branches.

def unfoldUT[A, Z](f: Z => S[A, Z])(init: Z): UT[A] = f(init) match {
  case Left(a)       => ULeaf(a)
  case Right(func)   => UBranch { _ =>     // It is important to delay the evaluation of func(()).
    val (z1, z2) = func(())                // Force the evaluation of branches at this level.
    (unfoldUT(f)(z1), unfoldUT(f)(z2))     // ‘unfold‘ will delay the evaluation of further branches.
  }
}

val tree1toInf = unfoldUT[Int, (Int, Boolean)] { case (z, makeLeaf) =>
  if (makeLeaf) Left(z) else Right(_ => ((z + 1, true), (z + 1, false)))
}((0, false))
```

The value `tree1toInf` is finite but can compute on demand a potentially unbounded number of leaves. To visualize `tree1toInf`, we write a function that converts `UT[A]` to `T2[A]` by stopping at a given maximum depth. The unevaluated parts of the tree will be marked with a value called `default`:

```
def toT2[A](maxDepth: Int, default: A): UT[A] => T2[A] = {
  case ULeaf(a)        => Leaf(a)e
  case UBranch(func)   => if (maxDepth == 0) Leaf(default) else {
    val (z1, z2) = func(())
    Branch(toT2(maxDepth - 1, default)(z1), toT2(maxDepth - 1, default)(z2))
  }
}
```

To test this code, let us truncate the infinite structure `tree1toInf` at depth 4. The result is a finite tree of type `T2[Int]`, where the unevaluated part of the tree is shown as "`-1`":

```
scala> toT2(maxDepth = 3, default = -1)(tree1toInf)
res0: T2[Int] = Branch(Leaf(1), Branch(Leaf(2), Branch(Leaf(3),
    Branch(Leaf(4), Leaf(-1)))))
```

12.2.9 Recursion schemes. III. Traversing operations

Folding with a recursion scheme (fold_S) allows us to implement operations such as `printLaTeXSubtree` (Section 12.2.7) that cannot be expressed via ordinary `fold` or `traverse` functions. Another operation not expressible via `traverse` is `zipWithDepth`, which we implemented in Section 12.2.6 through custom code. We will now implement `zipWithDepth` via a more general traversal operation (trav_S) parameterized by an arbitrary recursion scheme $S^{\bullet,\bullet}$ and an arbitrary functor $F^\bullet$ (not necessarily applicative).

To figure out the type signature of trav_S, consider the relationship between `foldMap`, fold_S, and the ordinary `traverse` (denoted trav_L):

$$\text{foldMap}_L(f^{:A\to Z}) : L^A \to Z \quad , \qquad \text{trav}_L(f^{:A\to F^B}) : L^A \to F^{L^B} \quad ,$$
$$\text{fold}_S(f^{:S^{A,Z}\to Z}) : L^A \to Z \quad , \qquad \text{trav}_S(f^{:???}) : L^A \to F^{L^B} \quad .$$

We recover `foldMap` from `traverse` by setting the applicative functor F as $F^B \stackrel{\text{def}}{=} Z$. So, we expect to obtain the type signature of trav_S from the type signature of fold_S if we replace Z by F^{L^B}. The first

argument f of trav_S will then have the type $S^{A,F^{L^B}} \to F^{L^B}$:

$$\text{trav}_S : (S^{A,F^{L^B}} \to F^{L^B}) \to L^A \to F^{L^B} \quad .$$

To implement this method, begin with the type equivalence $L^A \cong S^{A,L^A}$. We can apply $\text{trav}_S(f)$ recursively to the values of type L^A stored inside S^{A,L^A} and obtain a value of type $S^{A,F^{L^B}}$:

$$\left(s^{:S^{A,L^A}} \triangleright \overline{\text{trav}_S(f)}^{\uparrow S^{A,\bullet}}\right) : S^{A,F^{L^B}} \quad .$$

It remains to apply f to the last obtained value. This completes the code of trav_S:

$$\text{trav}_S\left(f^{:S^{A,F^{L^B}} \to F^{L^B}}\right) \stackrel{\text{def}}{=} \overline{\text{trav}_S(f)}^{\uparrow S^{A,\bullet}} \fatsemi f \quad .$$

The method trav_S works in the same way for all recursion schemes S and for all type constructors F. This makes trav_S powerful but hard to use because we need to work with data structures defined via the `Fix` type constructor. It is easier to use a specialized version of trav_S for the data structure at hand, similarly to what we did in the previous sections for folding and unfolding.

To illustrate this, let us implement `zipWithDepth` via a traversal with a binary tree recursion scheme. We will use the `State` monad as the functor F:

```
final case class St[A](run: Int => (A, Int))  { // A State monad with internal state of type Int.
  import io.chymyst.ch.implement                // Implement the monad methods automatically.
  def flatMap[B](f: A => St[B]): St[B] = implement
  def map[B](f: A => B): St[B] = implement
}
def incrementAndGet: St[Int] = St(s => (s + 1, s + 1))  // Increment the current state value.
def get: St[Int] = St(s => (s, s))              // Fetch the current state value.
def set(s: Int): St[Unit] = St(_ => ((), s))    // Set the state, ignore previous state value.
```

Next, define the recursion scheme $S^{A,R}$ and a specialized version of trav_S for trees of type `T2[A]`:

```
type S[A, R] = Either[A, (R, R)]          // Recursion scheme for T2[A].
def travT2[A, B, F[_]](f: S[A, F[T2[B]]] => F[T2[B]]): T2[A] => F[T2[B]] = {
  case Leaf(a)        => f(Left(a))
  case Branch(l, r)   => f(Right((travT2(f)(l), travT2(f)(r))))
}
```

It remains to apply `travT2` to a suitable function `f`. The value of the internal state will represent the current depth of the tree element. We need to increment the depth whenever we find a branch and then traverse the two subtrees starting from the same depth value. The code is:

```
def zipWithDepth[A](tree: T2[A]): T2[(A, Int)] = travS[A, (A, Int), St] {
  case Left(a) => for { s <- get } yield Leaf((a, s))   // Put the current depth into the Leaf value.
  case Right((l, r))   => for {
    s <- incrementAndGet   // Read the current depth after incrementing it.
    x <- l                 // Traverse the left branch starting from depth ‘s‘.
    _ <- set(s)            // Set the same initial depth ‘s‘ for traversing the right branch.
    y <- r                 // Traverse the right branch.
  } yield Branch(x, y)
}(tree).run(0)._1
```

To test the code, apply `zipWithDepth` to a sample tree `t2` used earlier in Sections 12.2.4 and 12.2.6:

```
scala> zipWithDepth(t2)
res0: T2[(Int, Int)] = Branch(Leaf((8, 1)), Branch(Branch(Leaf((3,
     3)), Leaf((5, 3))), Leaf((4, 2))))
```

Here we need to use the `State` monad's method `set(s)` with a value `s` obtained from a previous monadic computation. So, the code involves $\text{trav}_S(f)$ with a function $f : S^{A,F^{L^B}} \to F^{L^B}$ whose F-effect is *not* equivalent to an applicative functor's effect (which cannot depend on a previously computed value). We can see that the recursion scheme-based traversal (trav_S) is more powerful than the plain traversal, $\text{trav}_L(f^{:A \to F^B})$, that may only use F's applicative functor methods.

12.2.10 Exercises

Exercise 12.2.10.1 Implement `foldMap` and `traverse` for the following type constructors:

(a) $F^A \stackrel{\text{def}}{=} \text{Int} + A + A \times A$.

(b) $F^A \stackrel{\text{def}}{=} (\text{String} + A) \times (\mathbb{1} + A \times A)$.

(c) $F^A \stackrel{\text{def}}{=} \mathbb{1} + A + A \times A \times F^A$ (a recursive definition).

(d) $F^A \stackrel{\text{def}}{=} (\mathbb{1} + F^A) \times (\mathbb{1} + A \times F^A)$ (a recursive definition).

Exercise 12.2.10.2 For the binary tree `T2[A]` (Section 12.2.7), implement a `Traversable` instance for *right-to-left* depth-first traversal order. Use that to implement `zipWithIndex` for the type `T2[A]`. Verify that `zipWithIndex` transforms the tree into .

Exercise 12.2.10.3 Implement a `Traversable` instance for the data type `T3` defined in Exercise 11.2.9.2.

Exercise 12.2.10.4 Implement a `Functor` instance for the data type `TD` defined in Section 12.2.4. Use `TD`'s `map` method to re-implement the functions `addLeft` and `addRight` from that section. Show that the new implementations are equivalent to the old ones.

Exercise 12.2.10.5 Use the specialized version of fold_S for the binary tree `T2[A]` (Section 12.2.7) to compute the maximum depth of a tree:

```
def maxDepth[A](tree: T2[A]): Int = foldT2(???)(???)

scala> maxDepth(Branch(Leaf(0), Branch(Branch(Leaf(0), Leaf(0)), Leaf(0))))
res0: Int = 3
```

Exercise 12.2.10.6 For the data type `T3[A]` defined in Exercise 11.2.9.2, define a recursion scheme and implement a specialized version of `unfold` as `unfoldT3`. Using `unfoldT3`, write a function that generates ternary trees of the form starting from the given integer n at the root.

12.3 Laws and structure

To study the laws of the folding and traversing operations, it helps to choose simpler but equivalent versions of these operations. We have seen four methods that implement folding operations: `foldLeft`, `foldMap`, `reduce`, and `toList`. With suitable naturality laws, these methods are equivalent.

12.3.1 Equivalence of `reduce`, `foldLeft`, `foldMap`, and `toList`. Monoid morphisms

Scala's standard `reduce` method assumes a non-empty sequence and will fail otherwise:

```
def reduce[A](la: Seq[A])(red: (A, A) => A): A
```

For the purposes of this section, we will modify `reduce` to supply a default value for empty sequences:

```
def reduceE[A](la: Seq[A])(default: A)(red: (A, A) => A): A =
if (la.isEmpty) default else reduce(la)(red)
```

Having a default value of type A and a binary operation of type $A \times A \to A$ suggest that A may be a monoid (assuming that the monoid laws hold, see Example 8.2.5.6). So, we simplify the type signature of `reduceE` using a `Monoid` typeclass constraint:

```
def reduceE[M: Monoid](la: Seq[M]): M
```

After this change, let us compare the type signatures of the four methods:

```
def foldLeft[A, B](la: L[A])(init: B)(update: (B, A) => B): B
def foldMap[M: Monoid, A](f: A => M): L[A] => M
def reduceE[M: Monoid]: L[M] => M
def toList[A]: L[A] => List[A]
```

$$\text{foldLeft}_L : L^A \to B \to (B \times A \to B) \to B \quad , \qquad \text{foldMap}_L : (A \to M) \to L^A \to M \quad ,$$
$$\text{reduceE}_L : L^M \to M \quad , \qquad \text{toList}_L : L^A \to \text{List}^A \quad .$$

We will now show that these four functions are equivalent, assuming certain naturality laws.

The formulation of naturality laws is different for functions whose type parameters have typeclass constraints. For example, consider `reduceE` whose type parameter M is constrained to be a monoid. The naturality law for functions ϕ with type signature $\forall M. L^M \to M$ (but without the typeclass constraint) is:

$$\text{for all } f^{:M\to N} \quad : \qquad \phi^{:L^M\to M} \fatsemi f = f^{\uparrow L} \fatsemi \phi^{:L^N\to N} \quad .$$

Does this law hold with $\phi = \texttt{reduceE}$? Since `reduceE` would be used with type parameters M and N, those types have to be monoids for the law to make sense. Also, it turns out that the law must be used only with functions f that satisfy certain conditions with respect to the monoids M and N.

To see why, let us set $L = \texttt{List}$. It is reasonable to assume that `List`'s `reduceE` method should satisfy the naturality law, as long as we formulate that law correctly. We now apply `reduceE` to an empty list and to a list with two elements of type M:

$$\text{reduceE}\,([]) = e_M \quad , \qquad \text{reduceE}\,([x, y]) = x \oplus_M y \quad .$$

Similar values are found when `reduceE` is used with the monoid N. Then we have:

$$[] \triangleright \text{reduceE} \triangleright f = f(e_M) \quad , \qquad [] \triangleright f^{\uparrow L} \triangleright \text{reduceE} = [] \triangleright \text{reduceE} = e_N \quad ,$$
$$[x, y] \triangleright \text{reduceE} \triangleright f = (x \oplus_M y) \triangleright f = f(x \oplus_M y) \quad ,$$
$$[x, y] \triangleright f^{\uparrow L} \triangleright \text{reduceE} = [f(x), f(y)] \triangleright \text{reduceE} = f(x) \oplus_N f(y) \quad .$$

It follows that the law holds for `reduceE` only if the function f has the properties:

$$f(e_M) = e_N \quad , \qquad f(x \oplus_M y) = f(x) \oplus_N f(y) \quad .$$

These properties mean that $f : M \to N$ is compatible with the structure of the monoids M and N, in the sense that f maps the empty value e_M to the empty value e_N and M's binary operation to N's. Functions with these properties are called "monoid morphisms".

Definition 12.3.1.1 For any two monoids M and N, a function $f : M \to N$ is a **monoid morphism** if f satisfies the following laws:

$$\text{identity law}: \quad f(e_M) = e_N \quad ,$$
$$\text{composition law}: \quad f(x^{:M} \oplus_M y^{:M}) = f(x) \oplus_N f(y) \quad .$$

The **monoidal naturality law** of `reduceE` is then formulated as:

$$\begin{tikzcd} L^M \arrow[r, "\text{reduceE}^M"] \arrow[d, "f^{\uparrow L}"] & M \arrow[d, "f"] \\ L^N \arrow[r, "\text{reduceE}^N"] & N \end{tikzcd}$$

$$\text{reduceE}^M \fatsemi f^{:M\to N} = f^{\uparrow L} \fatsemi \text{reduceE}^N \quad . \tag{12.3}$$

Here the types M, N are arbitrary monoids and $f : M \to N$ is an arbitrary monoid morphism between M and N.

The monoidal naturality law expresses a programmer's expectation that the code of `reduceE[M]` should work in the same way for every monoid `M`. The code of `reduceE` should be fully parametric and may use the monoid `M`'s methods but may not, e.g., inspect the type parameter `M` via run-time reflection and make any decisions based on that.

With these definitions, we can now give a precise formulation of the following equivalences:

Statement 12.3.1.2 **(a)** The functions `foldMap` and `reduceE` are equivalent as long as `foldMap` satisfies the naturality law with respect to its type parameter A.

(b) The function `foldLeft` is equivalent to a new method we call `foldFn` ("fold with function"):

$$\text{foldFn} : L^{B\to B} \to B \to B \quad ,$$

as long as `foldLeft` satisfies the naturality law with respect to its type parameter A.

(c) The functions `foldFn` and `reduceE` are equivalent as long as `foldFn` obeys the laws (12.8) and (12.9) shown below and `reduceE` satisfies the monoidal naturality law.

We will show in Statement 12.3.1.3 below that the special laws (12.8) and (12.9) follow from parametricity. So, these laws will hold automatically when the code of `foldFn` is fully parametric. However, formulating these special laws allows us to prove the equivalence of `foldFn` and `reduceE` without assuming parametricity.

Proof **(a)** The equivalence of `foldMap` and `reduceE` follows from Statement 10.3.2.1 if we assume that `foldMap` satisfies the naturality law with respect to the type parameter A. To prove that equivalence, we just need to set $F^A \stackrel{\text{def}}{=} L^A$, $G^B \stackrel{\text{def}}{=} M$, and $H^B \stackrel{\text{def}}{=} M$ in Statement 10.3.2.1.

(b) We first reformulate `foldLeft`'s type signature in a form more similar to that of `foldMap`. We change the order of curried arguments in `foldLeft` and also replace the updater function of type $B \times A \to B$ by an equivalent curried function of type $A \to B \to B$ (similarly to what was done in Section 12.1.1). The result is a function we denote by `fldl`:

$$\text{fldl} : (A \to B \to B) \to L^A \to B \to B \quad , \qquad \text{fldl}\,(u^{:A\to B\to B})(p^{:L^A})(z^{:B}) \stackrel{\text{def}}{=} \text{foldLeft}\,(p)(z)(\tilde{u}) \quad ,$$
$$\text{where we defined :} \quad \tilde{u}^{:B\times A\to B} \stackrel{\text{def}}{=} b^{:B} \times a^{:A} \to u\,(a)\,(b) \quad .$$

The function `fldl` is equivalent to `foldLeft` since we only replaced some types with equivalent ones. Setting $F^A \stackrel{\text{def}}{=} L^A$, $G^A \stackrel{\text{def}}{=} B \to B$, and $H^A \stackrel{\text{def}}{=} B \to B$ in Statement 10.3.2.1, we obtain an equivalence between `fldl` and `foldFn`:

$$\text{foldFn} : L^{B\to B} \to B \to B \quad , \qquad \text{foldFn}\,(p^{:L^{B\to B}}) \stackrel{\text{def}}{=} \text{fldl}\,(\text{id}^{:(B\to B)\to B\to B})(p) \quad .$$

(c) The type signatures of `foldFn` and `reduceE` are similar except that `foldFn` uses the type $B \to B$ where `reduceE` uses an arbitrary monoid M (having an empty value e_M and a binary operation $\oplus_M$). The type of functions $B \to B$ is itself a monoid that we here denote by $\text{MF}^B \stackrel{\text{def}}{=} B \to B$. The empty value of that monoid is the identity function ($e_{\text{MF}^B} = \text{id}^{:B\to B}$), and the binary operation ($\oplus_{\text{MF}^B}$) is the *forward* function composition:

$$p^{:B\to B} \oplus_{\text{MF}^B} q^{:B\to B} \stackrel{\text{def}}{=} p \,⨾\, q \quad .$$

To relate `foldFn` and `reduceE`, we need to map an arbitrary monoid M to the monoid MF^M:

$$\text{inMF} : M \to \text{MF}^M \quad , \qquad \text{inMF}\,(m) \stackrel{\text{def}}{=} n^{:M} \to n \oplus_M m \quad .$$

The code of `inMF` is just the monoid M's binary operation ($\oplus_M$) in a flipped and curried form:

$$\text{inMF} : M \to M \to M \quad , \qquad \text{inMF} \stackrel{\text{def}}{=} m \to n \to n \oplus_M m \quad .$$

In particular, for any value $m^{:M}$ we have, due to the monoid M's identity law:

$$\text{inMF}\,(m)(e_M) = e_M \oplus m = m \quad . \tag{12.4}$$

The function `inMF` is a monoid morphism because the two laws of Definition 12.3.1.1 hold:

$$\text{inMF}\,(e_M) = n^{:M} \to n \oplus_M e_M = n^{:M} \to n = \text{id}^{:M\to M} = e_{\text{MF}^M} \quad ,$$
$$\text{inMF}\,(x \oplus_M y) = n^{:M} \to n \oplus_M x \oplus_M y \quad ,$$
$$\text{inMF}\,(x) \oplus_{\text{MF}^M} \text{inMF}\,(y) = \text{inMF}\,(x) \,⨾\, \text{inMF}\,(y) = (n \to n \oplus_M x) \,⨾\, (n \to n \oplus_M y)$$
$$= n \to n \oplus_M x \oplus_M y = \text{inMF}\,(x \oplus_M y) \quad .$$

So, we may use $f \stackrel{\text{def}}{=} \text{inMF}$ in the monoidal naturality law (12.3) with monoids M and $N \stackrel{\text{def}}{=} \text{MF}^M$:

$$\text{reduceE} \,⨾\, \text{inMF} = \text{inMF}^{\uparrow L} \,⨾\, \text{reduceE} \quad . \tag{12.5}$$

Now we can express `foldFn` and `reduceE` through each other:

$$\text{foldFn}^B = \text{reduceE}^{\text{MF}^B} \qquad \text{where} \qquad \text{MF}^B \stackrel{\text{def}}{=} B \to B \quad , \tag{12.6}$$

$$\text{reduceE}\,(p^{:L^M}) = \text{foldFn}\,(p \triangleright \text{inMF}^{\uparrow L})(e_M) \quad . \tag{12.7}$$

It remains to demonstrate an isomorphism between `foldFn` and `reduceE` in both directions.

1) For a given function `foldFn` with the type signature shown above, we define `reduceE` via Eq. (12.7) and then define a new `foldFn'` via Eq. (12.6). We need to show that `foldFn'` = `foldFn`:

$$\text{foldFn}'(p^{:L^{B\to B}}) = \text{reduceE}\,(p) = \text{foldFn}\,(p \triangleright \text{inMF}^{\uparrow L})(e_M) \quad .$$

Here the function `inMF` is used with the type signature corresponding to the monoid $M \stackrel{\text{def}}{=} B \to B$:

$$\text{inMF} : (B \to B) \to \text{MF}^{B\to B} \cong (B \to B) \to (B \to B) \to B \to B \quad ,$$
$$\text{inMF} \stackrel{\text{def}}{=} g^{:B\to B} \to h^{:B\to B} \to h \,⨾\, g \quad , \qquad e_{\text{MF}} \stackrel{\text{def}}{=} \text{id}^{:B\to B} \quad .$$

We find that $\text{foldFn}'(p) = \text{foldFn}\,(p)$ if the following equation holds:

$$\text{foldFn}^B(p^{:L^{B\to B}}) = \text{foldFn}^{B\to B}(p \triangleright \text{inMF}^{\uparrow L})(\text{id}^{:B\to B}) \quad . \tag{12.8}$$

It remains to prove that the law (12.3) will hold when `reduceE` is defined via Eq. (12.7):

$$\text{left-hand side}: \quad p^{:L^M} \triangleright \text{reduceE}^M \,⨾\, f^{:M\to N} = \big(\text{foldFn}\,(p \triangleright \text{inMF}^{\uparrow L})(e_M)\big) \triangleright f \quad ,$$
$$\text{right-hand side}: \quad p^{:L^M} \triangleright f^{\uparrow L} \,⨾\, \text{reduceE}^N = \text{foldFn}\,(p \triangleright f^{\uparrow L} \triangleright \text{inMF}^{\uparrow L})(e_N) \quad .$$

The two sides are equal if the following law holds for any $p^{:L^M}$ and any monoid morphism $f^{:M\to N}$:

$$e_M \triangleright \big(p \triangleright \text{inMF}^{\uparrow L} \triangleright \text{foldFn}\big) \triangleright f = e_N \triangleright \big(p \triangleright f^{\uparrow L} \triangleright \text{inMF}^{\uparrow L} \triangleright \text{foldFn}\big) \quad . \tag{12.9}$$

Since both laws (12.8) and (12.9) are assumed to hold, we have demonstrated the equivalence between `reduceE` and `foldFn` in one direction.

2) For a given function `reduceE` with the type signature shown above, we define `foldFn` via Eq. (12.6) and then define a new `reduceE'` via Eq. (12.7). We need to show that `reduceE'` = `reduceE`:

$$\begin{aligned}
\text{expect to equal reduceE}\,(p): \quad & \text{reduceE}'(p^{:L^M}) = \text{foldFn}\,(p \triangleright \text{inMF}^{\uparrow L})(e_M) = \text{reduceE}\,(p \triangleright \text{inMF}^{\uparrow L})(e_M) \\
\text{rewrite using } \triangleright\text{-notation}: \quad & = e_M \triangleright \big(p \triangleright \underline{\text{inMF}^{\uparrow L} \,⨾\, \text{reduceE}}\big) \\
\text{use Eq. (12.5)}: \quad & = e_M \triangleright \big(p \triangleright \text{reduceE} \,⨾\, \text{inMF}\big) = \underline{\text{inMF}}\,(p \triangleright \text{reduceE})\underline{(e_M)} \\
\text{use Eq. (12.4)}: \quad & = p \triangleright \text{reduceE} = \text{reduceE}\,(p) \quad .
\end{aligned}$$

It remains to prove that `foldFn` defined via Eq. (12.6) will always satisfy the laws (12.8) and (12.9). To verify that the law (12.8) holds:

$$\begin{aligned}
\text{expect to equal foldFn}^B(p): \quad & \text{foldFn}^{B\to B}(p^{:L^{B\to B}} \triangleright \text{inMF}^{\uparrow L})(e_{\text{MF}}) = \text{reduceE}\,(p \triangleright \text{inMF}^{\uparrow L})(e_{\text{MF}}) \\
& = e_{\text{MF}} \triangleright (p \triangleright \underline{\text{inMF}^{\uparrow L} \,⨾\, \text{reduceE}}) \\
\text{use Eq. (12.5)}: \quad & = e_{\text{MF}} \triangleright \big(p \triangleright \text{reduceE} \,⨾\, \text{inMF}\big) = \underline{\text{inMF}}\,(p \triangleright \text{reduceE})\underline{(e_{\text{MF}})} \\
\text{use Eq. (12.4)}: \quad & = p \triangleright \text{reduceE} = \text{reduceE}^{B\to B}(p) = \text{foldFn}^B(p) \quad .
\end{aligned}$$

To verify the law (12.9), write the two sides separately and substitute Eq. (12.6):

$$\begin{aligned}
\text{left-hand side}: \quad & e_M \triangleright (p \triangleright \underline{\text{inMF}^{\uparrow L} \triangleright \text{foldFn}}) \triangleright f = e_M \triangleright (p \triangleright \underline{\text{inMF}^{\uparrow L} \fatsemi \text{reduceE}}) \triangleright f \\
\text{use Eq. (12.5)}: \quad & = e_M \triangleright (p \triangleright \text{reduceE} \fatsemi \text{inMF}) \triangleright f = (\underline{\text{inMF}}\,(p \triangleright \text{reduceE})\underline{(e_M)}) \triangleright f \\
\text{use Eq. (12.4)}: \quad & = p \triangleright \text{reduceE} \triangleright f \quad , \\
\text{right-hand side}: \quad & e_N \triangleright (p \triangleright \underline{f^{\uparrow L} \triangleright \text{inMF}^{\uparrow L} \triangleright \text{foldFn}}) = e_N \triangleright (p \triangleright f^{\uparrow L} \fatsemi \underline{\text{inMF}^{\uparrow L} \fatsemi \text{reduceE}}) \\
\text{use Eq. (12.5)}: \quad & = e_N \triangleright (p \triangleright \underline{f^{\uparrow L} \fatsemi \text{reduceE}} \fatsemi \text{inMF}) \\
\text{use Eq. (12.3)}: \quad & = e_N \triangleright (p \triangleright \text{reduceE} \fatsemi f \fatsemi \text{inMF}) = \underline{\text{inMF}}\,(p \triangleright \text{reduceE} \fatsemi f)\underline{(e_N)} \\
\text{use Eq. (12.4)}: \quad & = p \triangleright \text{reduceE} \triangleright f \quad .
\end{aligned}$$

The two sides are now equal.

This proves that the equivalence between `reduceE` and `foldFn` holds also in the other direction. □

Let us clarify the meaning of the special laws (12.8) and (12.9). We will show below that these laws are a consequence of parametricity. In other words, `foldFn(p)` uses any functions of type $B \to B$ contained in $p : L^{B \to B}$ in a fully parametric manner. The code of `foldFn` cannot inspect the type B or the code of the functions of type $B \to B$. The only way of using those functions is to compose them with each other in some order, obtaining again a function of type $B \to B$. The laws (12.8) and (12.9) express this property in different ways by writing the function composition explicitly as part of the code of the function `inMF`.

Statement 12.3.1.3 Denote for brevity $E^A \stackrel{\text{def}}{=} A \to A$. Let L be any *polynomial* functor. (This restriction is acceptable because, as we will see below, only polynomial functors are traversable.)

(a) Any fully parametric function $\phi : \forall A.\, L^{E^A} \to E^A$ satisfies the law (12.8):

$$\phi\,(p^{:L^{E^A}}) = \phi\,(p \triangleright \text{inF}^{\uparrow L})(\text{id}^{:A \to A}) \quad , \tag{12.10}$$

$$\text{where we defined}: \quad \text{inF} : E^A \to E^A \to E^A \quad , \qquad \text{inF} \stackrel{\text{def}}{=} h^{:A \to A} \to k^{:A \to A} \to k \fatsemi h \quad .$$

(b) Any fully parametric function $\text{foldFn} : \forall A.\, L^{E^A} \to E^A$ satisfies Eq. (12.9).

Proof **(a)** Example D.3.7.4 (in Appendix D) shows that ϕ satisfies the strong dinaturality law. The law says that for any $f^{:B \to A}$, $x^{:L^{A \to A}}$, and $y^{:L^{B \to B}}$:

$$\text{if } x \triangleright (h^{:A \to A} \to f \fatsemi h)^{\uparrow L} = y \triangleright (g^{:B \to B} \to g \fatsemi f)^{\uparrow L} \quad \text{then} \quad f \fatsemi \phi(x) = \phi(y) \fatsemi f \quad . \tag{12.11}$$

We need to choose A, B, and f appropriately in order to be able to derive Eq. (12.10). After some guessing and trying, one finds that a good choice is $B \stackrel{\text{def}}{=} A \to A$ and $f^{:B \to A}$ defined by:

$$f : (A \to A) \to A \quad , \qquad f \stackrel{\text{def}}{=} k^{:A \to A} \to k(a_0) \quad \text{with a fixed } a_0 : A \quad . \tag{12.12}$$

Each value a_0 of type A gives us a function $f_{a_0} : (A \to A) \to A$ for which the law (12.11) holds.

We will now show that Eq. (12.10) follows from Eq. (12.11) with this choice of f. The value a_0 will be held fixed throughout most of this proof, so we will write just f instead of f_{a_0}.

In order to use Eq. (12.11) with this choice of f, we need to find suitable values x and y. Note that the conclusion of Eq. (12.11) relates $\phi(x)$ with $\phi(y)$, while Eq. (12.10) requires us to relate $\phi(p)$ with $\phi(p \triangleright \text{inF}^{\uparrow L})$. So, we must choose $x = p$ and $y = p \triangleright \text{inF}^{\uparrow L}$. This choice would work with Eq. (12.11) only if its precondition is satisfied with these x and y and with f defined by Eq. (12.12):

$$\begin{aligned}
& p \triangleright (h^{:A \to A} \to f \fatsemi h)^{\uparrow L} \stackrel{?}{=} y \triangleright (g^{:B \to B} \to g \fatsemi f)^{\uparrow L} = p \triangleright \text{inF}^{\uparrow L} \triangleright (g^{:B \to B} \to g \fatsemi f)^{\uparrow L} \\
& = p \triangleright \left(\text{inF} \fatsemi (g^{:B \to B} \to g \fatsemi f)\right)^{\uparrow L} \quad .
\end{aligned}$$

We will show that this equation holds for any $p : L^{A \to A}$ if we show that:

$$h^{:A \to A} \to f \fatsemi h \stackrel{?}{=} \text{inF} \fatsemi (g^{:B \to B} \to g \fatsemi f) \quad .$$

We simplify separately each side of this equation and find that they are equal:

$$\begin{aligned}
\text{left-hand side}: \quad & h^{:A\to A} \to f \,⨾\, h = h \to (k \to \underline{k(a_0)) \,⨾\, h} = h \to k \to h(k(a_0)) \quad , \\
\text{right-hand side}: \quad & \underline{\text{inF}} \,⨾\, (g^{:B\to B} \to g \,⨾\, \underline{f}) = (h \to k \to k \,⨾\, h) \,⨾\, (g \to g \,⨾\, (k \to k(a_0)) \\
\text{compute composition}: \quad & = h \to (k \to k \,⨾\, h) \,⨾\, (k \to k(a_0)) = h \to k \to \underline{(k \,⨾\, h)(a_0)} \\
& = h \to k \to h(k(a_0) \quad .
\end{aligned}$$

Since the precondition of Eq. (12.11) is satisfied, its conclusion holds too:

$$\begin{aligned}
& f \,⨾\, \phi(x) \stackrel{!}{=} \phi(y) \,⨾\, f \quad , \\
\text{equivalently}: \quad & (k^{:A\to A} \to k(a_0)) \,⨾\, \phi(p) \stackrel{!}{=} \phi(p \triangleright \text{inF}^{\uparrow L}) \,⨾\, (k^{:A\to A} \to k(a_0)) \quad , \\
\text{equivalently}: \quad & k^{:A\to A} \to \phi(p)(k(a_0)) \stackrel{!}{=} k^{:A\to A} \to \phi(p \triangleright \text{inF}^{\uparrow L})(k)(a_0) \quad .
\end{aligned}$$

The last equation can be applied to any function k of type $A \to A$:

$$\phi(p)(k(a_0)) \stackrel{!}{=} \phi(p \triangleright \text{inF}^{\uparrow L})(k)(a_0) \quad .$$

In Eq. (12.10), the function $\phi(p \triangleright \text{inF}^{\uparrow L})$ is applied to the identity function of type $A \to A$. So, we set $k \stackrel{\text{def}}{=} \text{id}$ in the line above:

$$\phi(p)(a_0) \stackrel{!}{=} \phi(p \triangleright \text{inF}^{\uparrow L})(\text{id})(a_0) \quad .$$

Since this holds for any a_0, we get $\phi(p) \stackrel{!}{=} \phi(p \triangleright \text{inF}^{\uparrow L})(\text{id})$, which proves Eq. (12.10).

(b) We already showed in part **(a)** that `foldFn` $= \phi$ satisfies Eq. (12.11). We will now show that Eq. (12.9) follows if we apply Eq. (12.11) with A, B, x, and y chosen as:

$$A \stackrel{\text{def}}{=} N \quad , \qquad B \stackrel{\text{def}}{=} M \quad , \qquad x^{:L^{N\to N}} \stackrel{\text{def}}{=} p \triangleright f^{\uparrow L} \triangleright \text{inMF}^{\uparrow L} \quad , \qquad y^{:L^{M\to M}} \stackrel{\text{def}}{=} p \triangleright \text{inMF}^{\uparrow L} \quad .$$

Before we can use Eq. (12.11), we need to verify that its precondition is satisfied:

$$x \triangleright (h \to f \,⨾\, h)^{\uparrow L} = p \triangleright \left(f \,⨾\, \text{inMF} \,⨾\, (h \to f \,⨾\, h)\right)^{\uparrow L} \stackrel{?}{=} y \triangleright (g \to g \,⨾\, f)^{\uparrow L} = p \triangleright \left(\text{inMF} \,⨾\, (g \to g \,⨾\, f)\right)^{\uparrow L} \quad .$$

To show that this equation holds, we compare separately the lifted functions that are applied to p:

$$\begin{aligned}
& f \,⨾\, \text{inMF} \,⨾\, (h \to f \,⨾\, h) = \left(m^{:M} \to n^{:N} \to n \oplus_N f(m)\right) \,⨾\, (h \to f \,⨾\, h) = m^{:M} \to l^{:M} \to f(l) \oplus_N f(m) \quad , \\
& \text{inMF} \,⨾\, (g \to g \,⨾\, f) = (m^{:M} \to l^{:M} \to l \oplus_M m) \,⨾\, (g \to g \,⨾\, f) = m^{:M} \to l^{:M} \to f(l \oplus_M m) \quad .
\end{aligned}$$

The two functions are equal because of the composition law of the monoid morphism f. So, the precondition of Eq. (12.11) holds, and we may use its conclusion:

$$f \,⨾\, \text{foldFn}\,(x) \stackrel{!}{=} \text{foldFn}(y) \,⨾\, f \quad \text{or equivalently}: \quad \text{foldFn}\,(p \triangleright \text{inMF}^{\uparrow L}) \,⨾\, f = f \,⨾\, \text{foldFn}(p \triangleright f^{\uparrow L} \,⨾\, \text{inMF}^{\uparrow L}).$$

This gives the law (12.9) if we use f's monoid morphism identity law: $e_M \triangleright f = e_N$.

Statement 12.3.1.4 Functions `reduceE` and `toList` are equivalent if expressed through each other as:

$$\text{toList} : L^A \to \text{List}^A \quad , \qquad \text{toList} = \text{pu}^{\uparrow L}_{\text{List}} \,⨾\, \text{reduceE}^{\text{List}^A} \quad , \tag{12.13}$$

$$\text{reduceE} : L^M \to M \quad , \qquad \text{reduceE} = \text{toList} \,⨾\, \text{reduceList} \quad , \tag{12.14}$$

assuming the naturality law of `toList` and the monoidal naturality law (12.3) of `reduceE`. Here the function $\text{reduceE}^{\text{List}^A}$ is `reduceE` applied to the monoidal type `List[A]` whose binary operation is the list concatenation. The monoidally natural function `reduceList[M]` is defined for monoids M by:

```
def reduceList[M: Monoid]: List[M] => M = {
  case Nil            => Monoid[M].empty
  case head :: tail   => head |+| reduceList(tail)
}
```

$$\text{reduceList}^M : \text{List}^M \to M \quad , \qquad \text{reduceList} \stackrel{\text{def}}{=} \left|\begin{array}{c|c} & M \\ \hline \mathbb{1} & _ \to e_M \\ M \times \text{List}^M & h^{:M} \times t^{:\text{List}^M} \to h \oplus_M \overline{\text{reduceList}}\,(t) \end{array}\right| \quad .$$

Proof We begin by showing that `reduceList` obeys the monoidal naturality law:

$$f^{\uparrow \text{List}} ⨾ \text{reduceList} = \text{reduceList} ⨾ f \quad .$$

This law assumes that $f^{:M \to N}$ is a monoid morphism between two monoids M and N.

Simplify the left-hand side of the law:

$$f^{\uparrow \text{List}} ⨾ \text{reduceList} = \left|\begin{array}{c|cc} & \mathbb{1} & N \times \text{List}^N \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ M \times \text{List}^M & \mathbb{0} & f \boxtimes f^{\uparrow \text{List}} \end{array}\right| ⨾ \left|\begin{array}{c|c} & N \\ \hline \mathbb{1} & _ \to e_N \\ N \times \text{List}^N & h \times t \to h \oplus_N (t \triangleright \overline{\text{reduceList}}) \end{array}\right|$$

$$= \left|\begin{array}{c|c} & N \\ \hline \mathbb{1} & _ \to e_N \\ M \times \text{List}^M & h \times t \to f(h) \oplus_N (t \triangleright f^{\uparrow \text{List}} ⨾ \overline{\text{reduceList}}) \end{array}\right| \quad .$$

We can use the inductive assumption that recursive calls to `reduceList` already obey the monoidal naturality law. So, we can simplify:

$$t \triangleright f^{\uparrow \text{List}} ⨾ \overline{\text{reduceList}} = t \triangleright \overline{\text{reduceList}} ⨾ f = f(\overline{\text{reduceList}}\,(t)) \quad .$$

The bottom-row expression in the last matrix is then rewritten to:

$$f(h) \oplus_N (t \triangleright f^{\uparrow \text{List}} ⨾ \overline{\text{reduceList}}) = f(h) \oplus_N f(\overline{\text{reduceList}}\,(t))$$

$$\text{monad morphism law}: \quad = f\big(h \oplus_M \overline{\text{reduceList}}\,(t)\big) \quad .$$

Using the monad morphism identity law ($f(e_M) = e_N$), we find:

$$f^{\uparrow \text{List}} ⨾ \text{reduceList} = \left|\begin{array}{c|c} & N \\ \hline \mathbb{1} & _ \to f(e_M) \\ M \times \text{List}^M & h \times t \to f\big(h \oplus_M \overline{\text{reduceList}}\,(t)\big) \end{array}\right|$$

$$= \left|\begin{array}{c|c} & N \\ \hline \mathbb{1} & _ \to e_M \\ M \times \text{List}^M & h \times t \to h \oplus_M \overline{\text{reduceList}}\,(t) \end{array}\right| ⨾ f = \text{reduceList} ⨾ f \quad .$$

This is equal to the right-hand side of the law.

We are now ready to show the isomorphism between `reduceE` and `toList` in both directions:

(1) Given a function `reduceE` that satisfies Eq. (12.3), define `toList` via Eq. (12.13) and then define a new function `reduceE'` via Eq. (12.14). To show that `reduceE'` = `reduceE`, we write:

$$\text{reduceE}' = \text{toList} ⨾ \text{reduceList} = \underline{\text{pu}_{\text{List}}^{\uparrow L} ⨾ \text{reduceE}^{\text{List}^M}} ⨾ \text{reduceList}$$

$$\text{Eq. (12.3) with } N \stackrel{\text{def}}{=} \text{List}^A: \quad = \text{reduceE}^M ⨾ \text{pu}_{\text{List}} ⨾ \text{reduceList} \quad .$$

It remains to show that the composition $\text{pu}_{\text{List}} ⨾ \text{reduceList}$ is equal to the identity function:

$$\text{pu}_{\text{List}} ⨾ \text{reduceList} = \text{id}^{:M \to M} \quad .$$

We use the definition of `reduceList[M]` and simplify the matrix composition:

$$\text{pu}_{\text{List}} ⨾ \text{reduceList} = (m^{:M} \to \mathbb{0} + m \times []) ⨾ \text{reduceList}$$

$$= \left| \begin{array}{c|c} & \begin{array}{cc} \mathbb{1} & M \times \text{List}^M \end{array} \\ \hline M & \begin{array}{cc} \mathbb{0} & m \to m \times [] \end{array} \end{array} \right| ⨾ \left| \begin{array}{c|c} & M \\ \hline \begin{array}{c} \mathbb{1} \\ M \times \text{List}^M \end{array} & \begin{array}{c} _ \to e_M \\ h \times t \to h \oplus_M \overline{\text{reduceList}}\,(t) \end{array} \end{array} \right|$$

$$= \left| \begin{array}{c|c} & M \\ \hline M & m \to m \oplus_M \underline{\overline{\text{reduceList}}\,([])} \end{array} \right| = m \to \underline{m \oplus e_M} = m \to m = \text{id} \quad .$$

In the last line, we used the fact that `reduceList[M]` applied to an empty list gives e_M.

(2) Given a function `toList`, define `reduceE` via Eq. (12.14) and then define a new function `toList'` via Eq. (12.13). To show that `toList'` = `toList`, we write:

$$\text{toList}' = \text{pu}^{\uparrow L}_{\text{List}} ⨾ \text{reduceE}^{\text{List}^A} = \underline{\text{pu}^{\uparrow L}_{\text{List}} ⨾ \text{toList}} ⨾ \text{reduceList}^{\text{List}^A}$$

naturality of toList : $= \text{toList} ⨾ \text{pu}^{\uparrow \text{List}}_{\text{List}} ⨾ \text{reduceList}^{\text{List}^A} \quad .$

It remains to show that the composition $\text{pu}^{\uparrow \text{List}}_{\text{List}} ⨾ \text{reduceList}$ is equal to the identity function:

$$\text{pu}^{\uparrow \text{List}}_{\text{List}} ⨾ \text{reduceList} = \text{id}^{:\text{List}^M \to \text{List}^M} \quad . \tag{12.15}$$

We use the definition of `reduceList[M]` and set `M` to the monoid type `List[A]`:

$$\text{pu}^{\uparrow \text{List}}_{\text{List}} ⨾ \text{reduceList}^{\text{List}^A}$$

$$= \left| \begin{array}{c|c} & \begin{array}{cc} \mathbb{1} & \text{List}^M \times \text{List}^{\text{List}^M} \end{array} \\ \hline \begin{array}{c} \mathbb{1} \\ M \times \text{List}^M \end{array} & \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h \times t \to \text{pu}_{\text{List}}(h) \times (t ▷ \text{pu}^{\uparrow \text{List}}_{\text{List}}) \end{array} \end{array} \right| ⨾ \left| \begin{array}{c|c} & \text{List}^M \\ \hline \begin{array}{c} \mathbb{1} \\ \text{List}^M \times \text{List}^{\text{List}^M} \end{array} & \begin{array}{c} _ \to 1 + \mathbb{0} \\ h \times t \to h +\!\!+ \overline{\text{reduceList}}\,(t) \end{array} \end{array} \right|$$

$$= \left| \begin{array}{c|c} & \text{List}^M \\ \hline \begin{array}{c} \mathbb{1} \\ M \times \text{List}^M \end{array} & \begin{array}{c} _ \to 1 + \mathbb{0} \\ h \times t \to \text{pu}_{\text{List}}(h) +\!\!+ (t ▷ \text{pu}^{\uparrow \text{List}}_{\text{List}} ▷ \overline{\text{reduceList}}) \end{array} \end{array} \right| \quad .$$

By the inductive assumption, a recursive call to `reduceList` already satisfies Eq. (12.15):

$$t ▷ \underline{\text{pu}^{\uparrow \text{List}}_{\text{List}} ▷ \overline{\text{reduceList}}} = t ▷ \text{id} = t \quad .$$

We can now simplify the code and obtain the identity function:

$$\text{pu}^{\uparrow \text{List}}_{\text{List}} ⨾ \text{reduceList}^{\text{List}^A} = \left| \begin{array}{c|c} & \text{List}^M \\ \hline \begin{array}{c} \mathbb{1} \\ M \times \text{List}^M \end{array} & \begin{array}{c} _ \to 1 + \mathbb{0} \\ h^{:M} \times t^{:\text{List}^M} \to \text{pu}_{\text{List}}(h) +\!\!+ t \end{array} \end{array} \right|$$

$$= \left| \begin{array}{c|c} & \text{List}^M \\ \hline \begin{array}{c} \mathbb{1} \\ M \times \text{List}^M \end{array} & \begin{array}{c} _ \to 1 + \mathbb{0} \\ h^{:M} \times t^{:\text{List}^M} \to \mathbb{0} + h \times t \end{array} \end{array} \right| = \left| \begin{array}{c|c} & \begin{array}{cc} \mathbb{1} & M \times \text{List}^M \end{array} \\ \hline \begin{array}{c} \mathbb{1} \\ M \times \text{List}^M \end{array} & \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h^{:M} \times t^{:\text{List}^M} \to h \times t \end{array} \end{array} \right|$$

$$= \text{id} \quad .$$

It remains to prove that `reduceE` will satisfy the monoidal naturality law (12.3) when defined via Eq. (12.14) through any `toList` function. For any monoid morphism $f : M \to N$ between arbitrary monoids M and N, we write the two sides of the monoidal naturality law (12.3):

$$\begin{aligned}
\text{left-hand side}: \quad & \text{reduceE} ⨾ f = \text{toList} ⨾ \underline{\text{reduceList} ⨾ f} \\
\text{monoidal naturality of reduceList}: \quad & = \underline{\text{toList} ⨾ f^{\uparrow \text{List}}} ⨾ \text{reduceList} \\
\text{naturality of toList}: \quad & = f^{\uparrow L} ⨾ \text{toList} ⨾ \text{reduceList} \quad , \\
\text{right-hand side}: \quad & f^{\uparrow L} ⨾ \text{reduceE} = f^{\uparrow L} ⨾ \text{toList} ⨾ \text{reduceList} \quad .
\end{aligned}$$

The two sides are now equal. □

12.3.2 The missing laws of `foldMap` and `reduce`

The equivalence of `foldLeft`, `foldMap`, `reduce`, and `toList` is ensured by assuming various naturality laws. However, those naturality laws do not fully describe the programmer's expectations about the behavior of folding operations.

To see why, let us consider `toList` with the type signature $L^A \to \text{List}^A$. If `toList` is fully parametric, it will satisfy the naturality law:

$$(f^{:A\to B})^{\uparrow L} ⨾ \text{toList} = \text{toList} ⨾ f^{\uparrow \text{List}} \quad .$$

This law describes the property that `toList[A]` works in the same way for all types `A`. Certainly, programmers expect this property to hold. But the main intent of `toList` is to extract values of type A out of L^A and store them in a list. Naturality laws do not express this intent.

More precisely, programmers expect that for any "container" value `p` of type `L[A]`, the value `toList(p)` should be a list of *all* values of type `A` stored in `p`. As an example of unexpected behavior, imagine implementing the type signature of `toList` by a function that always returns an empty list. For "containers" that store some values of type `A`, this implementation (although it is fully parametric) would be unacceptable since it loses information: `toList(x)` always ignores its argument `x`.

To prohibit such implementations, we would like to impose a law that holds only when `toList` extracts every value of type `A` stored in `p`. Unfortunately, it seems to be impossible to express this property in the form of an equation satisfied by `toList`. Often, a loss of information is prevented by imposing an identity law. Can we formulate an identity law for `toList`? Such a law could state that `toList(p)` should extract some known values contained in `p`. If L were a pointed functor (see Section 8.3.5), we could use its `pure` method to inject a known value `x` into the container `p` and then require `toList` to extract the same value `x`. But the definition of `toList` does not require the functor L to be pointed. So, in general we cannot inject values into L^A in a way that is guaranteed to preserve information. This prevents us from formulating an identity law for `toList`.

Another approach to finding laws is to look for type signatures in the form of a "lifting" that transforms a function of one type into a function of another type. We have summarized the methods of several standard typeclasses as "liftings" in Section 11.5.5. The laws of a "lifting" are the functor laws (identity and composition). Could we apply this approach to the folding operations? The type signature of the `foldMap` function is:

$$\text{foldMap} : (A \to M) \to L^A \to M \quad .$$

This resembles a "lifting" from functions of type $A \to M$ to functions of type $L^A \to M$. However, it is not possible to impose the functor laws on those liftings. For instance, the functor composition law involves applying `foldMap` to a composition of arguments. But the type signature $A \to M$ (where M is a fixed type) does not support composition since we cannot compose $A \to M$ with $B \to M$.

So, the "lifting" approach also fails to yield a suitable law for folding operations. However, we will see below that lifting-like laws may be imposed on the `traverse` operation, whose type signature is

a generalization of that of `foldMap`. We will show that the laws of `traverse` forbid information-losing implementations. Since `foldMap` can be derived from `traverse`, we may take the position that the only acceptable implementations of `foldMap` are those derived from a lawful `traverse` function.

12.3.3 All polynomial functors are foldable

It turns out that the lack of available laws does not prevent us from finding correct implementations of folding operations. The reason is that folding operations are available only for polynomial functors, such as `Option[A]` and `List[A]` (note that `List` is a recursively defined polynomial functor). These functors represent containers that store a finite number of values of type `A`. So, it is clear what it means to extract "all values of type A" from such containers. To show that all polynomial functors are foldable, we will define `toList` inductively via structural analysis. The definition will ensure that `toList` extracts exactly as many values as stored in the polynomial functor.

Let us first show that non-polynomial functors are *not* foldable. An example of a non-polynomial functor is the `Reader` monad:

$$\text{Reader}^{R,A} \stackrel{\text{def}}{=} R \to A \quad ,$$

where R is a fixed but arbitrarily chosen type. A value p of type $R \to A$ can be viewed as a container that stores one value of type A for each value of type R. We can extract all values of type A from p only if we can enumerate all possible values of type R. In general, it is not practical to enumerate all values of a given type (as an example, consider the type $R \stackrel{\text{def}}{=} \text{String} \to \text{String}$). So, we will not be able to implement `toList`, `foldMap`, `foldLeft`, or `reduce` for the `Reader` monad. The only exceptions are types R that have a known finite set of distinct values, such as $R = \texttt{Boolean}$. However, in those cases the type $R \to A$ is equivalent to a polynomial functor:

$$\text{Reader}^{\text{Boolean},A} = \mathbb{2} \to A \cong A \times A \quad .$$

We are now ready to show that all polynomial functors are foldable. It is convenient to use the `toList` operation to define the "foldable functor" typeclass:

```
trait Foldable[L[_]] {
  def toList[A]: L[A] => List[A]
}
```

Other folding operations (`foldLeft`, `foldMap`, `reduce`) can be derived from `toList` (Section 12.3.1).

Polynomial functors are built via the five standard type constructions (Table 5.1.2 without the function types). Defining `toList` for these constructions will provide an implementation of folding operations for all polynomial functors.

Fixed type Constant functors $L^A \stackrel{\text{def}}{=} Z$, where Z is a fixed type, are viewed as containers that are always empty and cannot store any values of type A. So, `toList` for L always returns an empty list.

Type parameter The identity functor $L^A \stackrel{\text{def}}{=} A$ is viewed as a container holding a single value of type A. So, we simply define $\text{toList} \stackrel{\text{def}}{=} \text{id}$.

Products If K and L are foldable functors that support toList_K and toList_L, we define the product $M^A \stackrel{\text{def}}{=} K^A \times L^A$ and the corresponding function toList_M as:

```
def toList_M[A]: ((K[A], L[A])) => List[A] = { case (p, q) => toList_K(p) ++ toList_L(q) }
```

$$\text{toList}_M : K^A \times L^A \to \text{List}^A \quad , \qquad \text{toList}_M \stackrel{\text{def}}{=} p^{:K^A} \times q^{:L^A} \to \text{toList}_K(p) +\!\!+ \text{toList}_L(q) \quad .$$

This implementation contains an arbitrary choice: the values stored in p are listed before the values stored in q. We could equally well write $\text{toList}_L(q) +\!\!+ \text{toList}_K(p)$ in the function body. This arbitrary choice corresponds to the arbitrariness of the order in which a folding operation may traverse the values stored in a container. Different instances of the `Foldable` typeclass may define different traversal orders, as we have seen in the code examples earlier in this chapter.

Co-products If K and L are foldable functors, we define the co-product $M^A \stackrel{\text{def}}{=} K^A + L^A$ and the corresponding function $\text{toList}_M : K^A + L^A \to \text{List}^A$ as:

```
def toList_M[A]: Either[K[A], L[A]] => List[A] = {
  case Left(p)    => toList_K(p)
  case Right(q)   => toList_L(q)
}
```

$$\text{toList}_M \stackrel{\text{def}}{=} \begin{array}{|c||c|} & \text{List}^A \\ \hline K^A & \text{toList}_K \\ L^A & \text{toList}_L \end{array} \quad .$$

Recursive types We need to implement `toList` for a functor L defined recursively by $L^A \stackrel{\text{def}}{=} S^{A,L^A}$, where the recursion scheme $S^{\bullet,\bullet}$ is a bifunctor that is itself foldable. A bifunctor $S^{\bullet,\bullet}$ is **foldable** if there is a `toList` function with this type signature:

$$\text{toList}_S : S^{A,A} \to \text{List}^A \quad .$$

We define toList_L using toList_S and the bifunctor S's liftings (denoted by `mapS1` and `mapS2`):

```
def toList_L[A]: S[A, L[A]] => List[A] = _.mapS2(toList_L).mapS1(List(_)).toList.flatten
```

$$\text{toList}_L : S^{A,L^A} \to \text{List}^A \quad , \qquad \text{toList}_L \stackrel{\text{def}}{=} \overline{\text{toList}_L}^{\uparrow S^{A,\bullet}} ⨾ \text{pu}_{\text{List}}^{\uparrow S^{\bullet,\text{List}^A}} ⨾ \text{toList}_S ⨾ \text{ftn}_{\text{List}} \quad .$$

This code first converts values of type S^{A,L^A} into type S^{A,List^A} by lifting `toList` recursively to the second type parameter of $S^{\bullet,\bullet}$. We then convert S^{A,List^A} into $S^{\text{List}^A,\text{List}^A}$ by creating one-element lists via pu_{List}. After that, all values of type A are extracted from $S^{\text{List}^A,\text{List}^A}$ using toList_S, which returns a nested $\text{List}^{\text{List}^A}$. The final `flatten` operation reduces that to a List^A.

This implements `toList` for the five type constructions that build up polynomial functors. In each case, `toList` extracts all stored values of type A. One could write a code generator to provide a `Foldable` instance for any polynomial functor automatically. When writing code by hand, it is more convenient to implement folding operations directly, as we did at the beginning of this chapter.

12.3.4 Equivalence of `traverse` and `sequence`

The standard Scala library contains the method `Future.sequence` in addition to `Future.traverse`. The main use of `Future.sequence` is for transforming a sequence of `Future` values that may run in parallel. The result is a single `Future` value that waits until all parallel computations are finished. Omitting some details, we may write the type signature of `Future.sequence` as:

```
def sequence[A, L[X] <: TraversableOnce[X]]: L[Future[A]] => Future[L[A]]
```

The function `Future.sequence` is limited to sequence-like data types such as `L[A] = List[A]`. To generalize the `sequence` operation to other data types, we replace a sequence-like type by an arbitrary traversable functor `L[A]`. We also replace the `Future` type constructor by an arbitrary applicative functor (since the `traverse` operation accepts one). It turns out that a function called `sequence` can then be defined via the `traverse` operation and vice versa:

```
def sequence[A, L[_]: Traversable, F[_]: Applicative : Functor]: L[F[A]] => F[L[A]] = _.traverse(id)
def traverse[A, B, L[_]: Traversable, F[_]: Applicative : Functor](la: L[A])(f: A => F[B]): F[L[B]]
    = sequence(la.map(f))
```

$$\text{seq}_L : L^{F^A} \to F^{L^A} \quad , \qquad \text{seq}_L \stackrel{\text{def}}{=} \text{trav}_L(\text{id}) \quad ,$$
$$\text{trav}_L : (A \to F^B) \to L^A \to F^{L^B} \quad , \qquad \text{trav}_L(f) \stackrel{\text{def}}{=} f^{\uparrow L} ⨾ \text{seq}_L \quad .$$

An inverse transformation ($F^{L^A} \to L^{F^A}$) is not always possible, as shown in Exercise 12.3.9.5. The example with $F = \texttt{Future}$ and $L = \texttt{List}$ may help remember the correct type signature of `sequence`.

The equivalence between `traverse` and `sequence` holds under the assumption that `traverse` obeys a naturality law with respect to the type parameter A. This follows from Statement 10.3.2.1 where we need to define $\text{tr} \stackrel{\text{def}}{=} \text{seq}_L$, $\text{ftr} \stackrel{\text{def}}{=} \text{trav}_L$, $F \stackrel{\text{def}}{=} L$, $G \stackrel{\text{def}}{=} F$, and $H \stackrel{\text{def}}{=} F \circ L$.

Because of this equivalence, we may define the `Traversable` typeclass via `sequence` instead of via `traverse`. It is easier to study the properties and laws of the `sequence` operation because it has a simpler type signature. It is also easier to verify the laws of `sequence`. So, we will use `sequence` when proving the type constructions for traversable functors.

12.3.5 Laws of `traverse`

Given a value `p: L[A]`, where `L` is a traversable functor, programmers expect that the code `p.traverse(f)` should evaluate the function `f: A => F[B]` on each value of type `A` stored within the container `p`. The resulting applicative F-effects should be merged into a single value of type `F[L[B]]`. At the same time, the structure of the initial value `p: L[A]` must be preserved as much as possible in the resulting values of type `L[B]` wrapped under `F`. Also, the `traverse` function should work in the same way for all applicative functors `F` and for all types `A` and `B`. How can we express these expectations as laws?[1]

A naturality law formalizes the requirement that a function with a type parameter should work in the same way regardless of the type chosen for that type parameter. There is one naturality law per type parameter. So, we impose the naturality laws with respect to type parameters `A` and `B`. The form of the naturality laws follows from the type signature of `traverse`:

$$\text{trav}_L^{F,A,B} : (A \to F^B) \to L^A \to F^{L^B} \quad .$$

If we fix `B` and let `A` vary, the type signature is of the form $J^A \to K^A$, which is a natural transformation between two contrafunctors. If we fix `A` and let `B` vary, this type signature has the form $G^A \to H^A$, which is a natural transformation between two functors. So, we can formulate the two naturality laws of `traverse` according to the recipes of Section 9.4.1. For any $f^{:X\to A}$, $g^{:A\to F^B}$, and $h^{:B\to C}$:

$$\text{trav}_L^{F,X,B}(f \,⨾\, g) = f^{\uparrow L} \,⨾\, \text{trav}_L^{F,A,B}(g) \quad , \qquad \text{trav}_L^{F,A,C}(g \,⨾\, h^{\uparrow F}) = \text{trav}_L^{F,A,B}(g) \,⨾\, h^{\uparrow L\uparrow F} \quad . \tag{12.16}$$

$$\begin{tikzcd} L^X \arrow[d, "f^{\uparrow L}"] \arrow[dr, "\text{trav}_L^{F,X,B}(f \,⨾\, g)"] & \\ L^A \arrow[r, "\text{trav}_L^{F,A,B}(g)"'] & F^{L^B} \end{tikzcd} \qquad \begin{tikzcd} L^A \arrow[r, "\text{trav}_L^{F,A,B}(g)"] \arrow[dr, "\text{trav}_L^{F,A,C}(g \,⨾\, h)"'] & F^{L^B} \arrow[d, "h^{\uparrow L\uparrow F}"] \\ & F^{L^C} \end{tikzcd}$$

We also need a naturality law with respect to the parameter `F`, which is a type constructor required to be an applicative functor. This law expresses the requirement that `traverse` may not inspect the type of `F` directly and make decisions based on that type. The code of `traverse` may only use F's applicative methods (`wu` and `zip`). To formulate this **applicative naturality law**, we write an equation similar to the second naturality law, except that the arbitrary function f will now map the applicative functor F to another arbitrary applicative functor G:

$$\begin{tikzcd} L^A \arrow[r, "\text{trav}_L^{F,A,B}(g)"] \arrow[dr, "\text{trav}_L^{G,A,B}(g \,⨾\, f)"'] & F^{L^B} \arrow[d, "f"] \\ & G^{L^B} \end{tikzcd} \qquad \text{trav}_L^{G,A,B}(g^{:A\to F^B} \,⨾\, f^{:F^B\to G^B}) = \text{trav}_L^{F,A,B}(g) \,⨾\, f^{:F^{L^B}\to G^{L^B}} \quad . \tag{12.17}$$

Here f is a natural transformation with type signature $f : \forall X. F^X \to G^X$. So, we are allowed to apply the same code of f to different types: f has type $F^B \to G^B$ in the left-hand side of the law and $F^{L^B} \to G^{L^B}$ in the right-hand side.

In addition to being a natural transformation, the function f must preserve the applicative typeclass. In other words, f should map F's applicative methods (`wu` and `zip`) to the corresponding methods of G. Otherwise, the code of $\text{trav}_L^{F,A,B}$ that uses those methods of F will not be mapped via f to the same code of $\text{trav}_L^{G,A,B}$ that uses G's applicative methods.

Natural transformations $f : F^X \to G^X$ that preserve the applicative methods of F and G are called **applicative morphisms** (compare to monoid morphisms defined in Section 12.3.1 and monad morphisms defined in Section 10.3.4). The laws of applicative morphisms are:

$$\text{identity law of applicative morphism } f : \quad f^{:F^{\mathbb{1}}\to G^{\mathbb{1}}}(\text{wu}_F) = \text{wu}_G \quad , \tag{12.18}$$

$$\text{composition law of applicative morphism } f : \quad \text{zip}_G\big(f(p^{:F^A}) \times f(q^{:F^B})\big) = f(\text{zip}_F(p \times q)) \quad . \tag{12.19}$$

[1] Those laws were given in the paper by M. Jaskelioff and O. Rypacek, "An investigation of the laws of traversals", see https://arxiv.org/abs/1202.2919

The applicative naturality law is required to hold only when f is an applicative morphism.

Here are some examples of applicative morphisms and applicative naturality.

Example 12.3.5.1 Consider applicative functors $\text{Id}^A \stackrel{\text{def}}{=} A$ and $L^A \stackrel{\text{def}}{=} A \times A$.

(a) Show that the standard function $\Delta : \text{Id}^A \to L^A$ is an applicative morphism.

(b) Show that the standard function $\pi_1 : L^A \to \text{Id}^A$ is an applicative morphism.

Solution We need to check that these functions obey the laws of applicative morphisms.

(a) To verify the identity law, write:

$$\Delta(\text{wu}_{\text{Id}}) = \Delta(1) = 1 \times 1 = \text{wu}_P \quad .$$

To verify the composition law, begin with its left-hand side:

$$\begin{aligned} &\text{zip}_L\big(\Delta(p^{:A}) \times \Delta(q^{:A})\big) = \text{zip}_L\big((p \times p) \times (q \times q)\big) \\ \text{definition of zip}_L : \quad &= (p \times q) \times (p \times q) = \Delta(p \times q) = \Delta(\text{zip}_{\text{Id}}(p \times q)) \quad . \end{aligned}$$

(b) To verify the identity law, write:

$$\pi_1(\text{wu}_L) = \pi_1(1 \times 1) = 1 = \text{wu}_{\text{Id}} \quad .$$

To verify the composition law, begin with its right-hand side:

$$\begin{aligned} &\pi_1\big(\text{zip}_L((p_1^{:A} \times p_2^{:A}) \times (q_1^{:B} \times q_2^{:B}))\big) = \pi_1\big((p_1 \times q_1) \times (p_2 \times q_2)\big) = p_1 \times q_1 \\ &= \text{zip}_{\text{Id}}(p_1 \times q_1) = \text{zip}_{\text{Id}}(\pi_1(p_1 \times p_2) \times \pi_1(q_1 \times q_2)) \quad . \end{aligned}$$

Example 12.3.5.2 Verify the naturality law of `traverse` for $L^A \stackrel{\text{def}}{=} A \times A$ using an applicative morphism between applicative functors $G^A \stackrel{\text{def}}{=} \mathbb{1} + A$ and $H^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$.

Solution The `traverse` method for $L^A \stackrel{\text{def}}{=} A \times A$ is defined by:

$$\text{trav}_L(f^{:A \to F^B}) \stackrel{\text{def}}{=} p^{:A} \times q^{:A} \to \text{zip}_F(f(p) \times f(q)) = (f \boxtimes f) ⨾ \text{zip}_F \quad .$$

The `zip` methods of the applicative functors G and H are:

$$\text{zip}_G \stackrel{\text{def}}{=} \left| \begin{array}{c|cc} & \mathbb{1} & A \times B \\ \hline \mathbb{1} \times \mathbb{1} & _ \to 1 & \mathbb{0} \\ A \times \mathbb{1} & _ \to 1 & \mathbb{0} \\ \mathbb{1} \times B & _ \to 1 & \mathbb{0} \\ A \times B & \mathbb{0} & \text{id} \end{array} \right| \quad , \quad \text{zip}_H \stackrel{\text{def}}{=} \left| \begin{array}{c|cc} & \mathbb{1} & (A \times B) \times (A \times B) \\ \hline \mathbb{1} \times \mathbb{1} & _ \to 1 & \mathbb{0} \\ (A \times A) \times \mathbb{1} & _ \to 1 & \mathbb{0} \\ \mathbb{1} \times (B \times B) & _ \to 1 & \mathbb{0} \\ (A \times A) \times (B \times B) & \mathbb{0} & \text{zip}_L \end{array} \right| \quad .$$

The function zip_L was shown in the solution to Example 12.3.5.1(a).

The applicative morphism $g : G^A \to H^A$ is implemented as:

$$g : \mathbb{1} + A \to \mathbb{1} + A \times A \quad , \quad g \stackrel{\text{def}}{=} \left| \begin{array}{c|cc} & \mathbb{1} & A \times A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \Delta \end{array} \right| \quad .$$

The fully parametric implementations of trav_L, zip_G, zip_H, and g are forced by their type signatures. The values wu_G and wu_H need to be chosen as $\text{wu}_G \stackrel{\text{def}}{=} \mathbb{0}^{:\mathbb{1}} + 1$ and $\text{wu}_H \stackrel{\text{def}}{=} \mathbb{0}^{:\mathbb{1}} + 1 \times 1$ in order to obey the identity laws of applicative morphisms.

Next, we verify that g is an applicative morphism. To check the identity law of g:

$$\text{wu}_G ▷ g = \left| \begin{array}{cc} \mathbb{0} & 1 \end{array} \right| ▷ \left| \begin{array}{c|cc} & \mathbb{1} & A \times A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \Delta \end{array} \right| = \left| \begin{array}{cc} \mathbb{0} & (1 ▷ \Delta) \end{array} \right| = \mathbb{0} + 1 \times 1 = \text{wu}_H \quad .$$

To check the composition law of g, first rewrite that law in the point-free style:

$$(g \boxtimes g) ⨾ \text{zip}_H = \text{zip}_G ⨾ g \quad . \tag{12.20}$$

The pair product $(g \boxtimes g)$ can be written in matrix form like this:

$$g \boxtimes g = \begin{array}{c||cc|} & \mathbb{1} & A \times A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & \Delta \end{array} \boxtimes \begin{array}{c||cc|} & \mathbb{1} & B \times B \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ B & \mathbb{0} & \Delta \end{array} = \begin{array}{c||cccc|} & \mathbb{1} & A \times A \times \mathbb{1} & \mathbb{1} \times B \times B & A \times A \times B \times B \\ \hline \mathbb{1} \times \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} & \mathbb{0} \\ A \times \mathbb{1} & \mathbb{0} & \Delta \boxtimes \text{id} & \mathbb{0} & \mathbb{0} \\ \mathbb{1} \times B & \mathbb{0} & \mathbb{0} & \text{id} \boxtimes \Delta & \mathbb{0} \\ A \times B & \mathbb{0} & \mathbb{0} & \mathbb{0} & \Delta \boxtimes \Delta \end{array} \quad .$$

The two sides of the composition law (12.20) are then simplified to:

$$(g \boxtimes g) ⨾ \text{zip}_H = \begin{array}{||cccc|} \text{id} & \mathbb{0} & \mathbb{0} & \mathbb{0} \\ \mathbb{0} & \Delta \boxtimes \text{id} & \mathbb{0} & \mathbb{0} \\ \mathbb{0} & \mathbb{0} & \text{id} \boxtimes \Delta & \mathbb{0} \\ \mathbb{0} & \mathbb{0} & \mathbb{0} & \Delta \boxtimes \Delta \end{array} ⨾ \begin{array}{||cc|} _ \to 1 & \mathbb{0} \\ _ \to 1 & \mathbb{0} \\ _ \to 1 & \mathbb{0} \\ \mathbb{0} & \text{zip}_L \end{array} = \begin{array}{||cc|} _ \to 1 & \mathbb{0} \\ _ \to 1 & \mathbb{0} \\ _ \to 1 & \mathbb{0} \\ \mathbb{0} & (\Delta \boxtimes \Delta) ⨾ \text{zip}_L \end{array} \quad ,$$

$$\text{zip}_G ⨾ g = \begin{array}{c||cc|} & \mathbb{1} & A \times B \\ \hline \mathbb{1} \times \mathbb{1} & _ \to 1 & \mathbb{0} \\ A \times \mathbb{1} & _ \to 1 & \mathbb{0} \\ \mathbb{1} \times B & _ \to 1 & \mathbb{0} \\ A \times B & \mathbb{0} & \text{id} \end{array} ⨾ \begin{array}{c||cc|} & \mathbb{1} & (A \times B) \times (A \times B) \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A \times B & \mathbb{0} & \Delta \end{array} = \begin{array}{||cc|} _ \to 1 & \mathbb{0} \\ _ \to 1 & \mathbb{0} \\ _ \to 1 & \mathbb{0} \\ \mathbb{0} & \Delta \end{array} \quad .$$

The remaining difference is:

$$(\Delta \boxtimes \Delta) ⨾ \text{zip}_L \stackrel{?}{=} \Delta \quad .$$

This equation can be verified by applying both sides to some $a^{:A} \times b^{:B}$:

$$\begin{aligned} &(a \times b) ▷ (\Delta \boxtimes \Delta) ⨾ \text{zip}_L = \big((a \times a) \times (b \times b)\big) ▷ \text{zip}_L = (a \times b) \times (a \times b) \quad , \\ &(a \times b) ▷ \Delta = (a \times b) \times (a \times b) \quad . \end{aligned}$$

We are now ready to check that the `traverse` method of L satisfies the naturality law (12.17), where we swap f with g and use applicative functors G, H instead of F, G. Write the two sides of the law:

$$\begin{aligned} \text{left-hand side :} \quad & \text{trav}_L^{H,A,B}(f^{:A \to G^B} ⨾ g^{:G^B \to H^B}) = \big((f ⨾ g) \boxtimes (f ⨾ g)\big) ⨾ \text{zip}_H \quad , \\ \text{right-hand side :} \quad & \text{trav}_L^{G,A,B}(f^{:A \to G^B}) ⨾ g^{:G^{L^B} \to H^{L^B}} = (f \boxtimes f) ⨾ \underline{\text{zip}_G ⨾ g} \\ \text{composition law (12.20) :} \quad & = \underline{(f \boxtimes f) ⨾ (g \boxtimes g)} ⨾ \text{zip}_H \\ \text{composition law (7.2) :} \quad & = \underline{\big((f ⨾ g) \boxtimes (f ⨾ g)\big)} ⨾ \text{zip}_H \quad . \end{aligned}$$

The two sides of the law are now equal. □

In this derivation, we were able to pass from zip_G to zip_H only because g, being an applicative morphism, maps G's `zip` operation into H's `zip` operation according to Eq. (12.19).

Example 12.3.5.3 For any applicative functor F, show that the `pure` method ($\text{pu}_F : A \to F^A$) is an applicative morphism between the identity functor ($\text{Id}^A \stackrel{\text{def}}{=} A$) and F.

Solution The identity functor's applicative methods are identity functions:

$$\text{wu}_{\text{Id}} = 1 \quad , \qquad \text{pu}_{\text{Id}}(x) = x \quad , \qquad \text{zip}_{\text{Id}}(p \times q) = p \times q \quad .$$

We need to show that pu_F obeys the laws (12.18)–(12.19). To verify the identity law:

$$\text{pu}_F(\text{wu}_{\text{Id}}) \stackrel{?}{=} \text{wu}_F = \text{pu}_F(1) = \text{pu}_F(\text{wu}_{\text{Id}}) \quad .$$

To verify the composition law:

$$\begin{aligned} & \text{zip}_F(\text{pu}_F(p^{:A}) \times \text{pu}_F(q^{:B})) \stackrel{?}{=} \text{pu}_F(\text{zip}_{\text{Id}}(p \times q)) = \text{pu}_F(p \times q) \\ \text{use Exercise 11.5.6.5 :} \quad & = \text{zip}_F(\text{pu}_F(p) \times \text{pu}_F(q)) \quad . \end{aligned}$$

□

We have analyzed the expectation that `traverse` should work in the same way for all types and applicative functors. Another reasonable expectation is that `p.traverse(f)` should visit *every* value of type `A` stored inside `p` (where `p` has type L^A). To formulate that expectation as a law, consider for simplicity a function f of type $A \to F^A$. Then the final result of `p.traverse(f)` has type F^{L^A}, which can be visualized as a value of type L^A wrapped under an F-effect. We expect that the data structure of `p` should be preserved in the final result of `p.traverse(f)`.

To formulate this condition in a simple way, we may choose functions f that always return an *empty* F-effect (that is, a result of applying F's `pure` method to some value). So, we choose $f \stackrel{\text{def}}{=} \text{pu}_F$. We expect that traversing `p` with this function f should reproduce the same value `p` wrapped in an empty F-effect. So, we write the law as `p.traverse(pure) == pure(p)`, or in the point-free style:

$$\text{trav}_L^{F,A,A}(\text{pu}_F) = \text{pu}_F \quad . \tag{12.21}$$

Alternatively, we may choose $F^A \stackrel{\text{def}}{=} \text{Id}^A = A$ (the identity functor), $A = B$, and $f^{:A \to F^B} = \text{id}^{:A \to A}$. Then the result of `p.traverse(f)` will have type $F^{L^A} = L^A$ and is expected to equal the initial value `p`, as the transformation `f` is an identity function. We obtain the **identity law** of `traverse`:

$$\text{trav}_L^{\text{Id},A,A}(\text{id}) = \text{id} \quad . \tag{12.22}$$

We can show that this law is equivalent to the more complicated law (12.21):

Statement 12.3.5.4 **(a)** If `traverse` obeys Eq. (12.21) then it also obeys Eq. (12.22).

(b) If `traverse` obeys Eq. (12.22) and the naturality law (12.17) then it also obeys Eq. (12.21).

Proof **(a)** We may set $F \stackrel{\text{def}}{=} \text{Id}$ in the law (12.21). Then we get Eq. (12.22) since $\text{pu}_{\text{Id}} = \text{Id}$.

(b) Example 12.3.5.3 shows that pu_F is an applicative morphism between Id and F. So, we may set $F \stackrel{\text{def}}{=} \text{Id}$, $G \stackrel{\text{def}}{=} F$, $f \stackrel{\text{def}}{=} \text{pu}_F$, and $g \stackrel{\text{def}}{=} \text{id}$ in the applicative naturality law (12.17). We then obtain:

$$\begin{aligned} \text{applicative naturality law (12.17) :} \quad & \text{trav}_L(\text{id} \,⨾\, f) \stackrel{!}{=} \underline{\text{trav}_L(\text{id})} \,⨾\, f \\ \text{identity law (12.22) :} \quad & = \text{id} \,⨾\, f = f = \text{pu}_F \quad . \end{aligned}$$

At the same time, we have: $\text{trav}_L(\text{id} \,⨾\, f) = \text{trav}_L(f) = \text{trav}_L(\text{pu}_F)$. So, Eq. (12.21) holds. □

Apart from visiting every stored value exactly once, we expect that all F-effects returned by f are collected by `traverse` exactly once. To see an example where that expectation fails, consider a simple traversable functor $L^A = A \times A$ and write the `traverse` method like this:

```
def trav2[A, B, F[_]: WuZip : Functor](f: A => F[B])(la: (A, A)): F[(B, B)] = {
  val (f1, f2) = (f(la._1), f(la._2))
  (f1 zip f1 zip f2).map { case ((x, y), z) => (y, z) }
}
```

This code applies the function `f` to both values in the given pair. The result is a pair `(f1, f2)` of type `(F[B], F[B])`. The code copies the first F-effect twice and uses `zip` to combine the three F-effects together. The first of the wrapped values of type `B` is discarded.

We expect that the function `trav2` should violate some law of `traverse`. It turns out that a suitable law is found by composing two `traverse` operations. Consider two different applicative functors (F and G), two functions $f : A \to F^B$ and $g : B \to G^C$, and compute the following composition:

$$p \triangleright \text{trav}_L^{F,A,B}(f) \triangleright \left(\text{trav}_L^{G,B,C}(g)\right)^{\uparrow F} : F^{G^{L^C}} \quad .$$

The second `traverse` operation needs to be lifted to `F` for the types to match. The result (of type $F^{G^{L^C}}$) looks like a `traverse` operation with respect to the functor $F \circ G$. By Statement 11.3.6.1, the functor $F \circ G$ is applicative. So, we may apply a single `traverse` operation using that functor and obtain:

$$p \triangleright \text{trav}_L^{F\circ G,A,C}(f \fatsemi g^{\uparrow F}) : F^{G^{L^C}} \quad .$$

$$\begin{tikzcd} L^A \arrow[r, "\text{trav}_L^{F,A,B}(f)"] \arrow[dr, "\text{trav}_L^{F\circ G,A,C}(f \fatsemi g^{\uparrow F})"'] & F^{L^B} \arrow[d, "\left(\text{trav}_L^{G,B,C}(g)\right)^{\uparrow F}"] \\ & F^{G^{L^C}} \end{tikzcd}$$

The **composition law** of `traverse` says that the two results should be equal:

$$\text{trav}_L^{F,A,B}(f) \fatsemi \left(\text{trav}_L^{G,B,C}(g)\right)^{\uparrow F} = \text{trav}_L^{F\circ G,A,C}(f \fatsemi g^{\uparrow F}) \quad . \tag{12.23}$$

So far, we have not given a motivation for the law (12.23). We just wrote that law by analogy with composition laws of other "lifting"-like functions. It is not obvious that the law (12.23) indeed prevents `traverse` from evaluating some effects more than once. To get a heuristic explanation, assume that `traverse` evaluates some effect twice. The left-hand side of the law (12.23) will evaluate some F-effect twice and then evaluate some G-effect twice, wrapped under the twice evaluated F-effect. The right-hand side will evaluate an $(F \circ G)$-effect twice. To illustrate the difference between the resulting effects, we may write heuristically $(F \times F) \circ (G \times G)$ and $(F \circ G) \times (F \circ G)$. In general, these results will not be the same when F-effects and G-effects do not commute.

To make this heuristic explanation more rigorous, let us show an example where the law (12.23) is violated by the function `trav2` shown above. We need to choose sufficiently complicated applicative functors F and G whose effects do not commute. A suitable choice is to set both F and G to the `State` monad.[2] For simplicity, we set all type parameters to `Int` and choose the functions f and g that add their arguments to the internal state. The complete code (using the `Zippable` typeclass defined in Section 11.3.2) is shown in Fig. 12.1.

We have shown that *some* incorrect implementations of `traverse` are excluded by the composition laws. But it remains unknown (Problem 15.3.2.4) whether the composition law (12.23) always guarantees that the traverse function collects each F-effect exactly once.

12.3.6 Laws of `sequence`

We now derive the laws of `sequence` that correspond to the laws of `traverse` found previously.

Begin with the naturality laws. Since the type signature of `sequence` ($L^{F^A} \to F^{L^A}$) has only one type parameter, `sequence` has only one naturality law:

$$\begin{tikzcd} L^{F^A} \arrow[r, "f^{\uparrow F\uparrow L}"'] \arrow[d, "\text{seq}_L^{F,A}"'] & L^{F^B} \arrow[d, "\text{seq}_L^{F,B}"] \\ F^{L^A} \arrow[r, "f^{\uparrow L\uparrow F}"] & F^{L^B} \end{tikzcd}$$

$$(f^{:A\to B})^{\uparrow F\uparrow L} \fatsemi \text{seq}_L^{F,B} = \text{seq}_L^{F,A} \fatsemi f^{\uparrow L\uparrow F} \quad . \tag{12.24}$$

It will be shown in Exercise 12.3.9.6 that this naturality law is equivalent to the two naturality laws of `traverse`.

Turning now to the applicative naturality law (12.17) of `traverse`, we derive the corresponding applicative naturality law of `sequence`. Express `traverse` via `sequence` and substitute into Eq. (12.17):

$$\text{left-hand side}: \quad \text{trav}_L^{G,A,B}(g \fatsemi f^B) = (g \fatsemi f^B)^{\uparrow L} \fatsemi \text{seq}_L^{G,B} = g^{\uparrow L} \fatsemi f^{\uparrow L} \fatsemi \text{seq}_L \quad ,$$

$$\text{right-hand side}: \quad \text{trav}_L^{F,A,B}(g) \fatsemi f^{L^B} = g^{\uparrow L} \fatsemi \text{seq}_L^{F,B} \fatsemi f^{L^B} \quad .$$

[2] The paper https://arxiv.org/abs/1202.2919 shows another example using the `List` monad.

```
import io.chymyst.ch.implement // Automatic implementation of some methods.

type L[A] = (A, A) // A very simple but nontrivial traversable functor.

implicit val functorL: Functor[L] = new Functor[L] {
  override def map[A, B](fa: (A, A))(f: A => B): (B, B) = implement
}

def trav2[A, B, F[_] : Zippable : Functor](f: A => F[B])(la: L[A]): F[L[B]] = la.map(f) match {
  case (f1, f2) => f1 zip f1 zip f2
}

final case class S[A](run: Int => (A, Int)) // State monad with internal state of type Int.

implicit val FunctorS: Functor[S] = new Functor[S] {
  override def map[A, B](fa: S[A])(f: A => B): S[B] = implement
}

implicit val ZippableS: Zippable[S] = new Zippable[S] {
  override def zip[A, B](fa: S[A], fb: S[B]): S[(A, B)] = S { i  =>
    val (a, j) = fa.run(i)
    val (b, k) = fb.run(j)
    ((a, b), k)
  }
}

  // Define F[A] as the composition S[S[A]] and implement WuZip and Functor instances.
final case class F[A](run: S[S[A]]) {
  // A runner method to help with testing.
  def eval(i: Int, j: Int): (A, Int, Int) = {
    val (sa, k) = run.run(i)
    val (a, l) = sa.run(j)
    (a, k, l)
  }
}

implicit val FunctorF: Functor[F] = new Functor[F] {
  override def map[A, B](fa: F[A])(f: A => B): F[B] = implement
}

implicit val ZippableF: Zippable[F] = new Zippable[F] {
  override def zip[A, B](fa: F[A], fb: F[B]): F[(A, B)] =
    F((fa.run zip fb.run).map { case (sa, sb) => sa zip sb })
}

val f: Int => S[Int] = i => S(j => (i + j, i + j))
val g: Int => S[Int] = f
val ff: Int => F[Int] = i => F(f(i).map(g))
val l: L[Int] = (1, 0)

val result1: F[L[Int]] = trav2[Int, Int, F](f1f2)(l)
val result2: F[L[Int]] = {
  val x: S[(Int, Int)] = trav2[Int, Int, S](f1)(l)
  val y: S[S[(Int, Int)]] = x.map(trav2[Int, Int, S](f2))
  F(y)
}

// Show that result1 is not equal to result2:

scala> result1.eval(0, 0)
res0: ((Int, Int), Int, Int) = ((3, 5), 2, 5)

scala> result2.eval(0,0)
res1: ((Int, Int), Int, Int) = ((4, 6), 2, 6)
```

Figure 12.1: A counterexample violating the composition law of `traverse`.

The function $g^{\uparrow L}$ can be omitted from both sides (see Exercise 11.5.6.6 for a similar property).

$$\begin{tikzcd} L^{F^A} \arrow[r, "(f^A)^{\uparrow L}"'] \arrow[d, "\text{seq}_L^{F,A}"'] & L^{G^A} \arrow[d, "\text{seq}_L^{G,A}"] \\ F^{L^A} \arrow[r, "f^{L^A}"] & G^{L^A} \end{tikzcd}$$

Renaming B to A, we rewrite the applicative naturality law of `sequence` as:

$$(f^A)^{\uparrow L} \fatsemi \text{seq}_L^{G,A} = \text{seq}_L^{F,A} \fatsemi f^{L^A} \quad .$$

Here we have written the full type parameters in f^X and $\text{seq}_L^{F,A}$ for clarity.

Turning now to the identity law, we again express `traverse` via `sequence` and substitute into the simpler formulation (12.22) of the identity law of `traverse`:

$$\text{id} \stackrel{!}{=} \text{trav}_L^{\text{Id},A,A}(\text{id}) = \text{id}^{\uparrow L} \fatsemi \text{seq}_L^{\text{Id},A} \stackrel{!}{=} \text{seq}_L^{\text{Id},A} \quad .$$

So, the **identity law** of `sequence` is:

$$\text{seq}_L^{\text{Id},A} = \text{id} \quad . \tag{12.25}$$

The composition law (12.23) of `traverse` leads to the following composition law of `sequence`:

$$\begin{aligned} \text{left-hand side}: \quad & \text{trav}_L^{F,A,B}(f^{:A\to F^B}) \fatsemi \left(\text{trav}_L^{G,B,C}(g^{:B\to G^C})\right)^{\uparrow F} = f^{\uparrow L} \fatsemi \text{seq}_L^{F,B} \fatsemi \left(g^{\uparrow L} \fatsemi \text{seq}_L^{G,C}\right)^{\uparrow F} \\ & = f^{\uparrow L} \fatsemi \underline{\text{seq}_L^{F,B} \fatsemi g^{\uparrow L \uparrow F}} \fatsemi (\text{seq}_L^{G,C})^{\uparrow F} \\ \text{naturality law (12.24)}: \quad & = f^{\uparrow L} \fatsemi g^{\uparrow F \uparrow L} \fatsemi \text{seq}_L^{F,G^C} \fatsemi (\text{seq}_L^{G,C})^{\uparrow F} \quad , \\ \text{right-hand side}: \quad & \text{trav}_L^{F\circ G,A,C}(f \fatsemi g^{\uparrow F}) = (f \fatsemi g^{\uparrow F})^{\uparrow L} \fatsemi \text{seq}_L^{F\circ G,C} \quad . \end{aligned}$$

$$\begin{tikzcd} L^{F^{G^A}} \arrow[r, "\text{seq}_L^{F,G^A}"] \arrow[dr, "\text{seq}_L^{F\circ G,A}"'] & F^{L^{G^A}} \arrow[d, "(\text{seq}_L^{G,A})^{\uparrow F}"] \\ & F^{G^{L^A}} \end{tikzcd}$$

Omitting the common function $(f \fatsemi g^{\uparrow F})^{\uparrow L}$ and renaming C to A, we obtain the **composition law** of `sequence`:

$$\text{seq}_L^{F,G^A} \fatsemi (\text{seq}_L^{G,A})^{\uparrow F} = \text{seq}_L^{F\circ G,A} \quad . \tag{12.26}$$

Since `sequence` has a simpler type signature than `traverse` and its laws have a simpler form, we will use `sequence` for verifying the laws of traversable functors.

12.3.7 All polynomial functors are traversable

To show that all polynomial functors are traversable, we implement a `sequence` operation for the five type constructions that build up polynomial functors. We will verify the laws in every case.

Fixed type If $L^A \stackrel{\text{def}}{=} Z$ where Z is a fixed type, we define `sequence` via F's `pure` method:

$$\text{seq}_L^{F,A} : L^{F^A} \to F^{L^A} \cong Z \to F^Z \quad , \qquad \text{seq}_L^{F,A} \stackrel{\text{def}}{=} \text{pu}_F^{:Z\to F^Z} \quad .$$

The identity law (12.25) holds since $\text{seq}_L^{\text{Id},A} = \text{pu}_{\text{Id}} = \text{id}$. To verify the composition law (12.26):

$$\begin{aligned} \text{left-hand side}: \quad & \text{seq}_L^{F,G^A} \fatsemi (\text{seq}_L^{G,A})^{\uparrow F} = \text{pu}_F \fatsemi (\text{pu}_G)^{\uparrow F} \\ \text{naturality law of } \text{pu}_F: \quad & = \text{pu}_G \fatsemi \text{pu}_F \quad , \\ \text{right-hand side}: \quad & \text{seq}_L^{F\circ G,A} = \text{pu}_{F\circ G} \\ \text{definition of } \text{pu}_{F\circ G}: \quad & = \text{pu}_G \fatsemi \text{pu}_F \quad . \end{aligned}$$

The two sides of the law are now equal.

Type parameter If $L^A \stackrel{\text{def}}{=} A$ (the identity functor), `sequence` is defined as the identity function. Identity functions always satisfy identity and composition laws.

The functor composition $L^A \stackrel{\text{def}}{=} M^{N^A}$ (where M and N are some traversable functors) is another case where the type parameter is used to define L. We assume that seq_M and seq_N are already available and satisfy the laws, and define seq_L as:

$$\text{seq}_L : M^{N^{F^A}} \to F^{M^{N^A}} \quad , \qquad \text{seq}_L \stackrel{\text{def}}{=} (\text{seq}_N^{F,A})^{\uparrow M} \fatsemi \text{seq}_M^{F,N^A} \quad . \tag{12.27}$$

The laws of seq_L will be proved in Exercise 12.3.9.8.

Products Here $L^A \stackrel{\text{def}}{=} M^A \times N^A$, with some traversable functors M and N. We assume that seq_M and seq_N are already available and satisfy the laws, and define seq_L as:

$$\text{seq}_L^{F,A} : M^{F^A} \times N^{F^A} \to F^{M^A \times N^A} \quad , \qquad \text{seq}_L^{F,A} \stackrel{\text{def}}{=} m^{:M^{F^A}} \times n^{:N^{F^A}} \to \text{zip}_F\big((m \triangleright \text{seq}_M^{F,A}) \times (n \triangleright \text{seq}_N^{F,A})\big) \quad .$$

In the point-free style, we can write the same code in a shorter formula:

$$\text{seq}_L^{F,A} \stackrel{\text{def}}{=} (\text{seq}_M^{F,A} \boxtimes \text{seq}_N^{F,A}) \fatsemi \text{zip}_F \quad .$$

To verify the identity law (12.25), use the fact that $\text{zip}_{\text{Id}} = \text{id}$:

$$\begin{aligned} \text{seq}_L^{\text{Id},A} &= (\underline{\text{seq}_M^{\text{Id},A}} \boxtimes \underline{\text{seq}_N^{\text{Id},A}}) \fatsemi \text{zip}_{\text{Id}} \\ \text{identity laws of } \text{seq}_M \text{ and } \text{seq}_N : \quad &= (\text{id} \boxtimes \text{id}) \fatsemi \text{id} = \text{id} \quad . \end{aligned}$$

To verify the composition law (12.26), begin with its left-hand side:

$$\begin{aligned} \text{left-hand side} : \quad & \text{seq}_L^{F,G^A} \fatsemi (\text{seq}_L^{G,A})^{\uparrow F} = (\text{seq}_M^{F,G^A} \boxtimes \text{seq}_N^{F,G^A}) \fatsemi \underline{\text{zip}_F \fatsemi (\text{seq}_M^{G,A} \boxtimes \text{seq}_N^{G,A})^{\uparrow F}} \fatsemi \text{zip}_G^{\uparrow F} \\ \text{naturality law of } \text{zip}_F : \quad & = \underline{(\text{seq}_M^{F,G^A} \boxtimes \text{seq}_N^{F,G^A}) \fatsemi ((\text{seq}_M^{G,A})^{\uparrow F} \boxtimes (\text{seq}_N^{G,A})^{\uparrow F})} \fatsemi \text{zip}_F \fatsemi \text{zip}_G^{\uparrow F} \\ \text{composition law (7.2)} : \quad & = \big((\text{seq}_M^{F,G^A} \fatsemi (\text{seq}_M^{G,A})^{\uparrow F}) \boxtimes (\text{seq}_N^{F,G^A} \fatsemi (\text{seq}_N^{G,A})^{\uparrow F})\big) \fatsemi \text{zip}_F \fatsemi \text{zip}_G^{\uparrow F} \quad . \end{aligned}$$

By assumption, seq_M and seq_N satisfy their composition laws. So, we may rewrite the last line as:

$$(\text{seq}_M^{F \circ G,A} \boxtimes \text{seq}_N^{F \circ G,A}) \fatsemi \text{zip}_F \fatsemi \text{zip}_G^{\uparrow F} \quad .$$

We can now transform the right-hand side of the law to the same expression:

$$\begin{aligned} \text{right-hand side} : \quad & \text{seq}_L^{F \circ G,A} = (\text{seq}_M^{F \circ G,A} \boxtimes \text{seq}_N^{F \circ G,A}) \fatsemi \text{zip}_{F \circ G} \\ \text{definition of } \text{zip}_{F \circ G} : \quad & = (\text{seq}_M^{F \circ G,A} \boxtimes \text{seq}_N^{F \circ G,A}) \fatsemi \text{zip}_F \fatsemi \text{zip}_G^{\uparrow F} \quad . \end{aligned}$$

The given implementation of `sequence` will first collect the F-effects stored in M^{F^A} and then the F-effects stored in N^{F^A}. An alternative implementation could first iterate over the data in N and then in M. That implementation still obeys the laws (see Exercise 12.3.9.4 for a proof in case $M = N = \text{Id}$).

Co-products Here $L^A \stackrel{\text{def}}{=} M^A + N^A$, with some traversable functors M and N. We assume that seq_M and seq_N are already available and satisfy the laws, and define seq_L as:

$$\text{seq}_L : M^{F^A} + N^{F^A} \to F^{M^A + N^A} \quad , \qquad \text{seq}_L \stackrel{\text{def}}{=} \begin{array}{|c||c|} & F^{M^A + N^A} \\ \hline M^{F^A} & \text{seq}_M^{F,A} \fatsemi (m^{:M^A} \to m + \mathbb{0}^{:N^A})^{\uparrow F} \\ N^{F^A} & \text{seq}_N^{F,A} \fatsemi (n^{:N^A} \to \mathbb{0}^{:M^A} + n)^{\uparrow F} \end{array} \quad .$$

To verify the identity law (12.25):

$$\text{seq}_L^{\text{Id},A} = \begin{array}{|c||c|} & M^A + N^A \\ \hline M^{\text{Id}^A} & \text{seq}_M^{\text{Id},A} \fatsemi (m^{:M^A} \to m + \mathbb{0}^{:N^A})^{\uparrow \text{Id}} \\ N^{\text{Id}^A} & \text{seq}_N^{\text{Id},A} \fatsemi (n^{:N^A} \to \mathbb{0}^{:M^A} + n)^{\uparrow \text{Id}} \end{array}$$

$$\text{identity laws of } \text{seq}_M \text{ and } \text{seq}_N : \quad = \begin{array}{|c||c|} & M^A + N^A \\ \hline M^A & m^{:M^A} \to m + \mathbb{0}^{:N^A} \\ N^A & n^{:N^A} \to \mathbb{0}^{:M^A} + n \end{array} = \begin{array}{|c||cc|} & M^A & N^A \\ \hline M^A & m \to m & \mathbb{0} \\ N^A & \mathbb{0} & n \to n \end{array} = \text{id} \quad .$$

To verify the composition law (12.26), write its two sides separately:

$$\text{left-hand side}: \quad \mathrm{seq}_L^{F,G^A} \fatsemi (\mathrm{seq}_L^{G,A})^{\uparrow F} = \left\| \begin{array}{l} \mathrm{seq}_M^{F,G^A} \fatsemi (m^{:M^{G^A}} \to m + \mathbb{0}^{:N^{G^A}})^{\uparrow F} \fatsemi (\mathrm{seq}_L^{G,A})^{\uparrow F} \\ \mathrm{seq}_N^{F,G^A} \fatsemi (n^{:N^{G^A}} \to \mathbb{0}^{:M^{G^A}} + n)^{\uparrow F} \fatsemi (\mathrm{seq}_L^{G,A})^{\uparrow F} \end{array} \right| \quad ,$$

$$\text{right-hand side}: \quad \mathrm{seq}_L^{F\circ G,A} = \left\| \begin{array}{l} \mathrm{seq}_M^{F\circ G,A} \fatsemi (m^{:M^A} \to m + \mathbb{0}^{:N^A})^{\uparrow G\uparrow F} \\ \mathrm{seq}_N^{F\circ G,A} \fatsemi (n^{:N^A} \to \mathbb{0}^{:M^A} + n)^{\uparrow G\uparrow F} \end{array} \right| \quad .$$

The left-hand side contains some function compositions lifted to F. Write them separately:

$$(m^{:M^{G^A}} \to m + \mathbb{0}^{:N^{G^A}}) \fatsemi \mathrm{seq}_L^{G,A} = \left| \begin{array}{c|cc} & M^{G^A} & N^{G^A} \\ \hline M^{G^A} & \mathrm{id} & \mathbb{0} \end{array} \right| \fatsemi \left| \begin{array}{c|c} & G^{M^A+N^A} \\ \hline M^{G^A} & \mathrm{seq}_M^{G,A} \fatsemi (m^{:M^A} \to m + \mathbb{0}^{:N^A})^{\uparrow G} \\ N^{G^A} & \mathrm{seq}_N^{G,A} \fatsemi (n^{:N^A} \to \mathbb{0}^{:M^A} + n)^{\uparrow G} \end{array} \right|$$

$$= \mathrm{seq}_M^{G,A} \fatsemi (m^{:M^A} \to m + \mathbb{0}^{:N^A})^{\uparrow G} \quad ,$$

$$(n^{:N^{G^A}} \to \mathbb{0}^{:M^{G^A}} + n) \fatsemi \mathrm{seq}_L^{G,A} = \left| \begin{array}{c|cc} & M^{G^A} & N^{G^A} \\ \hline N^{G^A} & \mathbb{0} & \mathrm{id} \end{array} \right| \fatsemi \left| \begin{array}{c|c} & G^{M^A+N^A} \\ \hline M^{G^A} & \mathrm{seq}_M^{G,A} \fatsemi (m^{:M^A} \to m + \mathbb{0}^{:N^A})^{\uparrow G} \\ N^{G^A} & \mathrm{seq}_N^{G,A} \fatsemi (n^{:N^A} \to \mathbb{0}^{:M^A} + n)^{\uparrow G} \end{array} \right|$$

$$= \mathrm{seq}_N^{G,A} \fatsemi (n^{:N^A} \to \mathbb{0}^{:M^A} + n)^{\uparrow G} \quad .$$

Then simplify the left-hand side using the composition laws of seq_M and seq_N:

$$\mathrm{seq}_L^{F,G^A} \fatsemi (\mathrm{seq}_L^{G,A})^{\uparrow F} = \left\| \begin{array}{l} \underline{\mathrm{seq}_M^{F,G^A} \fatsemi (\mathrm{seq}_M^{G,A})^{\uparrow F}} \fatsemi (m^{:M^A} \to m + \mathbb{0}^{:N^A})^{\uparrow G\uparrow F} \\ \underline{\mathrm{seq}_N^{F,G^A} \fatsemi (\mathrm{seq}_N^{G,A})^{\uparrow F}} \fatsemi (n^{:N^A} \to \mathbb{0}^{:M^A} + n)^{\uparrow G\uparrow F} \end{array} \right|$$

$$= \left\| \begin{array}{l} \mathrm{seq}_M^{F\circ G,A} \fatsemi (m^{:M^A} \to m + \mathbb{0}^{:N^A})^{\uparrow G\uparrow F} \\ \mathrm{seq}_N^{F\circ G,A} \fatsemi (n^{:N^A} \to \mathbb{0}^{:M^A} + n)^{\uparrow G\uparrow F} \end{array} \right| \quad .$$

The two sides are now equal.

Recursive types Here $L^A \stackrel{\text{def}}{=} S^{A,L^A}$, with a recursion scheme given by a bifunctor $S^{A,R}$. In order to obtain the traversable property of L, we will need to assume that S is traversable with respect to both its type parameters in a special way, which we call "bitraversable". It is not enough if $S^{A,R}$ is traversable with respect to each type parameter separately.

A bifunctor $S^{A,B}$ is called **bitraversable** if it has a `bisequence` method (denoted by $\mathrm{seq2}_S$):

$$\mathrm{seq2}_S^{F,A,B} : S^{F^A,F^B} \to F^{S^{A,B}} \quad ,$$

which is parameterized by an arbitrary applicative functor F and is natural in F, A, and B. The traversable laws of identity and composition must also hold for `bisequence`:

$$\text{identity law of } \mathrm{seq2}_S: \quad \mathrm{seq2}_S^{\mathrm{Id},A,B} = \mathrm{id}^{:S^{A,B}\to S^{A,B}} \quad , \tag{12.28}$$

$$\text{composition law of } \mathrm{seq2}_S: \quad \mathrm{seq2}_S^{F,G^A,G^B} \fatsemi (\mathrm{seq2}_S^{G,A,B})^{\uparrow F} = \mathrm{seq2}_S^{F\circ G,A,B} \quad . \tag{12.29}$$

Section 12.3.8 will show that all polynomial bifunctors are bitraversable. So, we are free to use any recursion scheme $S^{A,R}$ as long as S is a polynomial bifunctor. For now, we assume that a lawful $\mathrm{seq2}_S$ is available and define seq_L as:

$$\mathrm{seq}_L^{F,A} : S^{F^A,L^{F^A}} \to F^{S^{A,L^A}} \quad , \qquad \mathrm{seq}_L^{F,A} \stackrel{\text{def}}{=} (\overline{\mathrm{seq}}_L^{F,A})^{\uparrow S^{F^A,\bullet}} \fatsemi \mathrm{seq2}_S^{F,A,L^A} \quad .$$

This definition uses a recursive call to $\overline{\text{seq}}_L^{F,A}$ lifted with respect to the type parameter R of $S^{A,R}$. The inductive assumption is that the recursively called $\overline{\text{seq}}_L^{F,A}$ already obeys the laws we are proving.

To verify the identity law (12.25), we use the assumed law (12.28):

$$\text{seq}_L^{\text{Id},A} = \left(\overline{\text{seq}}_L^{\text{Id},A}\right)^{\uparrow S^{\text{Id}A,\bullet}} ⨾ \text{seq2}_S^{\text{Id},A,L^A} = \text{id}^{\uparrow S^{A,\bullet}} ⨾ \text{id} = \text{id} \quad .$$

To verify the composition law (12.26), write its two sides separately:

$$\text{seq}_L^{F,G^A} ⨾ \left(\text{seq}_L^{G,A}\right)^{\uparrow F} = \left(\overline{\text{seq}}_L^{F,G^A}\right)^{\uparrow S^{F^{G^A},\bullet}} ⨾ \text{seq2}_S^{F,G^A,L^{G^A}} ⨾ \left(\overline{\text{seq}}_L^{G,A}\right)^{\uparrow S^{G^A,\bullet}\uparrow F} ⨾ \left(\text{seq2}_S^{G,A,L^A}\right)^{\uparrow F} \quad ,$$

$$\text{seq}_L^{F\circ G,A} = \left(\overline{\text{seq}}_L^{F\circ G,A}\right)^{\uparrow S^{F^{G^A},\bullet}} ⨾ \text{seq2}_S^{F\circ G,A,L^A} \quad .$$

We use the naturality law of $\text{seq2}_S^{F,A,B}$ with respect to the parameter B:

$$\text{seq2}_S^{F,A,B} ⨾ \left(f^{:B\to C}\right)^{\uparrow S^{A,\bullet}\uparrow F} = f^{\uparrow F\uparrow S^{F^A,\bullet}} ⨾ \text{seq2}_S^{F,A,C} \quad .$$

Then the left-hand side of the law (12.26) becomes:

$$\begin{aligned}
&\text{seq}_L^{F,G^A} ⨾ \left(\text{seq}_L^{G,A}\right)^{\uparrow F} \\
&= \underline{\left(\overline{\text{seq}}_L^{F,G^A}\right)^{\uparrow S^{F^{G^A},\bullet}} ⨾ \left(\overline{\text{seq}}_L^{G,A}\right)^{\uparrow F\uparrow S^{F^{G^A},\bullet}}} ⨾ \text{seq2}_S^{F,G^A,G^{L^A}} ⨾ \left(\text{seq2}_S^{G,A,L^A}\right)^{\uparrow F} \\
\text{composition law (12.26)}: \quad &= \left(\text{seq}_L^{F\circ G,A}\right)^{\uparrow S^{F^{G^A},\bullet}} ⨾ \underline{\text{seq2}_S^{F,G^A,G^{L^A}} ⨾ \left(\text{seq2}_S^{G,A,L^A}\right)^{\uparrow F}} \\
\text{composition law (12.29)}: \quad &= \left(\text{seq}_L^{F\circ G,A}\right)^{\uparrow S^{F^{G^A},\bullet}} ⨾ \underline{\text{seq2}_S^{F\circ G,A,L^A}} \quad .
\end{aligned}$$

The two sides of the law (12.26) are now equal. □

These constructions allow us to implement a `Traversable` instance automatically for any polynomial functor. The corresponding `traverse` function will iterate in a naturally defined order over all values stored in the functor. To visualize the way `traverse` works, write a polynomial functor L as:

$$L^A = Z_0 + Z_1 \times A + Z_2 \times A \times A + \ldots \quad ,$$

where Z_0, Z_1, ..., are fixed types. Any polynomial functor may be equivalently rewritten in this way, perhaps with an infinite number of parts in the disjunction. Each part of the disjunction is written as $Z_k \times A \times \ldots \times A$ and contains a finite number k of values of type A and a value of a fixed type Z_k. The computation `traverse(f)`, where f is a function of type $A \to F^B$, iterates over the k values of type A and merges the F-effects in the natural order using F's `zip` operation. This creates a value of type $F^{B\times\ldots\times B}$. The constant value of type Z_k is then lifted into F to get the final value of type $F^{Z_k\times B\times B\times\ldots\times B}$.

If the traversal order needs to be chosen differently, we will need to provide a custom implementation of a `Traversable` instance.

Non-polynomial functors are, in general, not traversable. As an example, consider the functors $L^A \stackrel{\text{def}}{=} E \to A$ and $F^A \stackrel{\text{def}}{=} Z + A$, where the types E and Z are arbitrary but fixed. It is impossible to implement a fully parametric function with the type signature of `sequence` $(L^{F^B} \to F^{L^B})$, which is $(E \to Z + A) \to Z + (E \to A)$. To prove that any traversable functor *must* be polynomial, one needs advanced methods beyond the scope of this book.[3]

12.3.8 All polynomial bifunctors are bitraversable

Bitraversable bifunctors are used in the recursive-type construction for traversable functors. The properties of bitraversable bifunctors are similar to the properties of traversable functors. We will now prove that all polynomial bifunctors are bitraversable. The proof verifies that each of the five type constructions of polynomial bifunctors will produce a bitraversable bifunctor satisfying the laws (12.28)–(12.29).

[3] See the paper by R. Bird et al.: http://www.cs.ox.ac.uk/jeremy.gibbons/publications/uitbaf.pdf

Fixed type If $S^{A,B} \stackrel{\text{def}}{=} Z$ where Z is a fixed type, we define `bisequence` via F's `pure` method:

$$\mathrm{seq2}_S^{F,A,B} : S^{F^A,F^B} \to F^{S^{A,B}} \cong Z \to F^Z \quad , \qquad \mathrm{seq2}_S^{F,A,B} \stackrel{\text{def}}{=} \mathrm{pu}_F^{:Z\to F^Z} \quad .$$

The identity law (12.28) holds since $\mathrm{seq2}_S^{\mathrm{Id},A,B} = \mathrm{pu}_{\mathrm{Id}} = \mathrm{id}$. To verify the composition law (12.29):

$$\begin{aligned}
\text{left-hand side}: \quad & \mathrm{seq2}_S^{F,G^A,G^B} \fatsemi (\mathrm{seq2}_S^{G,A,B})^{\uparrow F} = \mathrm{pu}_F \fatsemi (\mathrm{pu}_G)^{\uparrow F} \\
\text{naturality law of } \mathrm{pu}_F: \quad & = \mathrm{pu}_G \fatsemi \mathrm{pu}_F \quad , \\
\text{right-hand side}: \quad & \mathrm{seq2}_S^{F\circ G,A,B} = \mathrm{pu}_{F\circ G} \\
\text{definition of } \mathrm{pu}_{F\circ G}: \quad & = \mathrm{pu}_G \fatsemi \mathrm{pu}_F \quad .
\end{aligned}$$

The two sides of the law are now equal.

Type parameter If $S^{A,B} \stackrel{\text{def}}{=} A$ or $S^{A,B} \stackrel{\text{def}}{=} B$, we define `bisequence` as the identity function:

$$\begin{aligned}
\text{if } S^{A,B} \stackrel{\text{def}}{=} A: \quad & \mathrm{seq2}_S^{F,A,B} \stackrel{\text{def}}{=} \mathrm{id}^{:F^A\to F^A} \quad , \\
\text{if } S^{A,B} \stackrel{\text{def}}{=} B: \quad & \mathrm{seq2}_S^{F,A,B} \stackrel{\text{def}}{=} \mathrm{id}^{:F^B\to F^B} \quad .
\end{aligned}$$

Identity functions always satisfy identity and composition laws.

Products Here $S^{A,B} \stackrel{\text{def}}{=} M^{A,B} \times N^{A,B}$, with some bitraversable bifunctors M and N. We assume that $\mathrm{seq2}_M$ and $\mathrm{seq2}_N$ are already available and satisfy the laws, and define $\mathrm{seq2}_S$ as:

$$\mathrm{seq2}_S^{F,A,B} : M^{F^A,F^B} \times N^{F^A,F^B} \to F^{M^{A,B}\times N^{A,B}} \quad , \qquad \mathrm{seq2}_S^{F,A,B} \stackrel{\text{def}}{=} (\mathrm{seq2}_M^{F,A,B} \boxtimes \mathrm{seq2}_N^{F,A,B}) \fatsemi \mathrm{zip}_F \quad .$$

To verify the identity law (12.28), use the fact that $\mathrm{zip}_{\mathrm{Id}} = \mathrm{id}$:

$$\begin{aligned}
& \mathrm{seq2}_S^{\mathrm{Id},A,B} = (\underline{\mathrm{seq2}_M^{\mathrm{Id},A,B}} \boxtimes \underline{\mathrm{seq2}_N^{\mathrm{Id},A,B}}) \fatsemi \mathrm{zip}_{\mathrm{Id}} \\
\text{identity laws of } \mathrm{seq2}_M \text{ and } \mathrm{seq2}_N: \quad & = (\mathrm{id} \boxtimes \mathrm{id}) = \mathrm{id} \quad .
\end{aligned}$$

To verify the composition law (12.29), begin with its left-hand side:

$$\begin{aligned}
\text{left-hand side}: \quad & \mathrm{seq2}_S^{F,G^A,G^B} \fatsemi (\mathrm{seq2}_S^{G,A,B})^{\uparrow F} \\
& = (\mathrm{seq2}_M^{F,G^A,G^B} \boxtimes \mathrm{seq2}_N^{F,G^A,G^B}) \fatsemi \underline{\mathrm{zip}_F \fatsemi (\mathrm{seq2}_M^{G,A,B} \boxtimes \mathrm{seq2}_N^{G,A,B})^{\uparrow F}} \fatsemi \mathrm{zip}_G^{\uparrow F} \\
\text{naturality of } \mathrm{zip}_F: \quad & = \underline{(\mathrm{seq2}_M^{F,G^A,G^B} \boxtimes \mathrm{seq2}_N^{F,G^A,G^B}) \fatsemi ((\mathrm{seq2}_M^{G,A,B})^{\uparrow F} \boxtimes (\mathrm{seq2}_N^{G,A,B})^{\uparrow F})} \fatsemi \mathrm{zip}_F \fatsemi \mathrm{zip}_G^{\uparrow F} \\
\text{the law (7.2)}: \quad & = ((\mathrm{seq2}_M^{F,G^A,G^B} \fatsemi (\mathrm{seq2}_M^{G,A,B})^{\uparrow F}) \boxtimes (\mathrm{seq2}_N^{F,G^A,G^B} \fatsemi (\mathrm{seq2}_N^{G,A,B})^{\uparrow F})) \fatsemi \mathrm{zip}_F \fatsemi \mathrm{zip}_G^{\uparrow F} \quad .
\end{aligned}$$

By assumption, $\mathrm{seq2}_M$ and $\mathrm{seq2}_N$ satisfy their composition laws. So, we may rewrite the last line as:

$$\mathrm{seq2}_S^{F,G^A,G^B} \fatsemi (\mathrm{seq2}_S^{G,A,B})^{\uparrow F} = (\mathrm{seq2}_M^{F\circ G,A,B} \boxtimes \mathrm{seq2}_N^{F\circ G,A,B}) \fatsemi \mathrm{zip}_F \fatsemi \mathrm{zip}_G^{\uparrow F} \quad .$$

We can now transform the right-hand side of the law to the same expression:

$$\begin{aligned}
\text{right-hand side}: \quad & \mathrm{seq2}_S^{F\circ G,A,B} = (\mathrm{seq2}_M^{F\circ G,A,B} \boxtimes \mathrm{seq2}_N^{F\circ G,A,B}) \fatsemi \mathrm{zip}_{F\circ G} \\
\text{definition of } \mathrm{zip}_{F\circ G}: \quad & = (\mathrm{seq2}_M^{F\circ G,A,B} \boxtimes \mathrm{seq2}_N^{F\circ G,A,B}) \fatsemi \mathrm{zip}_F \fatsemi \mathrm{zip}_G^{\uparrow F} \quad .
\end{aligned}$$

Co-products Here $S^{A,B} \stackrel{\text{def}}{=} M^{A,B} + N^{A,B}$, with some bitraversable bifunctors M and N. We assume that $\mathrm{seq2}_M$ and $\mathrm{seq2}_N$ are already available and satisfy the laws, and define $\mathrm{seq2}_S$ as:

$$\mathrm{seq2}_S : M^{F^A,F^B} + N^{F^A,F^B} \to F^{M^{A,B}+N^{A,B}} \quad , \qquad \mathrm{seq2}_S \stackrel{\text{def}}{=} \begin{array}{|c||c|} & F^{M^{A,B}+N^{A,B}} \\ \hline M^{F^A,F^B} & \mathrm{seq2}_M^{F,A,B} \fatsemi (m^{:M^{A,B}} \to m + \mathbb{0})^{\uparrow F} \\ N^{F^A,F^B} & \mathrm{seq2}_N^{F,A,B} \fatsemi (n^{:N^{A,B}} \to \mathbb{0} + n)^{\uparrow F} \end{array} \quad .$$

To verify the identity law (12.28):

$$\text{seq2}_S^{\text{Id},A} = \left| \begin{array}{c|c} & M^{A,B} + N^{A,B} \\ \hline M^{\text{Id}^A,\text{Id}^B} & \text{seq2}_M^{\text{Id},A,B} \,;\, (m \to m + \mathbb{0})^{\uparrow \text{Id}} \\ N^{\text{Id}^A} & \text{seq2}_N^{\text{Id},A,B} \,;\, (n \to \mathbb{0} + n)^{\uparrow \text{Id}} \end{array} \right|$$

identity laws of seq2_M and seq2_N :

$$= \left| \begin{array}{c|c} & M^{A,B} + N^{A,B} \\ \hline M^{A,B} & m \to m + \mathbb{0} \\ N^{A,B} & n \to \mathbb{0} + n \end{array} \right| = \left| \begin{array}{c|cc} & M^{A,B} & N^{A,B} \\ \hline M^{A,B} & m \to m & \mathbb{0} \\ N^{A,B} & \mathbb{0} & n \to n \end{array} \right| = \text{id} \quad .$$

To verify the composition law (12.29), write its two sides separately:

left-hand side : $\text{seq2}_S^{F,G^A,G^B} \,;\, (\text{seq2}_S^{G,A,B})^{\uparrow F} = \left| \begin{array}{c} \text{seq2}_M^{F,G^A,G^B} \,;\, (m \to m + \mathbb{0})^{\uparrow F} \,;\, (\text{seq2}_S^{G,A,B})^{\uparrow F} \\ \text{seq2}_N^{F,G^A,G^B} \,;\, (n \to \mathbb{0} + n)^{\uparrow F} \,;\, (\text{seq2}_S^{G,A,B})^{\uparrow F} \end{array} \right| \quad ,$

right-hand side : $\text{seq2}_S^{F \circ G,A,B} = \left| \begin{array}{c} \text{seq2}_M^{F \circ G,A,B} \,;\, (m \to m + \mathbb{0})^{\uparrow G \uparrow F} \\ \text{seq2}_N^{F \circ G,A,B} \,;\, (n \to \mathbb{0} + n)^{\uparrow G \uparrow F} \end{array} \right| \quad .$

The left-hand side contains some function compositions lifted to F. Write them separately:

$$(m \to m + \mathbb{0}) \,;\, \text{seq2}_S^{G,A,B} = \left| \begin{array}{c|cc} & M^{G^A,G^B} & N^{G^A,G^B} \\ \hline M^{G^A,G^B} & \text{id} & \mathbb{0} \end{array} \right| \,;\, \left| \begin{array}{c|c} & G^{M^{A,B}+N^{A,B}} \\ \hline M^{G^A,G^B} & \text{seq2}_M^{G,A,B} \,;\, (m \to m + \mathbb{0})^{\uparrow G} \\ N^{G^A,G^B} & \text{seq2}_N^{G,A,B} \,;\, (n \to \mathbb{0} + n)^{\uparrow G} \end{array} \right|$$

$$= \text{seq2}_M^{G,A,B} \,;\, (m^{:M^{A,B}} \to m + \mathbb{0}^{:N^{A,B}})^{\uparrow G} \quad ,$$

$$(n \to \mathbb{0} + n) \,;\, \text{seq2}_S^{G,A,B} = \left| \begin{array}{c|cc} & M^{G^A,G^B} & N^{G^A,G^B} \\ \hline N^{G^A,G^B} & \mathbb{0} & \text{id} \end{array} \right| \,;\, \left| \begin{array}{c|c} & G^{M^{A,B}+N^{A,B}} \\ \hline M^{G^A,G^B} & \text{seq2}_M^{G,A,B} \,;\, (m \to m + \mathbb{0})^{\uparrow G} \\ N^{G^A,G^B} & \text{seq2}_N^{G,A,B} \,;\, (n \to \mathbb{0} + n)^{\uparrow G} \end{array} \right|$$

$$= \text{seq2}_N^{G,A,B} \,;\, (n^{:N^{A,B}} \to \mathbb{0}^{:M^{A,B}} + n)^{\uparrow G} \quad .$$

We then simplify the left-hand side using the composition laws of seq2_M and seq2_N:

$$\text{seq2}_S^{F,G^A,G^B} \,;\, (\text{seq2}_S^{G,A,B})^{\uparrow F} = \left\| \begin{array}{c} \underline{\text{seq2}_M^{F,G^A,G^B} \,;\, (\text{seq2}_M^{G,A,B})^{\uparrow F}} \,;\, (m \to m + \mathbb{0})^{\uparrow G \uparrow F} \\ \underline{\text{seq2}_N^{F,G^A,G^B} \,;\, (\text{seq2}_N^{G,A,B})^{\uparrow F}} \,;\, (n \to \mathbb{0} + n)^{\uparrow G \uparrow F} \end{array} \right|$$

$$= \left\| \begin{array}{c} \text{seq2}_M^{F \circ G,A,B} \,;\, (m \to m + \mathbb{0})^{\uparrow G \uparrow F} \\ \text{seq2}_N^{F \circ G,A,B} \,;\, (n \to \mathbb{0} + n)^{\uparrow G \uparrow F} \end{array} \right| \quad .$$

The two sides are now equal.

Recursive types Here $S^{A,B} \stackrel{\text{def}}{=} T^{A,B,S^{A,B}}$, with a recursion scheme given by a **3-functor** $T^{A,B,R}$ (a type constructor covariant in each of its three type parameters). To show that S is bitraversable, we need to assume that T is traversable with respect to all its type parameters at once, which we call "3-traversable". We say that a 3-functor $T^{A,B,R}$ is **3-traversable** if it has a `sequence3` method:

$$\text{seq3}_T^{F,A,B,C} : T^{F^A,F^B,F^C} \to F^{T^{A,B,C}} \quad ,$$

which is parameterized by an arbitrary applicative functor F and is natural in F, A, B, and C. The traversable laws of identity and composition must also hold for `sequence3`:

$$\text{identity law of seq3}_T: \quad \text{seq3}_T^{\text{Id},A,B,C} = \text{id}^{:T^{A,B,C}\to T^{A,B,C}} \quad , \tag{12.30}$$

$$\text{composition law of seq3}_T: \quad \text{seq3}_T^{F,G^A,G^B,G^C} \fatsemi (\text{seq3}_T^{G,A,B,C})^{\uparrow F} = \text{seq3}_T^{F\circ G,A,B,C} \quad . \tag{12.31}$$

To prove that all polynomial 3-functors are 3-traversable, we would need to repeat the proofs in this section with more type parameters. The recursive type construction for 3-traversable 3-functors will require a 4-traversable 4-functor as a recursion scheme. To prove that all polynomial 4-functors are 4-traversable, we will need to use 5-traversable 5-functors, and so on. Note that the proofs in this section differ from the proofs in Section 12.3.7 only in having more type parameters in the functions. So, the proofs for 3-traversable 3-functors, 4-traversable 4-functors, etc., will remain essentially the same as the proofs for bitraversable bifunctors except for having more type parameters in each function. For this reason, we omit those proofs. The conclusion is that all polynomial n-functors are n-traversable for $n = 1, 2, 3, ...$

Assume that a lawful seq3_T is available and define seq2_S as:

$$\text{seq2}_S^{F,A,B} : T^{F^A,F^B,S^{F^A,F^B}} \to F^{T^{A,B,S^{A,B}}} \quad , \qquad \text{seq2}_S^{F,A,B} \stackrel{\text{def}}{=} (\overline{\text{seq2}}_S^{F,A,B})^{\uparrow T^{F^A,F^B,\bullet}} \fatsemi \text{seq3}_T^{F,A,B,S^{A,B}} \quad .$$

The inductive assumption is that the recursively called $\overline{\text{seq2}}_S$ already obeys the laws we are proving.

To verify the identity law (12.28), we use the assumed law (12.30):

$$\text{seq2}_S^{\text{Id},A,B} = (\overline{\text{seq2}}_S^{\text{Id},A,B})^{\uparrow T^{\text{Id}^A,\text{Id}^B,\bullet}} \fatsemi \text{seq3}_T^{\text{Id},A,B,S^{A,B}} = \text{id}^{\uparrow T^{A,B,\bullet}} \fatsemi \text{id} = \text{id} \quad .$$

To verify the composition law (12.29):

$$\begin{aligned}
&\text{seq2}_S^{F,G^A,G^B} \fatsemi (\text{seq2}_S^{G,A,B})^{\uparrow F} \\
&\quad = (\overline{\text{seq2}}_S^{F,G^A,G^B})^{\uparrow T^{F^{G^A},F^{G^B},\bullet}} \fatsemi \text{seq3}_T^{F,G^A,G^B,S^{G^A,G^B}} \fatsemi (\overline{\text{seq2}}_S^{G,A,B})^{\uparrow T^{G^A,G^B,\bullet}\uparrow F} \fatsemi (\text{seq3}_T^{G,A,B,S^{A,B}})^{\uparrow F} \quad , \\
&\text{seq2}_S^{F\circ G,A,B} = (\overline{\text{seq2}}_S^{F\circ G,A,B})^{\uparrow T^{F^{G^A},F^{G^B},\bullet}} \fatsemi \text{seq3}_T^{F\circ G,A,B,S^{A,B}} \quad .
\end{aligned}$$

We use the naturality law of $\text{seq3}_T^{F,A,B,C}$ with respect to the parameter C:

$$\text{seq3}_T^{F,A,B,C} \fatsemi (f^{:C\to D})^{\uparrow T^{A,B,\bullet}\uparrow F} = f^{\uparrow F\uparrow T^{F^A,F^B,\bullet}} \fatsemi \text{seq3}_T^{F,A,B,D} \quad .$$

Then the left-hand side of the law (12.29) becomes:

$$\begin{aligned}
&\text{seq2}_S^{F,G^A,G^B} \fatsemi (\text{seq2}_S^{G,A,B})^{\uparrow F} \\
&= \underline{(\overline{\text{seq2}}_S^{F,G^A,G^B})^{\uparrow T^{F^{G^A},F^{G^B},\bullet}} \fatsemi (\overline{\text{seq2}}_S^{G,A,B})^{\uparrow F\uparrow T^{F^{G^A},F^{G^B},\bullet}}} \fatsemi \text{seq3}_T^{F,G^A,G^B,G^{S^{A,B}}} \fatsemi (\text{seq3}_T^{G,A,B,S^{A,B}})^{\uparrow F} \\
\text{Eq. (12.28)}: \quad &= (\text{seq2}_S^{F\circ G,A,B})^{\uparrow T^{F^{G^A},F^{G^B},\bullet}} \fatsemi \underline{\text{seq3}_T^{F,G^A,G^B,G^{S^{A,B}}} \fatsemi (\text{seq3}_T^{G,A,B,S^{A,B}})^{\uparrow F}} \\
\text{Eq. (12.31)}: \quad &= (\text{seq2}_S^{F\circ G,A,B})^{\uparrow T^{F^{G^A},F^{G^B},\bullet}} \fatsemi \underline{\text{seq3}_T^{F\circ G,A,B,S^{A,B}}} \quad .
\end{aligned}$$

The two sides of the law (12.28) are now equal.

12.3.9 Exercises

Exercise 12.3.9.1 Show that any traversable functor L admits a method called `consume`:

$$\text{consume}_L : (L^A \to B) \to L^{F^A} \to F^B \quad ,$$

defined for any applicative functor F. Assuming a suitable naturality law for `consume`, show that `sequence` and `consume` are equivalent.

Exercise 12.3.9.2 Consider the following implementation of `traverse` for the functor $L^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$:

```
def badtrav[A, B, F[_]: Applicative : Functor](f: A => F[B]): Option[(A, A)] => F[Option[(B, B)]] =
  Applicative[F].pure(None)
```

Show that `badtrav` does *not* satisfy the laws of `traverse`.

Exercise 12.3.9.3 Show that the laws are *not* satisfied by the implementation of $\text{seq} : L^{F^A} \to F^{L^A}$ for $F^A \stackrel{\text{def}}{=} \mathbb{1} + A$ where `seq` always returns an empty option ($1 + \mathbb{0}$, or `None` in Scala).

Exercise 12.3.9.4 For $L^A \stackrel{\text{def}}{=} A \times A$, consider $\text{seq}_L^{F,A}$ which collects the F-effects in the opposite order:

```
def seq[F[_]: Applicative : Functor, A]: ((F[A], F[A])) => F[(A, A)] = {
  case (fa1, fa2) => (fa2 zip fa1).map(_.swap) // Use ‘swap‘ to restore the order of wrapped values.
}
```

In the point-free style, this code is written as:

$$\text{seq}_L^{F,A} : F^A \times F^A \to F^{A \times A} \quad , \qquad \text{seq}_L^{F,A} \stackrel{\text{def}}{=} \text{swap} \,\text{⨾}\, \text{zip}_F \,\text{⨾}\, \text{swap}^{\uparrow F} \quad .$$

Prove that this implementation of `sequence` satisfies its laws.

Exercise 12.3.9.5 Find an example of an applicative functor F and a traversable functor L such that one *cannot* implement a natural transformation with type signature $F^{L^A} \to L^{F^A}$ (the inverse to the type signature of `sequence`).

Exercise 12.3.9.6 Show that the naturality law (12.24) of `sequence` is equivalent to the two naturality laws (12.16) of `traverse`.

Exercise 12.3.9.7 Statement 10.3.2.1 shows that functions `tr` and `ftr` are equivalent if a naturality law holds for `ftr` with respect to the type parameter A. Under the same assumption, show that the naturality law of `ftr` with respect to the type parameter B is equivalent to the naturality law of `tr`.

Exercise 12.3.9.8 Prove that $L^A \stackrel{\text{def}}{=} M^{N^A}$ is a lawful traversable if $M^\bullet$ and $N^\bullet$ are traversable functors.

Exercise 12.3.9.9 Prove directly that all the bitraversable laws hold for the bifunctor $S^{A,B} \stackrel{\text{def}}{=} A \times B$.

Exercise 12.3.9.10 For the tree-like type defined as $T^A \stackrel{\text{def}}{=} \mathbb{1} + A \times T^A \times T^A$, define a `Traversable` instance. Verify that the laws hold by using a suitable recursion scheme $S^{A,R}$.

Exercise 12.3.9.11 Is the recursive type constructor $L^A \stackrel{\text{def}}{=} A + L^{\text{List}^A}$ traversable? Explain via examples what sort of data container it is.

Exercise 12.3.9.12 Prove that for any two monoid morphisms $\phi : M \to N$ and $\psi : N \to P$ (where M, N, and P are monoids), the composition $\phi \,\text{⨾}\, \psi : M \to P$ is again a lawful monoid morphism.

Exercise 12.3.9.13 Given a *monad* $M^\bullet$ and a monoid morphism $\phi : R \to S$ between some monoid types R and S, prove that $\phi^{\uparrow M} : M^R \to M^S$ is also a monoid morphism. (The types M^R and M^S are monoids due to Exercise 10.2.9.13).

Exercise 12.3.9.14 Given a monoid type R and a *monad* morphism $\phi : M \rightsquigarrow N$ between some monads M and N, prove that $\phi : M^R \to N^R$ is a monoid morphism between M^R and N^R. (The types M^R and N^R are monoids due to Exercise 10.2.9.13).

Exercise 12.3.9.15 Each monad is at the same time an applicative functor. Given a monad morphism $\phi : M \rightsquigarrow N$ between two monads, show that ϕ is also an applicative morphism between applicative functors M and N.

Exercise 12.3.9.16 Given an applicative functor M and a monoid morphism $\phi : R \to S$ between some monoid types R and S, prove that $\phi^{\uparrow M} : M^R \to M^S$ is also a monoid morphism. (The types M^R and M^S are monoids due to Exercise 11.5.6.9).

Exercise 12.3.9.17 Given a monoid type R and an applicative morphism $\phi : M \rightsquigarrow N$ between some applicative functors M and N, prove that $\phi : M^R \to N^R$ is a monoid morphism between M^R and N^R. (The types M^R and N^R are monoids due to Exercise 11.5.6.9).

Exercise 12.3.9.18 Given a monoid M, define the functions `inMF` and `outMF`:

$$\text{inMF} : M \to M \to M \quad , \qquad \text{inMF}(x^{:M}) \stackrel{\text{def}}{=} y^{:M} \to x \oplus y \quad ,$$
$$\text{outMF} : (M \to M) \to M \quad , \qquad \text{outMF}(p^{:M \to M}) \stackrel{\text{def}}{=} p(e_M) \quad .$$

This definition of `inMF` is similar to that used in the proof of Statement 12.3.1.2(c).

(a) Prove that `inMF` is a monoid morphism between M and the function monoid MF^M consisting of all functions of type $M \to M$. Define the empty element and the binary operation of MF^M appropriately.

(b) Prove that $\text{inMF} ⨾ \text{outMF} = \text{id}$ but $\text{outMF} ⨾ \text{inMF} \neq \text{id}$. (Give an example of a monoid M where the second equation does not hold.)

(c) Prove that `outMF` is *not* a monoid morphism between MF^M and M.

12.4 Discussion and further developments

12.4.1 Traversable contrafunctors or profunctors are not useful

In Chapter 11, we found some uses for applicative contrafunctors and profunctors. We will now briefly investigate whether contrafunctors or profunctors may be traversable. To answer that question, we will try implementing a lawful `sequence` function for contrafunctors and profunctors.

Suppose that L^A is a contrafunctor. A `sequence` function with the type signature:

$$\text{seq}_L^{F,A} : L^{F^A} \to F^{L^A} \quad ,$$

is required for L to be traversable. That function must obey the applicative naturality law, which ensures that $\text{seq}_L^{F,A}$ works in the same way for any applicative functor F. We note that seq_L is covariant in F because $L^\bullet$ is contravariant. Since $\text{pu}_F : \text{Id}^A \to F^A$ is an applicative morphism (Example 12.3.5.3), we may use this morphism to write the applicative naturality law:

$$\text{seq}_L^{F,A} = \text{pu}_F^{\downarrow L} ⨾ \text{seq}_L^{\text{Id},A} ⨾ \text{pu}_F \quad .$$

The identity law gives $\text{seq}_L^{\text{Id},A} = \text{id}$. So, the code of $\text{seq}_L^{F,A}$ must be this:

$$\text{seq}_L^{F,A} = \text{pu}_F^{\downarrow L} ⨾ \text{pu}_F \quad .$$

The implementation of seq_L follows unambiguously from the applicative naturality law.

We can now verify the composition law (12.26):

$$\begin{aligned}
&\text{left-hand side :} \quad \text{seq}_L^{F,G^A} ⨾ (\text{seq}_L^{G,A})^{\uparrow F} = \text{pu}_F^{\downarrow L} ⨾ \underline{\text{pu}_F ⨾ (\text{pu}_G^{\downarrow L} ⨾ \text{pu}_G)^{\uparrow F}} \\
&\text{naturality law of } \text{pu}_F : \quad = \underline{\text{pu}_F^{\downarrow L} ⨾ \text{pu}_G^{\downarrow L}} ⨾ \text{pu}_G ⨾ \text{pu}_F = (\text{pu}_G ⨾ \text{pu}_F)^{\downarrow L} ⨾ \text{pu}_G ⨾ \text{pu}_F \quad , \\
&\text{right-hand side :} \quad \text{seq}_L^{F \circ G,A} = \text{pu}_{F \circ G}^{\downarrow L} ⨾ \text{pu}_{F \circ G} = (\text{pu}_G ⨾ \text{pu}_F)^{\downarrow L} ⨾ \text{pu}_G ⨾ \text{pu}_F \quad .
\end{aligned}$$

So, every contrafunctor's `sequence` method satisfies the laws of traversables.

However, the code of $\text{seq}_L^{F,A}$ always produces values with empty F-effects. Any information about nontrivial F-effects is ignored. For example, if $L^A \stackrel{\text{def}}{=} A \to Z$ (where Z is a fixed type) then:

$$\text{seq}_L^{F,A} : (F^A \to Z) \to F^{A \to Z} \quad , \qquad \text{seq}_L^{F,A}(f^{:F^A \to Z}) = \text{pu}_F(a^{:A} \to f(\text{pu}_F(a))) \quad .$$

The function f will be never applied to nontrivial F-effects. So, the function seq_L will never obtain any information that f would return when applied to nontrivial F-effects. We conclude that contrafunctors are traversable in a way that does not seem to be practically useful.

Turning now to the case where L is a profunctor, consider a simple example:

$$L^A \stackrel{\text{def}}{=} A \to A \quad , \qquad \text{seq}_L : (F^A \to F^A) \to F^{A \to A} \quad .$$

Since the applicative functor F is unknown and the type signature of seq_L does not provide any arguments of type F^X, we cannot produce values of type F other than via pu_F. So, the only implementation of seq_L is:

$$\text{seq}_L \stackrel{\text{def}}{=} p^{:F^A \to F^A} \to \text{pu}_F(\text{id}^{:A \to A}) \quad .$$

This implementation ignores its argument p, losing information and always returning an empty F-effect. As with contrafunctors, we find that profunctors are not traversable in a useful way.

12.4.2 Traversals for nested recursive types

In most of this book, recursive type constructors are defined via equations of the form $L^A \stackrel{\text{def}}{=} S^{A,L^A}$, where S is a recursion scheme. We have seen one example of a recursive type constructor (a perfect-shaped tree, Sections 3.3.5 and 12.2.1.4) that cannot be defined in this way. The reason is that the recursive type equation for a perfect-shaped binary tree PT^A is:

$$\text{PT}^A \stackrel{\text{def}}{=} A + \text{PT}^{A \times A} \quad . \tag{12.32}$$

This type equation is not of the form $\text{PT}^A \stackrel{\text{def}}{=} S^{A,\text{PT}^A}$ because the recursive use of $\text{PT}^\bullet$ contains a non-trivial type expression $(A \times A)$ instead of just A. To express the type equation (12.32) via a recursion scheme, we may introduce an additional functor P and write:

$$\text{PT}^A \stackrel{\text{def}}{=} S^{A,\text{PT}^{P^A}} \quad , \qquad P^A \stackrel{\text{def}}{=} A \times A \quad .$$

Nested recursive types are defined using type equations of this more general form, for example:

$$T^A \stackrel{\text{def}}{=} S^{A,T^{P^A}} \quad , \quad \text{or} \quad T^{A,B} \stackrel{\text{def}}{=} S^{A,B,T^{P^A,Q^B}} \quad , \quad \text{etc.} \tag{12.33}$$

The nested functors P, Q replace the type parameters A, B of $T^{A,B}$ by different type expressions.

One could write a nested recursive type equation of an even more general form, e.g., $T^A \stackrel{\text{def}}{=} S^{A,T^{T^A}}$, where the nested type expressions may use the type T itself more than once. We will not use such type equations in this book. The nested recursive types of the form (12.33) are sufficient in practice.

Let us see how one can implement a `Traversable` instance for type constructors of the form (12.33).

Statement 12.4.2.1 Given a bitraversable bifunctor $S^{\bullet,\bullet}$ and a traversable functor $P^\bullet$, define a nested recursive type constructor $L^\bullet$ by:

$$L^A \stackrel{\text{def}}{=} S^{A,L^{P^A}} \quad .$$

Then L is a lawful traversable functor with the `sequence` function defined as:

$$\text{seq}_L^{F,A} : S^{F^A,L^{P^{F^A}}} \to F^{S^{A,L^{P^A}}} \quad , \qquad \text{seq}_L^{F,A} \stackrel{\text{def}}{=} \left(\text{seq}_P^{F,A}\right)^{\uparrow L \uparrow S^{F^A,\bullet}} \fatsemi \left(\overline{\text{seq}}_L^{F,P^A}\right)^{\uparrow S^{F^A,\bullet}} \fatsemi \text{seq2}_S^{F,A,L^{P^A}} \quad .$$

$$S^{F^A,L^{P^{F^A}}} \xrightarrow{\left(\text{seq}_P^{F,A}\right)^{\uparrow L \uparrow S^{F^A,\bullet}}} S^{F^A,L^{F^{P^A}}} \xrightarrow{\left(\overline{\text{seq}}_L^{F,P^A}\right)^{\uparrow S^{F^A,\bullet}}} S^{F^A,F^{L^{P^A}}} \xrightarrow{\text{seq2}_S^{F,A,L^{P^A}}} F^{S^{A,L^{P^A}}}$$

Proof We assume that the methods seq_P and seq2_S, as well as the recursively called $\overline{\text{seq}}_L$, already satisfy the laws. To verify the identity law (12.25):

$$\text{seq}_L^{\text{Id},A} = \left(\text{seq}_P^{\text{Id},A}\right)^{\uparrow L \uparrow S^{F^A,\bullet}} \fatsemi \left(\overline{\text{seq}}_L^{\text{Id},P^A}\right)^{\uparrow S^{F^A,\bullet}} \fatsemi \text{seq2}_S^{\text{Id},A,L^{P^A}} = \text{id}^{\uparrow L \uparrow S} \fatsemi \text{id}^{\uparrow S} \fatsemi \text{id} = \text{id} \quad .$$

To verify the composition law (12.26), write its two sides, omitting some type parameters for brevity:

$$\begin{aligned}
&\text{seq}_L^{F,G^A} \fatsemi (\text{seq}_L^{G,A})^{\uparrow F} \\
&\quad = \left(\text{seq}_P^{F,G^A}\right)^{\uparrow L\uparrow S} \fatsemi \left(\overline{\text{seq}}_L^{F,P^{G^A}}\right)^{\uparrow S} \fatsemi \text{seq2}_S^{F,G^A,L^{P^{G^A}}} \fatsemi \left((\text{seq}_P^{G,A})^{\uparrow L\uparrow S} \fatsemi \left(\overline{\text{seq}}_L^{G,P^A}\right)^{\uparrow S} \fatsemi \text{seq2}_S^{G,A,L^{P^A}}\right)^{\uparrow F} \quad , \\
&\text{seq}_L^{F\circ G,A} = \left(\text{seq}_P^{F\circ G,A}\right)^{\uparrow L\uparrow S} \fatsemi \left(\overline{\text{seq}}_L^{F\circ G,P^A}\right)^{\uparrow S} \fatsemi \text{seq2}_S^{F\circ G,A,L^{P^A}} \\
&\quad = \left(\text{seq}_P^{F,G^A} \fatsemi (\text{seq}_P^{G,A})^{\uparrow F}\right)^{\uparrow L\uparrow S} \fatsemi \left(\overline{\text{seq}}_L^{F,G^{P^A}} \fatsemi (\overline{\text{seq}}_L^{G,P^A})^{\uparrow F}\right)^{\uparrow S} \fatsemi \text{seq2}_S^{F,G^A,G^{L^{P^A}}} \fatsemi \left(\text{seq2}_S^{G,A,L^{P^A}}\right)^{\uparrow F} \quad .
\end{aligned}$$

To get the last line, we used the assumed composition laws of seq_P, $\overline{\text{seq}}_L$, and seq2_S.

It is now clear that the two sides of the law differ only in the order of function compositions. The remaining difference is:

$$\begin{aligned}
&\left(\overline{\text{seq}}_L^{F,P^{G^A}}\right)^{\uparrow S} \fatsemi \text{seq2}_S^{F,G^A,L^{P^{G^A}}} \fatsemi \left((\text{seq}_P^{G,A})^{\uparrow L} \fatsemi \overline{\text{seq}}_L^{G,P^A}\right)^{\uparrow S\uparrow F} \\
&\stackrel{?}{=} (\text{seq}_P^{G,A})^{\uparrow F\uparrow L\uparrow S} \fatsemi \left(\overline{\text{seq}}_L^{F,G^{P^A}}\right)^{\uparrow S} \fatsemi \left(\overline{\text{seq}}_L^{G,P^A}\right)^{\uparrow F\uparrow S} \fatsemi \text{seq2}_S^{F,G^A,G^{L^{P^A}}} \quad .
\end{aligned}$$

We will show that the two sides are equal if we rewrite the left-hand side so that the various `sequence` methods are composed in the same order as in the right-hand side. This can be done using naturality laws, which allow us to change the order of composition of lifted functions:

$$\begin{aligned}
\text{left-hand side}: \quad & \left(\overline{\text{seq}}_L^{F,P^{G^A}}\right)^{\uparrow S} \fatsemi \underline{\text{seq2}_S^{F,G^A,L^{P^{G^A}}} \fatsemi \left((\text{seq}_P^{G,A})^{\uparrow L} \fatsemi \overline{\text{seq}}_L^{G,P^A}\right)^{\uparrow S\uparrow F}} \\
\text{naturality law of } \text{seq2}_S: \quad & = \underline{\left(\overline{\text{seq}}_L^{F,P^{G^A}}\right)^{\uparrow S} \fatsemi \left((\text{seq}_P^{G,A})^{\uparrow L} \fatsemi \overline{\text{seq}}_L^{G,P^A}\right)^{\uparrow F\uparrow S}} \fatsemi \text{seq2}_S^{F,G^A,G^{L^{P^A}}} \\
\text{composition under } {}^{\uparrow S}: \quad & = \left(\underline{\overline{\text{seq}}_L^{F,P^{G^A}} \fatsemi (\text{seq}_P^{G,A})^{\uparrow L\uparrow F}}\right)^{\uparrow S} \fatsemi \left(\overline{\text{seq}}_L^{G,P^A}\right)^{\uparrow F\uparrow S} \fatsemi \text{seq2}_S^{F,G^A,G^{L^{P^A}}} \quad . \\
\text{naturality law of } \overline{\text{seq}}_L: \quad & = \left(\overline{\text{seq}}_L^{F,P^{G^A}} \fatsemi (\text{seq}_P^{G,A})^{\uparrow L\uparrow F}\right)^{\uparrow S} \fatsemi \left(\overline{\text{seq}}_L^{G,P^A}\right)^{\uparrow F\uparrow S} \fatsemi \text{seq2}_S^{F,G^A,G^{L^{P^A}}}
\end{aligned}$$

It remains to show that:

$$\left(\overline{\text{seq}}_L^{F,P^{G^A}} \fatsemi (\text{seq}_P^{G,A})^{\uparrow L\uparrow F}\right)^{\uparrow S} \stackrel{?}{=} (\text{seq}_P^{G,A})^{\uparrow F\uparrow L\uparrow S} \fatsemi \left(\overline{\text{seq}}_L^{F,G^{P^A}}\right)^{\uparrow S} \quad .$$

But this is just the naturality law of $\overline{\text{seq}}_L$ under the lifting $(\ldots)^{\uparrow S}$. □

As an advanced example of a nested traversable functor, we will derive a type for square-shaped matrices[4] with elements of type A. For motivation, recall how Example 10.1.2.8 encoded square matrices via nested lists of type `List[List[A]]`. However, a value of that type is not guaranteed to represent a matrix of a consistent shape. We would like to define a type `Sq[A]` whose values always contain n nested lists of length n (for $n = 1, 2, \ldots$). We need to disallow inconsistent nested lists such as $[[1,2], [], [3]]$ that do not correspond to the rows of a square matrix.

Begin by considering an example of a 2×2 matrix. The type of such matrices is not `List[List[A]]` but `List2[List2[A]]`, where `List2[A]` is a list of *exactly* 2 elements of type A. We can rewrite the type `List2[A]` equivalently as a pair $A\times A$ and also as a function $\mathbb{2} \to A$ (where $\mathbb{2}$ is the type containing exactly 2 distinct values). This technique allows us to formulate the types of lists of exactly 3, 4, etc., elements as $\text{List}_3^A \stackrel{\text{def}}{=} \mathbb{3} \to A$, $\text{List}_4^A \stackrel{\text{def}}{=} \mathbb{4} \to A$, and so on. The type of an $n\times n$ matrix is then written symbolically as $\mathbb{n} \to \mathbb{n} \to A$ (where $\mathbb{n} = \mathbb{1}, \mathbb{2}$, etc.) or equivalently as $\mathbb{n}\times\mathbb{n} \to A$.

The type `Sq[A]` is equivalent to an infinite disjunction of types representing square matrices of every size (1×1, 2×2, and so on). To define an infinite disjunctive type in a program of finite size, we need to use recursion at type level. In a mathematical sense, this recursion will be induction

[4] This and other advanced examples of nested recursive types are explained in the paper "Manufacturing datatypes" (1999) by R. Hinze, see `https://www.cs.ox.ac.uk/ralf.hinze/publications/WAAAPL99a.ps.gz`

on the size of the matrix. So, let us introduce the size of the matrix as an extra *type parameter* N. It will be convenient to define `SqSize[N, A]` as the type of matrices of size *at least* N. We intend N to be equivalent to one of the types $\mathbb{1}$, $\mathbb{2}$, etc.

The base case ($N = \mathbb{1}$) and the inductive step (from N to $\mathbb{1} + N$) are written as:

$$\text{Sq}^A \stackrel{\text{def}}{=} \text{SqSize}^{\mathbb{1},A} \quad , \qquad \text{SqSize}^{N,A} \stackrel{\text{def}}{=} (N \times N \to A) + \text{SqSize}^{\mathbb{1}+N,A} \quad .$$

To implement this in Scala, we first define `SqSize[N, A]` as a disjunctive type and then define `Sq[A]`:

```
sealed trait SqSize[N, A]
final case class Matrix[N, A](byIndex: ((N, N)) => A) extends SqSize[N, A]
final case class Next[N, A](next: SqSize[Option[N], A])  extends SqSize[N, A]

type Sq[A] = SqSize[Unit, A]
```

As an example, the 2×2 matrix $\begin{vmatrix} 11 & 12 \\ 21 & 22 \end{vmatrix}$ is represented by a value of type `Sq[Int]` as:

```
val matrix2x2: Sq[Int] = Next(Matrix {
  case (None, None)         => 11
  case (None, Some(_))      => 12
  case (Some(_), None)      => 21
  case (Some(_), Some(_))   => 22
})
```

A 3×3 matrix will have the form `Next(Next(Matrix { ... }))`. An $n \times n$ matrix has $(n - 1)$ nested `Next` constructors in front of a `Matrix` constructor. The size of a matrix is encoded by its type in this way.

We now implement a `sequence` function for `Sq`. Since `Sq` is defined by induction using `SqSize[N, A]`, we must first implement `sequence` for `SqSize`:

```
def sequence[A, N, F[_]: Applicative: Functor]: SqSize[N, F[A]] => F[SqSize[N, A]] = {
  case Matrix(byIndex)   => ???    // Base case.
  case Next(next)        => ???    // Inductive step.
}
```

In the base case, we have a function `byIndex` of type $N \times N \to F^A$ that can return $n \times n$ different values of type F^A. We need to combine all those values together by using F's `zip` method. The result will be a value of type $F^{A \times A \times \ldots \times A}$, which we will then need to convert to the type $F^{N \times N \to A}$. The only way of performing these computations is by enumerating all possible values of type N.

Note that the type `Sq` sets the type parameter `N` in `SqSize[N, A]` as `N = Unit`. This forces the type parameters `N` in all of the `Next()` constructors to be `Unit` wrapped in a number of `Option` constructors. All values of a type of this form can be enumerated explicitly. However, the type `SqSize[N, A]` does not ensure that `N` will be a type with a known finite number of values. This problem prevents us from implementing the `sequence` method for our current definition of `Sq`.

A solution is to add a typeclass constraint (with a typeclass called "`Finite`") on the type parameter `N`. A suitable typeclass instance of `Finite[N]` contains a list of all values of type `N`:

```
type Finite[N] = List[N] // A list of all possible values of type N.
```

We can implement functions that create typeclass instances automatically for all the types we will actually use instead of the type parameter `N`, namely, the types `Unit`, `Option[Unit]`, `Option[Option[Unit]]` and so on. Suitable typeclass instances are defined inductively:

```
implicit val finiteUnit: Finite[Unit] = List(())
implicit def finiteOptionN[N: Finite]: Finite[Option[N]] = None +: Finite[N].map(Some(_))
```

Using these definitions, we can now extract all values of type A from a value of type $N \times N \to A$:

```
def access[N: Finite, A](s: SqSize[N, A], i: Int, j: Int): A = s match {
  case Matrix(byIndex) => byIndex((Finite[N].apply(i), Finite[N].apply(j)))
  case Next(next) => access[Option[N], A](next, i, j)
}
```

```
type Finite[N] = List[N] // A list of all possible values of type N.

object Finite { def apply[N: Finite]: Finite[N] = implicitly[Finite[N]] }

sealed abstract class SqSize[N: Finite, A]

final case class Matrix[N: Finite, A](byIndex: ((N, N)) => A) extends SqSize[N, A]

final case class Next[N: Finite, A](next: SqSize[Option[N], A]) extends SqSize[N, A]

type Sq[A] = SqSize[Unit, A]

implicit val finiteUnit: Finite[Unit] = List(())

implicit def finiteOptionN[N: Finite]: Finite[Option[N]] = None +: Finite[N].map(Some(_))

// Access the matrix element at zero-based index (i, j).
def access[N: Finite, A](s: SqSize[N, A], i: Int, j: Int): A = s match {
  case Matrix(byIndex) => byIndex((Finite[N].apply(i), Finite[N].apply(j)))
  case Next(next) => access[Option[N], A](next, i, j)
}

def size[N: Finite, A](s: SqSize[N, A]): Int = s match {
  case Matrix(_) => Finite[N].length
  case Next(next) => size[Option[N], A](next)
}

def toSeqSeq[N: Finite, A](s: SqSize[N, A]): Seq[Seq[A]] = {
  val length = size(s)
  (0 until length).map(i => (0 until length).map(j => access(s, i, j)))
}

// Test: visualize the matrix defined previously by converting it to nested lists.
scala> toSeqSeq(matrix2x2)
res1: List[List[Int]] = List(List(11, 12), List(21, 22))

def sequenceList[F[_]: Applicative : Functor, A](l: List[F[A]]): F[List[A]] = l match {
  case Nil          => Applicative[F].pure(Nil)
  case head :: tail => (head zip sequenceList(tail)).map { case (x, y) => x +: y }
}

def sequence[N: Finite, F[_]: Applicative : Functor, A](sq: SqSize[N, F[A]]): F[SqSize[N, A]] =
  sq match {
    case Matrix(byIndex)   =>
      val allValuesF: List[F[((N, N), A)]] = for {
        i <- Finite[N]
        j <- Finite[N]
      } yield byIndex((i, j)).map(a => ((i, j), a))

      val fList: F[List[((N, N), A)]] = sequenceList(allValuesF)
      fList.map { values =>
        val valuesMap: ((N, N)) => A = values.toMap.apply
        Matrix[N, A](valuesMap)
      }
    case Next(next)        => sequence[Option[N], F, A](next).map(Next(_))
  }
```

Figure 12.2: Implementing `sequence` for square matrices.

```
// Test: use List as an Applicative Functor.
implicit val applicativeList: Applicative[List] = new Applicative[List] {
  override def pure[A](a: A): List[A] = List(a)
  override def zip[A, B](fa: List[A], fb: List[B]): List[(A, B)] = fa zip fb     }

implicit val functorList: Functor[List] = new Functor[List] {
  override def map[A, B](fa: List[A])(f: A => B): List[B] = fa map f
}

// Test value of type Sq[List[Int]]:
val matrix2x2List: Sq[List[Int]] = Next(Matrix {
  case (None, None)         => List(0, 10, 100)
  case (None, Some(_))      => List(1, 11, 101)
  case (Some(_), None)      => List(2, 12, 102)
  case (Some(_), Some(_))   => List(3, 13, 103)
})

// Apply ‘sequence‘ to the test value. The result has type List[Sq[Int]].
val list2x2Matrix: List[Sq[Int]] = sequence(matrix2x2List)

// Visualize the result by converting it to nested lists.

scala> toSeqSeq(list2x2Matrix)
res1: List[List[List[Int]]] = List(List(List(0, 1), List(2, 3)), List(List(10, 11), List(12, 13)),
     List(List(100, 101), List(102, 103)))
```

Figure 12.3: Tests for the `sequence` method implemented in Figure 12.2.

Let us test this code:

```
scala> access(matrix2x2, 0, 1)
res0: Int = 12
```

Figure 12.2 shows the complete code of `sequence` for `Sq`. For testing (Figure 12.3), we define a value `matrix2x2List` that represents a square matrix of lists:

$$\text{matrix2x2List} : \text{Sq}^{\text{List}^{\text{Int}}} \quad , \qquad \text{matrix2x2List} \stackrel{\text{def}}{=} \left| \begin{array}{cc} [0, 10, 100] & [1, 11, 101] \\ [2, 12, 102] & [3, 13, 103] \end{array} \right| \quad .$$

Applying `sequence` to that value, we obtain a list of 3 square matrices:

$$\text{seq}_{\text{Sq}}\,(\text{matrix2x2List}) = \left[\left| \begin{array}{cc} 0 & 1 \\ 2 & 3 \end{array} \right| , \left| \begin{array}{cc} 10 & 11 \\ 12 & 13 \end{array} \right| , \left| \begin{array}{cc} 100 & 101 \\ 102 & 103 \end{array} \right| \right] \quad .$$

This represents a kind of transposition operation for tensors of dimension $2 \times 2 \times 3$.

While the type `Sq[A]` assures (at compile time) that all matrices have consistent shapes, it is hard to use because of complicated type parameters and deeply nested type constructors. Usually, square matrices and other tensor-like quantities are represented by flat arrays with an interface that recalculates the indices. When the dimensions of the matrices are known at compile time, one could use macros or other metaprogramming features to assure that all matrix operations are consistent, without resorting to complicated type constructors.

Part III

Advanced level

13 Free typeclass constructions

Free typeclass constructions (free monoids, free functors, free monads, free applicative functors, and so on) are used to implement the DSL (domain-specific language) design pattern. As a first motivation, we will look at how the free monad emerges from elaborating a certain kind of DSL in the functional programming paradigm.

13.1 Motivation for the free monad: create a DSL in five stages

The main point of using a DSL is to separate the description of computations from the process of their evaluation. One creates a **DSL program** as a data structure that fully describes what computations and side effects need to be run but does not actually perform any of those computations or side-effects. Helper functions are implemented for creating the DSL program values, for combining several DSL programs into larger ones, and for interpreting (or "running") DSL programs.

This design pattern is known as an **embedded DSL**, meaning that it is implemented not as a separate new language but as a library in an existing programming language (e.g., Scala). This chapter only considers embedded DSLs and calls them just "DSL''s for brevity.

We will look at two examples: a DSL for file operations and a DSL for complex numbers. Refining and refactoring these DSLs to make them more powerful and safer to use, we will obtain a construction known as the "free monad".

13.1.1 Stage 1: unevaluated expression trees

The first example is a DSL for reading and writing files at given file paths. Direct Scala code for such operations may look like this:

```
import java.nio.file.{Files, Paths}
val p = Paths.get("config_location.txt")
val configLocation = Paths.get(new String(Files.readAllBytes(p)))
val config = new String(Files.readAllBytes(configLocation))
```

The DSL will represent these operations by the case classes `Val`, `Path`, and `Read`:

```
sealed trait PrgFile
final case class Val(s: String)   extends PrgFile
final case class Path(p: PrgFile) extends PrgFile
final case class Read(p: PrgFile) extends PrgFile
```

A DSL program of type `PrgFile` is a value consisting of many nested case classes:

```
val prgFile: PrgFile = Read(Path(Read(Path(Val("config_location.txt")))))
```

Such values represent an *unevaluated* expression tree corresponding to the operations that need to be performed. To actually perform those operations and extract the final `String` value, we use a "runner" function:

```
def runFile: PrgFile => String = {
  case Val(s)          => s
  case Path(p)         => "path=" + runFile(p)                // For debugging.
  case Read(Path(p))   => new String(Files.readAllBytes(Paths.get(runFile(p))))
  case x               => throw new Exception(s"Illegal PrgFile operation: $x")
}
```

To test this code, we prepare a file `config.txt` containing the text `version = 1` and a file `config_location.txt` containing the text `config.txt`. Then we can run the DSL program and get the result:

```
scala> runFile(prgFile)
res2: String = "version = 1"
```

The second example is a DSL for calculations with complex numbers. Begin by implementing a data structure (`case class Complex`) with some operations:

```
final case class Complex(x: Double, y: Double) {
  def +(other: Complex): Complex = Complex(x + other.x, y + other.y)
  def *(other: Complex): Complex = Complex(x * other.x - y * other.y, x * other.y + y * other.x)
  def conj: Complex = Complex(x, -y)
  def phase: Double = math.atan2(y, x)         // Obtain the phase of a complex number.
  def rotate(alpha: Double): Complex = this * Complex(math.cos(alpha), math.sin(alpha))
}

val a = Complex(1, 2)          // a = 1+2i
val b = a * Complex(3, -4)     // b = (1+2i)(3-4i) = 11+2i
val c = b.conj                 // c = 11-2i

scala> c
res0: Complex = Complex(11.0, -2.0)

scala> c.rotate(Complex(0, 1).phase) // Multiply by i
res1: Complex = Complex(2.000000000000001, 11.0)
```

Instead of running a complex-number program directly, the DSL pattern first creates a data structure that contains an unevaluated expression tree describing what needs to be computed. Separate case classes are used for different operations as well as for inserting a literal `Complex` value (`Val`):

```
sealed trait PrgComplex
final case class Val(c: Complex)                      extends PrgComplex
final case class Add(p1: PrgComplex, p2: PrgComplex)  extends PrgComplex
final case class Mul(p1: PrgComplex, p2: PrgComplex)  extends PrgComplex
final case class Conj(p: PrgComplex)                  extends PrgComplex
final case class Phase(p: PrgComplex)                 extends PrgComplex
final case class Rotate(p: PrgComplex, a: PrgComplex) extends PrgComplex
```

Complex-number calculations are now represented by nested case classes:

```
val prgComplex1: PrgComplex = Conj(Mul(Val(Complex(1, 2)), Val(Complex(3, -4))))
val prgComplex2: PrgComplex = Rotate(prgComplex1, Phase(Val(Complex(0, 1))))
```

A simple runner for the DSL programs of type `PrgComplex` can be written like this:

```
def runComplex: PrgComplex => Complex = {
  case Val(c)             => c
  case Add(p1, p2)        => runComplex(p1) + runComplex(p2)
  case Mul(p1, p2)        => runComplex(p1) * runComplex(p2)
  case Conj(p)            => runComplex(p).conj
  case Phase(p)           => Complex(runComplex(p).phase, 0) // Pretend the phase is a Complex value.
  case Rotate(p, alpha)   => runComplex(p).rotate(runComplex(alpha).x) // Here alpha must be a phase.
}
```

We can now apply `runComplex` to some DSL programs and get the results as `Complex` values:

```
scala> runComplex(prgComplex1)
res2: Complex = Complex(11.0, -2.0)

scala> runComplex(prgComplex2)
res3: Complex = Complex(2.000000000000001, 11.0)
```

Although the DSLs are simple, we already get several benefits. By representing file operations as values of type `PrgFile` and complex-number calculations as values of type `PrgComplex`, we can compose, verify, or optimize our DSL programs *before* running them. We may also implement different

runners that execute the same DSL program on a remote computer, on a distributed file system, on GPUs instead of CPUs, or in a test-only sandbox environment, with logging or benchmarking.

13.1.2 Stage 2: implementing type safety in the DSLs

The two DSLs defined in the previous section are not fully type-checked at compile time. This becomes clear by looking at the DSL program `prgFile` shown in the previous section. The code of `runFile` assumes that the `Read` operation is always applied to a `Path`. However, this assumption is not enforced by the DSL. When composing a larger DSL program from separately defined parts, one could by mistake create an invalid DSL program such as `Read(Read(Val("file")))` where the `Read` operation is used incorrectly. Running this program causes a run-time error:

```
scala> runFile(Read(Read(Val("file"))))
java.lang.Exception: Illegal PrgFile operation: Read(Read(Val(file)))
```

All DSL program values are of the same type (`PrgFile`) regardless of their intended meaning. So, the Scala compiler cannot verify that we are using the DSL correctly. It would be better if the DSL program `Read(Read(Val("file")))` failed to compile. To achieve that, we need to use different Scala types for DSL programs returning a `String` and a `Path`. So, let us replace the type `PrgFile` by a type constructor `PrgFile[A]`, representing a DSL program that will return a value of type `A` when we run it:

```
import java.nio.file.{Path => JPath}
import java.nio.file.{Files, Paths}
sealed trait PrgFile[A]
final case class Val(s: String)           extends PrgFile[String]
final case class Path(p: PrgFile[String]) extends PrgFile[JPath]
final case class Read(p: PrgFile[JPath])  extends PrgFile[String]

def runFile[A]: PrgFile[A] => A = {
  case Val(s)   => s
  case Path(p)  => Paths.get(runFile(p))
  case Read(p)  => new String(Files.readAllBytes(runFile(p)))
}
```

Our example DSL is now fully type-safe. Invalid programs are rejected at compile time:

```
val prgFile: PrgFile[String] = Read(Path(Read(Path(Val("config_location.txt")))))

scala> runFile(prgFile)
res3: String = "version = 1"

scala> runFile(Read(Read(Val("file"))))
type mismatch;
 found   : Val
 required: PrgFile[java.nio.file.Path]
val res4 = runFile(Read(Read(Val("file"))))
                                 ^
Compilation Failed
```

A similar problem exists in our DSL for complex numbers. The code of `runComplex` assumes that the `Rotate` operation is always applied to a `Phase`, but the DSL does not enforce that assumption. Both arguments of the `Rotate` operation are values of type `PrgComplex`, and so the Scala compiler is unable to verify that the DSL is being used correctly. If the programmer uses a plain complex number (`Val`) instead of a `Phase` value, the program will return an incorrect result *without* any indication of error:

```
val prgComplex3: PrgComplex = Rotate(prgComplex1, Val(Complex(0, 1)))     // Forgot Phase()!

scala> runComplex(prgComplex3)    // The result is incorrect, but there is no error message!
val res5: Complex = Complex(11.0, -2.0)
```

To achieve type safety, we add a type parameter to `PrgComplex`. A DSL program of type `PrgComplex[A]` will return a value of type `A` when run. (So, the new DSL programs will be able to compute values

of any type, not only of type `Complex`.) The new definitions of the case classes are:

```
sealed trait PrgComplex[A]
final case class Val(c: Complex)                                      extends PrgComplex[Complex]
final case class Add(p1: PrgComplex[Complex], p2: PrgComplex[Complex]) extends PrgComplex[Complex]
final case class Mul(p1: PrgComplex[Complex], p2: PrgComplex[Complex]) extends PrgComplex[Complex]
final case class Conj(p: PrgComplex[Complex])                         extends PrgComplex[Complex]
final case class Phase(p: PrgComplex[Complex])                        extends PrgComplex[Double]
final case class Rotate(p: PrgComplex[Complex], a: Phase)             extends PrgComplex[Complex]
```

The revised code of `runComplex` is clearer because phases are no longer wrapped in a `Complex` type:

```
def runComplex[A]: PrgComplex[A] => A = {
  case Val(c)             => c
  case Add(p1, p2)        => runComplex(p1) + runComplex(p2)
  case Mul(p1, p2)        => runComplex(p1) * runComplex(p2)
  case Conj(p)            => runComplex(p).conj
  case Phase(p)           => runComplex(p).phase
  case Rotate(p, alpha)   => runComplex(p).rotate(runComplex(alpha))
}
```

The example programs, `prgComplex1` and `prgComplex2`, do not change except for their type:

```
val prgComplex1: PrgComplex[Complex] = Conj(Mul(Val(Complex(1, 2)), Val(Complex(3, -4))))

val prgComplex2: PrgComplex[Complex] = Rotate(prgComplex1, Phase(Val(Complex(0, 1))))

scala> runComplex(prgComplex1)
res6: Complex = Complex(x = 11.0, y = -2.0)

scala> runComplex(prgComplex2)
res7: Complex = Complex(x = 2.000000000000001, y = 11.0)
```

Since `Rotate` now requires a `PrgComplex[Double]`, forgetting to use `Phase` will cause a type error:

```
scala> val prgComplex3 = Rotate(prgComplex1, Val(Complex(0, 1)))
<console>:1: type mismatch;
 found   : Val
 required: PrgComplex[Double]
  case Phase(p)           => runComplex(p)
                                        ^
Compilation Failed
```

13.1.3 Stage 3: implementing bound variables

Another limitation of our DSLs is that we cannot define and use new variables within a DSL program. We also cannot use any non-DSL code, such as a library for file manipulation, a numerical algorithms library for complex numbers, or any other Scala code.

Both of these limitations would be removed if the DSL could define *new Scala variables* and set them to values computed within the DSL. We would then be able to add arbitrary Scala code for working with those variables. Let us see how this feature may be implemented.

As an example, consider the task of reading a file only if it exists and reporting an error otherwise. A direct (non-DSL) code for this computation looks like this:

```
val p = Paths.get("config_location.txt")
val result = if (Files.exists(p)) new String(Files.readAllBytes(p)) else "No file."
```

Trying to translate the above code to the `PrgFile` DSL, we find that we need to set the Scala variable `p` to the path computed by a previous operation. Here is a first attempt:

```
val p: JPath = runFile(Path(Val("config_location.txt")))
val prg: PrgFile[String] = if (Files.exists(p)) Read(p) else Val("No file.")
```

There are two problems with this code. The first problem is that `Read(p)` does not compile because

the argument of `Read` should be a `PrgFile[JPath]` rather than just a `JPath`. To solve this problem, we generalize the `Val` class to hold values of arbitrary type:

```
final case class Val[A](a: A) extends PrgFile[A]
```

We then replace `Read(p)` by `Read(Val(p))` in the code above. The code of `run` remains unchanged.

The second problem is that we are using `runFile` in the middle of a DSL program. This breaks the assumption that a DSL program merely describes the computations without executing them. The use of a specific runner inside a DSL program will also prevent us from being able to apply a different runner to the entire DSL program later.

To solve this problem, we introduce a new DSL operation that binds Scala variables to values returned by other DSL operations. A standard way of creating bound variables is by using those variables as arguments of nameless functions. So, let us consider the nameless function:

```
{ p => if (Files.exists(p)) Read(Val(p)) else Val("No file.") }
```

This function creates a local variable `p` and uses it directly in an expression of type `PrgFile[String]`. We can call this function at the time when the entire DSL program is run. So, we replace the code `val p = runFile(...)` by a new DSL operation that we will call `Bind`. The above code that computes `p` and `prg` is replaced by:

```
val prg2: PrgFile[JPath] = Path(Val("config.txt"))
val prg: PrgFile[String] = Bind(prg2){ p => if (Files.exists(p)) Read(Val(p)) else Val("No file.") }
```

Here, the case class `Bind` has two curried arguments: a value of type `PrgFile[JPath]` and a function of type `JPath => PrgFile[String]`. That function will be called by `runFile` when we run the DSL program. At that time, the argument `p` will be set to the value obtained by running `prg2`. We will implement this logic in the new code for `runFile` (shown below).

In a different DSL program, we may need to use different types instead of `String` and `JPath`. So, let us replace those types by type parameters `A` and `B`. The declaration of the class `Bind` becomes:

```
final case class Bind[A, B](pa: PrgFile[B])(f: B => PrgFile[A]) extends PrgFile[A]
```

After adding the `Bind`-handling code to the runner, the implementation of the DSL is:

```
sealed trait PrgFile[A]
final case class Val[A](a: A)                                        extends PrgFile[A]
final case class Bind[A, B](pa: PrgFile[B])(val f: B => PrgFile[A]) extends PrgFile[A]
final case class Path(p: PrgFile[String])                            extends PrgFile[JPath]
final case class Read(p: PrgFile[JPath])                             extends PrgFile[String]

def runFile[A]: PrgFile[A] => A = {
  case Val(a)          => a
  case bind@Bind(pa)   => runFile(bind.f(runFile(pa)))
  case Path(p)         => Paths.get(runFile(p))
  case Read(p)         => new String(Files.readAllBytes(runFile(p)))
}
```

Let us explain certain features of Scala used in this DSL. Since the arguments of the constructor `Bind[A, B]` are curried, the first curried argument (of type `PrgFile[B]`) will allow the Scala compiler to determine the type parameter `B` automatically. At the same time, `Bind`'s second curried argument may be written as a nameless function in a separate set of braces:

```
val prg: PrgFile[String] = Bind(prg2){ p => if (Files.exists(p)) Read(Val(p)) else Val("No file.") }
```

Because the type parameter `B` is already fixed, the argument `p` may be written without a type annotation (in this case, it would have been `p:JPath`). This makes the DSL easier to use. However, the underlying implementation becomes more complicated. Scala's pattern matching syntax is limited to the first curried argument. So, the second curried argument of `Bind` needs a `val` declaration and is accessed using another pattern variable (here called `bind`). Finally, the type parameters `A` and `B` are defined such that `Bind[A, B]` is a subtype of `PrgFile[A]` and not of `PrgFile[B]`. This allows Scala's type checking to handle correctly the unknown type `B` in the pattern `Bind(pa)`.

The new `Bind` operation allows one part of a DSL program to use values computed by other parts. Since a `Bind` value contains a *Scala function* (of type `B => PrgFile[A]`), that function may run arbitrary Scala code, not limited to DSL operations, in order to compute a result of type `PrgFile[A]`. All non-DSL code is enclosed inside a `Bind` and will be evaluated only when the DSL program is run:

```
scala> runFile(prg)
res8: String = "version = 1"
```

As before, a value of type `PrgFile[A]` describes a computation but does not execute it.

13.1.4 Stage 4: refactoring to a monadic DSL

The next observation is that `Bind` and `Val` have the same type signatures as the standard `flatMap` and `pure` methods of a monad. For convenience, let us define those methods on the `PrgFile` class:

```
sealed trait PrgFile[A] {
  def flatMap[B](f: A => PrgFile[B]): PrgFile[B] = Bind(this)(f)
  def map[B](f: A => B): PrgFile[B] = flatMap(f andThen PrgFile.pure)
}
final case class Val[A](a: A)                                        extends PrgFile[A]
final case class Bind[A, B](pa: PrgFile[B])(val f: B => PrgFile[A]) extends PrgFile[A]
final case class Path(p: PrgFile[String])                            extends PrgFile[JPath]
final case class Read(p: PrgFile[JPath])                             extends PrgFile[String]

object PrgFile {
  def pure[A](a: A): PrgFile[A] = Val(a)
}
```

Here we defined `map` via `flatMap` by using a monad's right identity law, as shown in Eq. (10.21).

Like other DSL operations, the methods `map`, `flatMap`, and `pure` do not run or evaluate any parts of a DSL program. They only create some nested data structures containing not-yet-called functions. The necessary logic will be executed when the `run` method is invoked. At that time, the entire DSL program will be converted into a result value.

Because `map` and `flatMap` are defined, DSL programs can now be written as functor blocks:

```
val prg: PrgFile[String] = for {
  p <- Path(Val("config.txt"))
  r <- if (Files.exists(p)) Read(Val(p)) else Val("No file.")
} yield r
```

Longer DSL programs may be composed from shorter ones:

```
def readFileContents(filename: String): PrgFile[String] = for {
  path <- Path(Val(filename))
  text <- if (Files.exists(path)) Read(Val(path)) else Val("No file.")
} yield text

val prg2: PrgFile[String] = for {
  str <- readFileContents("config.txt")
  result <- if (str.nonEmpty) readFileContents(str) else Val("No filename.")
} yield result

scala> runFile(prg2)
res9 : String = "version = 1"
```

Another benefit of the monadic refactoring is that `flatMap` provides a general way of applying a file operation to a value wrapped under `PrgFile`. So, we may simplify the case classes `Path` and `Read` by removing the `PrgFile` wrapping:

```
final case class Path(p: String) extends PrgFile[JPath]
final case class Read(p: JPath)  extends PrgFile[String]
```

This simplifies the usage of the DSL since we may write, e.g., just `Read(path)` instead of `Read(Val(path))`. The code of `runFile` is also shortened:

```
def runFile[A]: PrgFile[A] => A = {
  case Val(a)          => a
  case bind@Bind(pa)   => runFile(bind.f(runFile(pa)))
  case Path(p)         => Paths.get(p)
  case Read(p)         => new String(Files.readAllBytes(p))
}
```

13.1.5 Stage 5: refactoring to reuse common code

Let us rewrite our second example (a DSL for complex numbers) in the same way:

```
sealed trait PrgComplex[A] {
  def flatMap[B](f: A => PrgComplex[B]): PrgComplex[B] = Bind(this)(f)
  def map[B](f: A => B): PrgComplex[B] = flatMap(f andThen PrgComplex.pure)
}
object PrgComplex {
  def pure[A](a: A): PrgComplex[A] = Val(a)
}

final case class Val[A](a: A)                                          extends PrgComplex[A]
final case class Bind[A, B](pa: PrgComplex[B])(val f: B => PrgComplex[A]) extends PrgComplex[A]
final case class Add(x: Complex, y: Complex)                           extends PrgComplex[Complex]
final case class Mul(x: Complex, y: Complex)                           extends PrgComplex[Complex]
final case class Conj(x: Complex)                                      extends PrgComplex[Complex]
final case class Phase(p: Complex)                                     extends PrgComplex[Double]
final case class Rotate(p: Complex, p: Phase)                          extends PrgComplex[Complex]

def runComplex[A]: PrgComplex[A] => A = {
  case Val(a)                => a
  case bind@Bind(pa)         => runComplex(bind.f(runComplex(pa)))
  case Add(p1, p2)           => p1 + p2
  case Mul(p1, p2)           => p1 * p2
  case Conj(p)               => p.conj
  case Phase(p)              => p.phase
  case Rotate(p, Phase(a))   => p.rotate(a.phase)
}
```

Comparing the code of `runFile` and `runComplex`, we notice that the `Val` and `Bind` operations are implemented in the same way. The differences are in the code for operations specific to file handling or to complex numbers. So, let us refactor these data structures, separating the domain-specific operations and reusing the common code. We can gather all the domain-specific operations in a new type constructor that we call an **effect constructor**. The common code will then use the effect constructor as a type parameter. Let us now refactor the two example DSLs in this manner.

The effect constructor for the `PrgFile` DSL should contain the operations `Path` and `Read`. Let us copy the existing case classes for these operations but rename their parent type to `PrgFileC`. We will also move the relevant parts of `runFile` into a new function (`runFileC`) that we call an **effect runner**:

```
sealed trait PrgFileC[A]
final case class Path(p: String) extends PrgFileC[JPath]
final case class Read(p: JPath)  extends PrgFileC[String]

def runFileC[A]: PrgFileC[A] => A = {
  case Path(p)   => Paths.get(p)
  case Read(p)   => new String(Files.readAllBytes(p))
}
```

The code of `PrgFile` becomes shorter since we moved some code out of it:

```
// The same code for sealed trait PrgFile[A] and object PrgFile.
// The case classes are changed to:
final case class Val[A](a: A) extends PrgFile[A]
final case class Bind[A, B](pa: PrgFile[B])(val f: B => PrgFile[A]) extends PrgFile[A]
```

```
final case class Op[A](op: PrgFileC[A]) extends PrgFile[A]  // Wrap custom operations.

def runFile[A]: PrgFile[A] => A = {
  case Val(a)          => a
  case bind@Bind(pa)   => runFile(bind.f(runFile(pa)))
  case Op(op)          => runFileC(op) // Run the custom operation and get the result.
}
```

The code of `PrgComplex` is refactored in a similar way:

```
// The same code for ‘sealed trait PrgComplex[A]‘ and ‘object PrgComplex‘.
// The case classes are changed to:
final case class Val[A](a: A) extends PrgComplex[A]
final case class Bind[A, B](pa: PrgComplex[B])(val f: B => PrgComplex[A]) extends PrgComplex[A]
final case class Op[A](op: PrgComplexC[A]) extends PrgComplex[A]     // Wrap custom operations.

def runComplex[A]: PrgComplex[A] => A = {
  case Val(a)              => a
  case bind@Bind(pa)       => runComplex(bind.f(runComplex(pa)))
  case Op(op)              => runComplexC(op)    // Run the custom operation and get the result.

sealed trait PrgComplexC[A]
final case class Add(x: Complex, y: Complex)      extends PrgComplexC[Complex]
final case class Mul(x: Complex, y: Complex)      extends PrgComplexC[Complex]
final case class Conj(x: Complex)                 extends PrgComplexC[Complex]
final case class Phase(p: Complex)                extends PrgComplexC[Double]
final case class Rotate(p: Complex, alpha: Phase) extends PrgComplexC[Complex]

def runComplexC[A]: PrgComplexC[A] => A = {
  case Add(p1, p2)           => p1 + p2
  case Mul(p1, p2)           => p1 * p2
  case Conj(p)               => p.conj
  case Phase(p)              => p.phase
  case Rotate(p, Phase(a))   => p.rotate(a.phase)
}
```

The code of `PrgComplex` is now the same as the code of `PrgFile` except for using a different effect constructor (`PrgComplexC` instead of `PrgFileC`) and the corresponding runner (`runComplexC` instead of `runFileC`). The effect constructor and its runner encapsulate the entire domain-specific logic. The code in the classes `PrgComplex` and `PrgFile` is only concerned with providing the monadic functionality to the DSL. We can replace those two classes by a single class (called, say, `MonadDSL`) that takes the effect constructor as a *type parameter* `F`. Since the type parameter `F` is itself a type constructor, we must declare it via the Scala syntax `F[_]`. The code is:

```
sealed trait MonadDSL[F[_], A] {
  def flatMap[B](f: A => MonadDSL[F, B]): MonadDSL[F, B] = Bind(this)(f)
  def map[B](f: A => B): MonadDSL[F, B] = flatMap(f andThen MonadDSL.pure)
}
object MonadDSL {
  def pure[F[_], A](a: A): MonadDSL[F, A] = Val(a)
}
final case class Val[F[_], A](a: A) extends MonadDSL[F, A]
final case class Bind[F[_], A, B](pa: MonadDSL[F, B])(val f: B => MonadDSL[F, A]) extends
     MonadDSL[F, A]
final case class Op[F[_], A](op: F[A]) extends MonadDSL[F, A] // Wrap all domain-specific operations.
```

The runner for `MonadDSL` needs to run the effects described by the effect constructor `F`. We try passing `F`'s runner as an additional curried argument to `MonadDSL`'s runner:

```
def run[F[_], A](runner: F[A] => A): MonadDSL[F, A] => A = { // This code does not compile.
  case Val(a)          => a
  case bind@Bind(pa)   => run(runner)(bind.f(run(runner)(pa)))
  case Op(op)          => runner(op)
}
```

However, this code gives a type error. The type of `bind.f` is `B => MonadDSL[F, A]`, so the argument of `bind.f` must have type `B`. A value of type `B` will be produced by `run(runner)(pa)` only if we set type parameters as `run[F, B]`, and only if `runner` has type `F[B] => B`. But the given argument `runner` has a fixed type `F[A] => A` and cannot be used with type `B`.

The function `run(runner)` will work with arbitrary types only if `runner` has type `F[X] => X` where `X` remains free and is *not* known in advance. The type signatures of the effect runners `runComplexC` and `runFileC` already have that form:

```
def runComplexC[X]: PrgComplexC[X] => X    // PrgComplexC[X] => X with arbitrary types X
def runFileC[X]: PrgFileC[X] => X          // PrgFileC[X] => X with arbitrary types X
```

We need to pass those functions as the `runner` arguments to `run(runner)` while not losing the freedom provided by the type parameter `X`.

To achieve that, we define a new type `Runner` as a Scala trait encapsulating the type signature of an effect runner. The two effect runners can then be declared as values of type `Runner`:

```
trait Runner[F[_]] { def run[X]: F[X] => X }
val runnerComplex = new Runner[PrgComplexC] { def run[X]: PrgComplexC[X] => X = runComplexC[X] }
val runnerFile = new Runner[PrgFileC] { def run[X]: PrgFileC[X] => X = runFileC[X] }
```

Function values that encapsulate a type parameter are called **polymorphic functions**. The type notation for the type `Runner` is:

$$\text{Runner}^F \stackrel{\text{def}}{=} \forall X.\, F^X \to X \quad .$$

The universal quantifier ($\forall X$) indicates that a value of type `Runner[F]` still has the freedom of using any type `X` when the method `run` is called.

In Scala 3, the type of an effect runner is written in a shorter syntax:

```
type Runner[F[_]] = [X] => F[X] => X
```

This corresponds to the type notation $\forall X.\, F^X \to X$. The programmer does not need to declare a trait `Runner` in Scala 3 because the compiler does that automatically for polymorphic functions.

Using the `Runner` type, we rewrite the code for the function `run` as:

```
def run[F[_], A](runner: Runner[F]): MonadDSL[F, A] => A = {
  case Val(a)          => a
  case bind@Bind(pa)   => run(runner)(bind.f(run(runner)(pa)))
  case Op(op)          => runner.run(op)
}
```

We can now define `PrgComplex` through `MonadDSL` and test the new code:

```
type PrgComplex[A] = MonadDSL[PrgComplexC, A]
val prgComplex: PrgComplex[Complex] = for {
  x <- Op(Add(Complex(1, 1), Complex(0, 1)))
  y <- Op(Mul(x, Complex(3, -4)))
  z <- Op(Conj(y))
  r <- Op(Rotate(x, Phase(Complex(0, 1))))
} yield r

scala> runComplex(runnerComplex)(prgComplex2)
res0: Complex = Complex(x = 2.000000000000001, y = 11.0)
```

13.1.6 A first recipe for monadic DSLs

The previous section gave motivation for defining the type constructor `MonadDSL[F[_], A]`. That type constructor is called the "free monad on `F`", for reasons explained later in this chapter. Note that `MonadDSL[F[_], A]` supports the monad's methods `map`, `flatMap`, and `pure`, which `F` does not necessarily have. So, we may view `MonadDSL[F[_], A]` as a special wrapper that adds the monadic variable-binding functionality to any given DSL whose operations are described by an effect constructor `F`.

Below we will show that monadic DSL programs will satisfy the monad laws *after* the effects are run (but not necessarily before that!). Let us now summarize a recipe for using `MonadDSL` in practice.

Begin by writing down the types of the available operations in the DSL. The operations may be functions between fixed known types, such as Int → Int → String or String → 𝟙. Operations may be also functions with type parameters, for example, $A \times \text{Int} \to \text{String}$ where A is a type parameter. In every case, we need to write down the types in the form of a simple (non-curried) function, such as $P \to Q$, where P and Q are type expressions possibly depending on some type parameters.

The next step is to define a new type constructor `F[_]` that will serve as the DSL's effect constructor:

```
sealed trait F[_]
```

The effect constructor `F` will be a disjunctive type whose parts correspond to each of the domain-specific operations. For an operation with type $P \to Q$, define the corresponding case class as:

```
final case class ReadPWriteQ(x: P) extends F[Q]
```

The names of case classes (such as `ReadPWriteQ`) may be chosen for clarity.

It is an important part of the recipe that the result type (`Q`) is wrapped in the effect constructor `F` while the argument type (`P`) is *not*. For example, the `PrgFile` DSL described in the previous section has two operations with types `String => JPath` and `JPath => String`. The corresponding effect constructor (`PrgFileC`) is defined by:

```
sealed trait PrgFileC[_]
final case class Path(p: String) extends PrgFileC[JPath]
final case class Read(p: JPath)  extends PrgFileC[String]
```

Typically, a type constructor `F` defined in this way will be an "unfunctor". In the example shown above, `PrgFileC` cannot be a functor because it is impossible to create values of type `PrgFileC[A]` with an arbitrary type `A` (the type `A` must be either `JPath` or `String`). However, in some cases the effect constructor `F` could be itself a lawful functor.

Next, we implement an "effect runner" for `F`. An **effect runner** for an effect constructor `F` is a polymorphic function with the type signature $\forall X.\, F^X \to X$:

```
val runF: Runner[F] = new Runner[F] { def run[X]: F[X] => X = ??? }
```

If the definition of `F` allows only certain types `A` in values of type `F[A]` then the runner only needs to work for those types. In the `PrgFile` DSL, the runner only needs to work with arguments of types `PrgFileC[JPath]` and `PrgFileC[String]` because those are the only allowed types of values that can be wrapped by the unfunctor `PrgFileC`. The runner will never be given an argument of type `PrgFile[X]` with an arbitrary unknown `X`. However, different effect constructors may have different allowed types, and `MonadDSL` needs to be able to work with any effect constructor. So, the runner's type signature still needs a general type parameter `X`:

```
val runnerFile = new Runner[PrgFileC] { def run[X]: (PrgFileC[X] => X) = runFileC[X] }
```

We now define the monadic DSL type (`MyDSL`) and its runner function (`runMyDSL`) as:

```
type MyDSL[A] = MonadDSL[F, A]
def runMyDSL[A]: MyDSL[A] => A = MonadDSL.run(runF)
```

DSL programs may be written using functor blocks:

```
type FileDSL[A] = MonadDSL[PrgFileC, A]
def runFileDSL[A]: FileDSL[A] => A = MonadDSL.run(runnerFile)
val prgFile1: FileDSL[String] = for {
  x <- Op(Path("config.txt"))
  y <- Op(Read(x))
} yield y

scala> runFileDSL(prgFile1)
res0: String = "version = 1"
```

Note that each DSL operation needs to be wrapped in an `Op` case class. To make the code shorter, we may define an implicit conversion from `PrgFileC[A]` to `Op[A]`:

```
implicit def toOp[A](p: PrgFileC[A]): Op[A] = Op(p) // *** check if this code actually works
val prgFile2: FileDSL[String] = for { // Does it help if we begin the functor block with a Pure()?
  x <- Path("config.txt")
  y <- Read(x)
} yield y

scala> runFileDSL(prgFile1)
res1: String = "version = 1"
```

Does the type constructor `MonadDSL[F, A]` satisfy the laws of the monad? We can quickly find out that it does *not*. The left identity law of `flatMap` fails:

```
MonadDSL.pure(a).flatMap(f) == Bind(Val(a))(f) // This should equal f(a) by the left identity law.
```

However, the law will hold *after* we apply a runner function to both sides. For brevity, let us denote `MonadDSL`'s runner function by `r(x)` instead of `run(runner)(x)`. The code of `run` shows that `r(Bind(p)(f)) == r(f(r(p)))` and `r(Val(a)) == a`. Then we find that both sides of the law give `r(f(a))` when run:

```
r(MonadDSL.pure(a).flatMap(f)) == r(Bind(Val(a))(f)) == r(f(a))
```

The associativity law (10.4) of `flatMap` also does not hold directly for `MonadDSL` values but will hold after a `MonadDSL` program has been run. To see that, first compare the two sides of the law (10.4):

```
/* left-hand side:  */   p.flatMap(x => f(x).flatMap(g)) == Bind(p)(x => Bind(f(x))(g))
/* right-hand side: */   p.flatMap(f).flatMap(g) == Bind(Bind(p)(f))(g)
```

The law fails since the two data structures are not the same. When we apply a runner `r` to both sides, we get:

```
/* left-hand side:  */   r(Bind(p)(x => Bind(f(x))(g))) == r(Bind(f(r(p)))(g)) == r(g(r(f(r(p)))))
/* right-hand side: */   r(Bind(Bind(p)(f))(g)) == r(g(r(Bind(p)(f)))) == r(g(r(f(r(p)))))
```

We may say that failures of the monad laws are "not observable" since the laws will hold after running the DSL programs.

Could we implement `MonadDSL`'s `flatMap` method differently so that the monad laws already hold before applying a runner? To satisfy the left identity law, we need to avoid creating a nested structure of the form `Bind(Val(a))` when applying `flatMap` to a value of the form `Val(a)`. Instead, `Val(a).flatMap(f)` should return `f(a)` as required by the left identity law. To satisfy the associativity law, we need to avoid creating a nested structure of the form `Bind(Bind(...))` when applying `flatMap` to a `Bind` value. Instead, applying `flatMap` to `Bind` should directly return the value required by the associativity law. The new code of `flatMap` is:

```
sealed trait MonadDSL[F[_], A] { *** check that this code works
  def flatMap[B](f: A => MonadDSL[F, B]): MonadDSL[F, B] = this match {
    case Val(a)          => f(a)
    case bind@Bind(pa)   => Bind(pa)(x => Bind(bind.f(x))(f))
    case _               => Bind(this)(f)
  }
  ... // Other code remains unchanged.
}
```

The resulting code creates fewer nested case classes in memory, which may improve performance.

Unlike other laws, the right identity law already holds for any value `p` of type `MonadDSL[F, A]`:

```
p.flatMap(f andThen MonadDSL.pure) == p.map(f)
```

The reason is that our current code implements `MonadDSL`'s `map` method via `flatMap` in exactly this way.

13.1.7 Running a DSL program into another monad: Handling errors

In the code from the previous section, one may use different effect runners with *the same* function `MonadDSL.run`. We may apply `MonadDSL.run` to any effect runner of type $\forall X.\, F^X \to X$ and obtain a monad runner of type `MonadDSL[F, A] => A`.

More generally, we may need to run the F-effects into *another monad*. Given an effect runner of type $\forall X.\, F^X \to M^X$, where M is some monad, we hope to obtain a general runner with type signature $\forall A.\, \text{MonadDSL}^{F,A} \to M^A$.

To see an example of using such general runners in practice, note that the execution of F-effects may produce errors.[1] For instance, the code of `runFileC` (the runner for the `PrgFile` DSL) will throw exceptions when files are not found or not readable. We may catch those exceptions with Scala's standard `Try` class. We will then need to replace a runner of type `PrgFileC[X] => X` by a polymorphic function of type `PrgFileC[X] => Try[X]`. The new code looks like this:

```
trait RunnerTry[F[_]] { def run[X]: F[X] => Try[X] }
val runnerFileTry = new RunnerTry[PrgFileC] { def run[X]: PrgFileC[X] => Try[X] = p =>
    Try(runFileC(p)) }
```

The code of the `MonadDSL` runner needs to be modified to accommodate the new type:

```
def runTry[F[_], A](runnerTry: RunnerTry[F]): MonadDSL[F, A] => Try[A] = {
  case Val(a)          => Success(a)
  case bind@Bind(pa)   => runTry(runner)(pa).flatMap(bind.f andThen runTry(runner)) // Use flatMap
     from Try.
  case Op(op)          => runner.run(op)
} *** check that this code works
```

Now we can generalize the code of `runTry` to an arbitrary monad M instead of `Try`. We note that `runTry` uses only two values specific to `Try`: the `flatMap` method of the `Try` monad and the `Success` case class, which corresponds to the `Try` monad's `pure` method. To generalize the code to another monad M instead of `Try`, we need to use M's `flatMap` and to replace `Success` by M's `pure`:

```
trait RunnerM[F[_], M[_]] { def run[X]: F[X] => M[X] }
def runM[F[_], M[_]: Monad, A](runnerM: RunnerM[F, M]): MonadDSL[F, A] => M[A] = {
  case Val(a)          => Monad[M].pure(a)
  case bind@Bind(pa)   => runM(runner)(pa).flatMap(bind.f andThen runM(runner)) // Use flatMap from
     M.
  case Op(op)          => runner.run(op)
} *** check that this code works
```

The use of general monadic runners (with an arbitrary monad M) allows the programmer to better separate the business logic of an application from its execution environment. For instance, M may itself be a monadic DSL with a different set of operations and its own effect runner that works more efficiently or supports a different run-time environment.

13.2 Different encodings of the free monad

13.2.1 Motivation

The type `MonadDSL[F, A]` is called a **free monad** on the type constructor `F`. The word "free" indicates that `MonadDSL` is free of any domain-specific code. We may view `MonadDSL[F, A]` as a wrapper that adds the monad's functionality to any type constructor `F` and creates a new monadic DSL based on `F`'s operations. The construction uses `F` as a type parameter, and so it works equally well with every `F`.

A monad must support the methods `pure`, `flatMap`, and `map`. The effect constructor `F` often does not have these methods. The `MonadDSL` constructor is able to convert an arbitrary `F` into a monad because `MonadDSL` defines the methods `pure`, `flatMap`, and `map` in a special way. Those methods create some data structures (nested case classes) in memory but do not perform any domain-specific effects or

[1] "A function that launches real-world missiles can run out of missiles." (A quote attributed to Simon Peyton Jones.)

actions. The effects will be performed at a later time by an effect runner (`runF` of type `F[A] => A`) when we apply `MonadDSL.run` to a DSL program. Effect runners are not part of the code of `MonadDSL` and will need to be supplied separately. So, we may run the same `MonadDSL` program using different runners.

The implementations of the monadic methods in `MonadDSL` satisfy our goal of producing a type-safe DSL out of a given set of domain-specific operations. But it turns out that `MonadDSL` is not the only way of turning a given type constructor `F` into a monad. Various presentations and blog posts about the free monad have shown different implementations that are not obviously equivalent to each other. Here are three examples that we will call `Free1`, `Free2`, and `Free3`.

In a 2012 blog post,[2] G. Gonzalez showed a free monad corresponding to this Scala code:

```
absract class Free1[F[_]: Functor, T] {
  def flatMap[A](f: T => Free1[F, A]): Free1[F, A] = this match {
    case Pure(t)      => f(a)
    case Flatten(p)   => Flatten(p.map(g => g.flatMap(f))) // Use F\textsf{'}s Functor instance.
  }
}
final case class Pure[F[_], T](t: T)                 extends Free1[F, T]
final case class Flatten[F[_], T](p: F[Free1[F, T]]) extends Free1[F, T]
```

In 2014,[3] R. Bjarnason presented the following implementation of a free monad:

```
sealed trait Free2[F[_], T] {
  def flatMap[A](f: T => Free2[F, A]): Free2[F, A] = this match {
    case Return(t)    => f(t)
    case Bind(p, g)   => Bind(p, (b => g(b) flatMap f))
  }
}
final case class Return[F[_], T](t: T)                          extends Free2[F, T]
final case class Bind[F[_], T, A](p: F[A], g: A => Free2[F, T]) extends Free2[F, T]
```

In 2016, K. Robinson gave a talk[4] where the code for the free monad looked like this:

```
sealed trait Free3[F[_], T] {
  def flatMap[A](f: T => Free3[F, A]): Free3[F, A] = FlatMap(this, f)
}
final case class Pure[F[_], T](t: T)                                     extends Free3[F, T]
final case class Suspend[F[_], T](f: F[T])                               extends Free3[F, T]
final case class FlatMap[F[_], T, A](p: Free3[F, A], g: A => Free3[F, T]) extends Free3[F, T]
```

The main differences between `Free1`, `Free2`, and `Free3` are in the definitions of the case classes and of the `flatMap` functions, so we omitted all other code here. The type of `Free3` is the same as `MonadDSL` except for renaming `Pure` to `Val`, `Suspend` to `Op`, and `FlatMap(f, g)` to `Bind(f)(g)`. However, `Free1` and `Free2` have different data in their case classes and are not obviously equivalent to each other (or to `Free3`). We call these implementations **encodings** of the free monad. The different codes implement the same idea (turning a type constructor `F` into a monad) in different ways.

To figure out how to use these codes in practice, a programmer might ask the following questions: Are the three encodings equivalent? What laws does their code need to satisfy in order to be considered "correct"? The `Free2` type has 2 case classes and `Free3` has 3; are there any other encodings of the free monad? Is there a systematic way of finding constructions that convert any type into a "free functor", a "free monoid", or into another "free" typeclass? The rest of this chapter will develop the required theory and show the derivations needed to answer these questions.

13.2.2 The raw tree encoding

To understand the relationships between the encodings of the free monad, let us first examine how the code of `MonadDSL` (or equivalently `Free3`) implements the `map` function. In `MonadDSL`, we implemented

[2] See http://www.haskellforall.com/2012/06/you-could-have-invented-free-monads.html
[3] See http://functionaltalks.org/2014/11/23/runar-oli-bjarnason-free-monad/
[4] See https://www.slideshare.net/KelleyRobinson1/why-the-free-monad-isnt-free-61836547

`map` through `flatMap` for brevity. As a result, our code for the `map` method is different from the code of the two other monadic methods (`pure` and `flatMap`) that merely create new values of the case classes `Val` and `Bind` without performing any computations.

We could implement `map` in a similar way if we add a new case class, say `FMap`, to `MonadDSL`. The result is a new encoding of the free monad that we may call `Free4` (since it uses 4 case classes):

```
sealed trait Free4[F[_], A] {
  def flatMap[B](f: A => Free4[F, B]): Free4[F, B] = Bind(this)(f)
  def map[B](f: A => B): Free4[F, B] = FMap(this)(f)
}
object Free4 {
  def pure[F[_], A](a: A): Free4[F, A] = Val(a)
}
final case class Val[F[_], A](a: A)                                          extends Free4[F, A]
final case class Bind[F[_], A, B](pa: Free4[F, B])(val f: B => Free4[F, A])  extends Free4[F, A]
final case class FMap[F[_], A, B](pa: Free4[F, B])(val f: B => A)            extends Free4[F, A]
final case class Op[F[_], A](op: F[A]) extends Free4[F, A] // Wrap all domain-specific operations.
```

The code of the runner needs to be revised accordingly:

```
def run[F[_], A](runner: Runner[F]): Free4[F, A] => A = {
  case Val(a)            => a
  case bind @ Bind(pa)   => run(runner)(bind.f(run(runner)(pa)))
  case fmap @ FMap(pa)   => fmap.f(run(runner)(pa))
  case Op(op)            => runner.run(op)
} *** check that this code works
```

The code of `Free4`'s methods `map`, `flatMap`, and `pure` merely wraps the arguments of each method into a case class without performing any computations with those arguments. We call this implementation of the free monad the **raw tree encoding**. The name means that a DSL program is a fully unevaluated (or "raw") expression tree representing all steps of the required computations.

13.2.3 Deriving reduced encodings of the free monad

In the raw tree encoding (`Free4`), *none* of the monad laws hold. This is not a problem in practice because the monad laws will hold after running a DSL program. However, we notice that the `Free3` encoding (which is the same as `MonadDSL` from Section 13.1.5) satisfies the monad's right identity law by design. A byproduct of that design is that `Free3` has fewer case classes than `Free4`. We also notice that the code of `Free2` uses just 2 case classes and satisfies all monad laws.

These observations suggest that we might find an encoding with fewer case classes (which we call a **reduced encoding**) if we use the monad laws when designing the free monad's type constructor. To see how this works, we will first reduce `Free4` to `Free3` and then `Free3` to `Free2` by imposing certain monad laws directly on the DSL program values. These derivations will also show that `Free2`, `Free3`, and `Free4` are equivalent to each other in their expressive power.[5]

To derive `Free3` from `Free4`, we use the right identity law of `flatMap` to express `map` through `flatMap`:

```
p.map(f) == p.flatMap (f andThen pure)
```

As we have already seen, this definition makes the right identity law hold for `Free3`. One can write a function for transforming a `Free4` program into a `Free3` program:

```
def free4toFree3[F[_], A]: Free4[F, A] => Free3[F, A] = {
  case fmap @ FMap(p)   => Bind(p)(fmap.f andThen Free3.pure) // Replace FMap by Bind.
  case Val(a)           => Val(a)                             // All other cases are unchanged.
  case bind@Bind(p)     => Bind(p)(bind.f)
  case Op(op)           => Op(op)
}
```

[5] The `Free1` encoding cannot be derived from the others because `Free1` *requires* that the effect constructor `F` be a functor with a `map` method, while the other encodings work for any `F`. Later in this chapter, we will prove that `Free1` is equivalent to `Free2` when `F` is a functor. We will also show more examples of free typeclass constructions that require another typeclass.

An inverse transformation (from `Free3` to `Free4`) can be implemented by translating all case classes identically. That code, however, will never create values of type `FMap` since `Free3` does not have that case class. It is impossible to convert any values of type `Bind` to `FMap` because it is impossible to determine whether a given function of type `A => Free3[B]` is expressible in the form `(f andThen pure)` with some `f: A => B`.

After running the corresponding `Free3` and `Free4` DSL programs, the results will be the same:

```
runFree4(runner)(free4program) == runFree3(runner)(free4toFree3(free4program))
```

To verify this ***

Let us now derive `Free2` from `Free3`. ***

13.2.4 Types with existential quantifiers

The previous sections performed all derivations using the Scala code syntax rather than the code notation. The main reason is that the constructions `Free2`, `Free3`, and `Free4` involve a special use of type parameters that is not supported by the type and code notations shown in this book until now. We need to introduce a new feature into the notation, namely a type with an **existential quantifier**, which is denoted by $\exists$ in the type notation.

To clarify the usage of the symbol $\exists$, we begin by recalling the definition of the type `MonadDSL`:

```
sealed trait MonadDSL[F[_], T] {
  def flatMap[A](...) = ...
  def map[A](f: T => A): MonadDSL[F, A] = Bind(this)(f andThen MonadDSL.pure)
}
final case class Val[F[_], T](t: T)                                        extends MonadDSL[F, T]
final case class Op[F[_], T](f: F[T])                                      extends MonadDSL[F, T]
final case class Bind[F[_], T, A](p: MonadDSL[F, A])(g: A => MonadDSL[F, T]) extends MonadDSL[F, T]
```

The case class `Bind[F, T, A]` uses the type parameter `A` in a special way. Whenever a value of type `Bind[F, T, A]` is created, the type parameter `A` must be set to a specific type. However, the result will be a value of type `MonadDSL[F, T]`, which is *not* parameterized by `A`. The extra type parameter (`A`) is hidden inside the definition of `Bind[F, T, A]`.

To see this in a code example, let us apply `map` to a given value `p` of type `MonadDSL[F, Int]`:

```
val p: MonadDSL[F, Int] = ...
val q: MonadDSL[F, String] = p.map { i: Int => s"value = $i" } // Creates a Bind[F, String, Int].
```

The value `q` is declared to be of type `MonadDSL[F, T]` with `T = String`. However, the actual data structure in `q` contains a value of type `Bind[F, T, A]` with a type parameter `A` chosen as `A = Int`. The type declaration `Bind[F, T, A](...) extends MonadDSL[F, T]` hides the type parameter `A` from the rest of the code, although that type parameter will be needed when constructing a value `q` of type `Bind`. This usage of a type parameter is called an **existentially quantified** type and is denoted by $\exists A$.

The other type parameters (`F`, `T`) in `Bind[F, T, A]` are not existentially quantified because they appear both in the case class and in the parent trait. So, we write the definition of `Bind[F, T, A]` as:

$$\exists A.\,\text{Bind}^{F,T,A} \stackrel{\text{def}}{=} \exists A.\,\text{MonadDSL}^{F,A} \times (A \to \text{MonadDSL}^{F,T}) \quad .$$

Adding the type notation for the case classes `Val` and `Op`, we get a definition of the type `MonadDSL`:

$$\begin{aligned}\text{MonadDSL}^{F,T} &\stackrel{\text{def}}{=} \text{Val}^{F,T} + \text{Op}^{F,T} + \exists A.\,\text{Bind}^{F,T,A} \\ &= T + F^T + \exists A.\,\text{MonadDSL}^{F,A} \times (A \to \text{MonadDSL}^{F,T}) \quad .\end{aligned}$$

To interpret the name "existential", we may imagine that a type parameter `A` "still exists" inside the value `q: MonadDSL[F, T]`, since `q` must have been created as `Bind[F, T, A]` with a specific (somehow chosen) type `A`. However, we cannot inspect or obtain the type `A` if we only have the value `q`. No pattern-matching or other operations on `q` can determine whether the type `A` is, say, `Int` or `String`. We can use the value `q` in our code as long as we treat `A` as an unknown, arbitrary, but fixed type.

Scala gives two ways of defining types with existential quantifiers: via case classes with an extra type parameter, or via a class with a type member. To illustrate these techniques, consider a type expression $\exists C.\, L^{A,B,C}$ where L is a type constructor with type parameters A, B, C and an existential quantifier on C. Note that the type parameter C is **bound** by the quantifier and will not be visible outside the type expression $\exists C.\, L^{A,B,C}$. The type parameters A and B remain free in $\exists C.\, L^{A,B,C}$.

An example of such a type expression is $\exists C.\, A \times C \times (C \to B)$. We will now implement this type via case classes and via type members. The code using case classes may look like this:

```
sealed trait L[A, B]
final case class L1[A, B, C](a: A, c: C, p: C => B) extends L[A, B]
```

The data inside the case class may use the types A, B, C but the outside code will only see A and B:

```
val q1: L[Int, Int] = L1[Int, Int, String](1, "abc", _.length)
```

Nevertheless, we may write code that uses data of type C without knowing what C is:

```
def toInt: L[Int, Int] => Int = {
  case L1(a, c, p) => a + p(c)
}

scala> toInt(q1)
res0: Int = 4
```

In this code, the intermediate result `p(c)` uses data of unknown type `C` but does not try to inspect that type.

To implement the type $\exists C.\, A \times C \times (C \to B)$ via a trait with a type member, we write:

```
trait L2[A, B] {
  type C
  val a: A
  val c: C
  val p: C => B
}
val q2: L2[Int, Int] = new L2[Int, Int] {
  type C = String
  val a: Int = 1
  val c: String = "abc"
  val p: String => Int = _.length
}
def toInt(q: L2[Int, Int]): Int = q.a + q.p(q.c)

scala> toInt(q2)
res1: Int = 4
```

The code is more verbose but has equivalent functionality to the code that uses a case class. Each time we create a value of type `L[A, B]` or `L2[A, B]`, we will have to specify some type for `C`. That type will become hidden. Trying to access the type `C` from external code will be a type error:

```
scala> val x: q2.C = "abc"        // The type q2.C is String but the code may not use that knowledge.
                   ^
error: type mismatch;
found   : String("abc")
required: q2.C
```

***

With this notation, the four free monad constructions are written like this:

$$\begin{aligned}
\text{free monad on a functor } F: &\quad \text{Free}_1^{F,T} \stackrel{\text{def}}{=} A + F^{\text{Free}_1^{F,T}} \\
\text{free monad on } F \text{ in a reduced encoding}: &\quad \text{Free}_2^{F,T} \stackrel{\text{def}}{=} T + \exists A.\, F^A \times (A \to \text{Free}_2^{F,T}) \\
\text{free monad on } F \text{ in a reduced encoding}: &\quad \text{Free}_3^{F,T} \stackrel{\text{def}}{=} T + F^T + \exists A.\, \text{Free}_3^{F,A} \times (A \to \text{Free}_3^{F,T}) \\
\text{free monad on } F \text{ in a raw tree encoding}: &\quad \text{Free}_4^{F,A} \stackrel{\text{def}}{=} A + F^A + \exists B.\, \text{Free}_4^{F,B} \times (B \to A) \\
&\qquad + \exists B.\, \text{Free}_4^{F,B} \times (B \to \text{Free}_4^{F,A})
\end{aligned}$$

13.3 Free constructions for other typeclasses

let us look at the types used by `MonadDSL[F, A]`. For brevity, we will use the type notation:

$$\text{MonadDSL}^{F,A} \stackrel{\text{def}}{=} A + F^A + \exists B.\, F^B \times (B \to F^A) \quad .***$$

So, the ***

Can combine two or more DSLs in a disjunction: $\text{DSL}^{F+G+H,A}$

13.3.1 Free constructions for inductive typeclasses

13.3.2 Free constructions that assume other typeclasses

13.4 Laws of free constructions

13.5 Slides

Worked example I: Free semigroup Implement a free semigroup `FSIS` generated by two types `Int` and `String`

A value of `FSIS` can be an `Int`; it can also be a `String`

If `x`, `y` are of type `FSIS` then so is `x |+| y`

```
sealed trait FSIS // tree encoding: full expression tree
case class Wrap1(x: Int) extends FSIS
case class Wrap2(x: String) extends FSIS
case class Comb(x: FSIS, y: FSIS) extends FSIS
```

Short type notation: $\text{FSIS} \stackrel{\text{def}}{=} \text{Int} + \text{String} + \text{FSIS} \times \text{FSIS}$

For a semigroup S and given $\text{Int} \to S$ and $\text{String} \to S$, map $\text{FSIS} \to S$

Simplify and generalize this construction by setting $Z = \text{Int} + \text{String}$

The tree encoding is $\text{FS}^Z \stackrel{\text{def}}{=} Z + \text{FS}^Z \times \text{FS}^Z$

```
def |+|(x: FS[Z], y: FS[Z]): FS[Z] = Comb(x, y)
def run[S: Semigroup, Z](extract: Z → S): FS[Z] → S = {
 case Wrap(z) → extract(z)
 case Comb(x, y) → run(extract)(x) |+| run(extract)(y)
} // Semigroup laws will hold after applying run().
```

The reduced encoding is $\text{FSR}^Z \stackrel{\text{def}}{=} Z \times \text{List}^Z$ (non-empty list of Z's)

`x |+| y` requires concatenating the lists, but `run()` is faster

Worked example II: Free monoid Implement a free monoid `FM[Z]` generated by type `Z`

A value of `FM[Z]` can be the empty value; it can also be a `Z`

If `x`, `y` are of type `FM[Z]` then so is `x |+| y`

```
sealed trait FM[Z] // tree encoding
case class Empty[Z]() extends FM[Z]
case class Wrap[Z](z: Z) extends FM[Z]
case class Comb[Z](x: FM[Z], y: FM[Z]) extends FM[Z]
```

Short type notation: $\text{FM}^Z \stackrel{\text{def}}{=} 1 + Z + \text{FM}^Z \times \text{FM}^Z$

For a monoid M and given $Z \to M$, map $\text{FM}^Z \to M$

```
def |+|(x: FM[Z], y: FM[Z]): FM[Z] = Comb(x, y)
def run[M: Monoid, Z](extract: Z → M): FM[Z] → M = {
 case Empty() → Monoid[M].empty
 case Wrap(z) → extract(z)
 case Comb(x, y) → run(extract)(x) |+| run(extract)(y)
} // Monoid laws will hold after applying run().
```

The reduced encoding is $\text{FMR}^Z \stackrel{\text{def}}{=} \text{List}^Z$ (list of Z's)

Implementing `|+|` requires concatenating the lists

Reduced encoding and tree encoding give identical results after `run()`

Mapping a free semigroup to different targets What if we interpret FS^X into *another* free semigroup?

Given $Y \to Z$, can we map $\text{FS}^Y \to \text{FS}^Z$?

Need to map $\text{FS}^Y \stackrel{\text{def}}{=} Y + \text{FS}^Y \times \text{FS}^Y \to Z + \text{FS}^Z \times \text{FS}^Z$

This is straightforward since FS^X is a functor in X:

```
def fmap[Y, Z](f: Y → Z): FS[Y] → FS[Z] = {
 case Wrap(y) → Wrap(f(y))
 case Comb(a, b) → Comb(fmap(f)(a), fmap(f)(b))
}
```

Now we can use `run` to interpret $\text{FS}^X \to \text{FS}^Y \to \text{FS}^Z \to S$, etc.

Functor laws hold for FS^X, so `fmap` is composable as usual

The "interpreter" commutes with `fmap` as well (naturality law):

$$\begin{tikzcd} & \text{FS}^Y \arrow[dr, "\text{run}^S\, g : Y \to S"] & \\ \text{FS}^X \arrow[ur, "\text{fmap}\, f : X \to Y"] \arrow[rr, "\text{run}^S\,(f \,⨾\, g) : X \to S"'] & & S \end{tikzcd}$$

Combine two free semigroups: FS^{X+Y}; inject parts: $\text{FS}^X \to \text{FS}^{X+Y}$

Church encoding I: Motivation Multiple target semigroups S_i require many "extractors" $\text{ex}_i : Z \to S_i$

Refactor extractors ex_i into evidence of a typeclass constraint on S_i

```
// Typeclass ExZ[S] has a single method, extract: Z → S.
implicit val exZ: ExZ[MySemigroup] = { z → ... }
def run[S: ExZ : Semigroup](fs: FS[Z]): S = fs match {
 case Wrap(z) → implicitly[ExZ[S]].extract(z)
 case Comb(x, y) → run(x) |+| run(y)
}
```

`run()` replaces case classes by fixed functions parameterized by `S: ExZ`; instead we can represent `FS[Z]` directly by such functions, for example:

```
def wrap[S: ExZ](z: Z): S = implicitly[ExZ[S]].extract(z)
def x[S: ExZ : Semigroup]: S = wrap(1) |+| wrap(2)
```

The type of `x` is $\forall S.\,(Z \to S) \times (S \times S \to S) \to S$; an equivalent type is

$$\forall S.\,((Z + S \times S) \to S) \to S$$

This is the "**Church encoding**" (of the free semigroup over Z)

The Church encoding is based on the theorem $A \cong \forall X.\,(A \to X) \to X$

this *resembles* the type of the continuation monad, $(A \to R) \to R$

but $\forall X$ makes the function fully generic, like a natural transformation

Church encoding II: Disjunction types Consider the Church encoding for the disjunction type $P + Q$

The encoding is $\forall X.\,(P + Q \to X) \to X \cong \forall X.\,(P \to X) \to (Q \to X) \to X$

```
trait Disj[P, Q] { def run[X](cp: P → X)(cq: Q → X): X }
```

Define some values of this type:

```
def left[P, Q](p: P) = new Disj[P, Q] {
 def run[X](cp: P → X)(cq: Q → X): X = cp(p)
}
```

Now we can implement the analog of the `case` expression simply as

```
val result = disj.run {p → ...} {q → ...}
```

This works in programming languages that have no disjunction types

General recipe for implementing the Church encoding:

```
trait Blah { def run[X](cont: ... → X): X }
```

For convenience, define a type class `Ex` describing the inner function:

```
trait Ex[X] { def cp: P → X; def cq: Q → X }
```

Different methods of this class return `X`; convenient with disjunctions

Church-encoded types have to be "run" for pattern matching to work on the results

Church encoding III: How it works Why is the type $\text{Ch}^A \stackrel{\text{def}}{=} \forall X.\,(A \to X) \to X$ equivalent to the type A?

```
trait Ch[A] { def run[X](cont: A → X): X }
```

- If we have a value of A, we can get a Ch^A

```
def a2c[A](a: A): Ch[A] = new Ch[A] {
  def run[X](cont: A → X): X = cont(a)
}
```

$$\begin{tikzcd} \text{id} : (A \to A) \arrow[r, "\text{ch.run}^A"] \arrow[d, "\text{fmap}_{\text{Reader}_A}(f)"] & A \arrow[d, "f"] \\ f : (A \to X) \arrow[r, "\text{ch.run}^X"'] & X \end{tikzcd}$$

- If we have a $\text{ch} : \text{Ch}^A$, we can get an $a : A$

```
def c2a[A](ch: Ch[A]): A = ch.run[A](a→a)
```

The functions `a2c` and `c2a` are inverses of each other

To implement a value $\text{ch}^{:\text{Ch}^A}$, we must compute an $x^{:X}$ given $f^{:A\to X}$, for *any* X, which *requires* having a value $a^{:A}$ available

To show that `ch = a2c(c2a(ch))`, apply both sides to an `f: A→X` and get `ch.run(f) = a2c(c2a(ch)).run(f) = f(c2a(ch)) = f(ch.run(a→a))`

This is naturality of `ch.run` as a transformation between `Reader` and `Id`

Naturality of `ch.run` follows from parametricity of its code

It is straightforward to compute `c2a(a2c(a)) = identity(a) = a`

Church encoding satisfies laws: it is built up from parts of `run` method

Worked example III: Free functor I The `Functor` type class has one method, `fmap`: $(Z \to A) \to F^Z \to F^A$

The tree encoding of a free functor over $F^\bullet$ needs two case classes:

```
sealed trait FF[F[_], A]
case class Wrap[F[_], A](fa: F[A]) extends FF[F, A]
case class Fmap[F[_], A, Z](f: Z => A)(ffz: FF[F, Z]) extends FF[F, A]
```

The constructor `Fmap` has an extra type parameter Z, which is "hidden"

Consider a simple example of this:

```
sealed trait Q[A]; case class QZ[A, Z](a: A, z: Z) extends Q[A]
```

Need to use specific type Z when constructing a value of `Q[A]`, e.g.,

```
val q: Q[Int] = QZ[Int, String](123, "abc")
```

The type Z is hidden inside $q : Q^{\text{Int}}$; all we know is that Z "exists"

Type notation for this: $Q^A \stackrel{\text{def}}{=} \exists Z.A \times Z$

The existential quantifier applies to the "hidden" type parameter

The constructor `QZ` has type $\exists Z.\,(A \times Z \to Q^A)$

It is not $\forall Z$ because a specific Z is used when building up a value

The code does not show $\exists Z$ explicitly! We need to keep track of that

Encoding with an existential type: How it works Show that $P^A \stackrel{\text{def}}{=} \exists Z.Z \times (Z \to A) \cong A$

```
sealed trait P[A]; case class PZ[A, Z](z: Z, f: Z → A) extends P[A]
```

How to construct a value of type P^A for a given A?

Have a function $Z \to A$ and a Z, construct $Z \times (Z \to A)$

Particular case: $Z \stackrel{\text{def}}{=} A$, have $a : A$ and build $a \times \text{id}^{:A\to A}$

```
def a2p[A](a: A): P[A] = PZ[A, A](a, identity)
```

Cannot extract Z out of P^A – the type Z is hidden

Can extract A out of P^A – do not need to know Z

```
def p2a[A]: P[A] → A = { case PZ(z, f) → f(z) }
```

Cannot transform P^A into anything else other than A

A value of type P^A is observable only via `p2a`

Therefore the functions `a2p` and `p2a` are "observational" inverses (i.e. we need to use `p2a` in order to compare values of type P^A)

If $F^\bullet$ is a functor then $Q^A \stackrel{\text{def}}{=} \exists Z.F^Z \times (Z \to A) \cong F^A$

A value of Q^A can be observed only by extracting an F^A from it

Can define `f2q` and `q2f` and show that they are observational inverses

Worked example III: Free functor II Tree encoding of `FF` has type $\text{FF}^{F^\bullet,A} \stackrel{\text{def}}{=} F^A + \exists Z.\text{FF}^{F^\bullet,Z} \times (Z \to A)$

Derivation of the reduced encoding:

A value of type $\text{FF}^{F^\bullet,A}$ must be of the form

$$\exists Z_1.\exists Z_2...\exists Z_n.F^{Z_n} \times (Z_n \to Z_{n-1}) \times ... \times (Z_2 \to Z_1) \times (Z_1 \to A)$$

The functions $Z_1 \to A$, $Z_2 \to Z_1$, etc., must be composed associatively

The equivalent type is $\exists Z_n.F^{Z_n} \times (Z_n \to A)$

Reduced encoding: $\text{FreeF}^{F^\bullet,A} \stackrel{\text{def}}{=} \exists Z.F^Z \times (Z \to A)$

Substituted F^Z instead of $\text{FreeF}^{F^\bullet,Z}$ and eliminated the case F^A

The reduced encoding is non-recursive

Requires a proof that this encoding is equivalent to the tree encoding

If $F^\bullet$ is already a functor, can show $F^A \cong \exists Z.F^Z \times (Z \to A)$

Church encoding (starting from the tree encoding): $\text{FreeF}^{F^\bullet,A} \stackrel{\text{def}}{=} \forall P^\bullet.\left(\forall C.\left(F^C + \exists Z.P^Z \times (Z \to C)\right) \rightsquigarrow P^C\right) \to P^A$

The structure of the type expression: $\forall P^\bullet.\left(\forall C.(...)^C \rightsquigarrow P^C\right) \to P^A$

Cannot move $\forall C$ or $\exists Z$ to the outside of the type expression!

Church encoding IV: Recursive types and type constructors Consider the recursive type $P \stackrel{\text{def}}{=} Z + P \times P$ (tree with Z-valued leaves)

The Church encoding is $\forall X.((Z + X \times X) \to X) \to X$

This is *non-recursive*: the inductive use of P is replaced by X

Generalize to recursive type $P \stackrel{\text{def}}{=} S^P$ where $S^\bullet$ is a "induction functor":

The Church encoding of P is $\forall X.(S^X \to X) \to X$

Church encoding of recursive types is non-recursive

Example: Church encoding of `List[Int]`

Church encoding of a type constructor $P^\bullet$:

Notation: $P^\bullet$ is a type function; Scala syntax is `P[_]`

The Church encoding is $\text{Ch}^{P^\bullet,A} = \forall F^\bullet.(\forall X.P^X \to F^X) \to F^A$

Note: $\forall X.P^X \to F^X$ or $P^\bullet \rightsquigarrow F^\bullet$ resembles a natural transformation

Except that $P^\bullet$ and $F^\bullet$ are not necessarily functors, so no naturality law

Example: Church encoding of `Option[_]`

Church encoding of a *recursively* defined type constructor $P^\bullet$:

Definition: $P^A \stackrel{\text{def}}{=} S^{P^\bullet,A}$ where $S^{P^\bullet,A}$ describes the "induction principle"

Notation: $S^{\bullet,A}$ is a higher-order type function; Scala syntax: `S[_[_],A]`

Example: $\text{List}^A \stackrel{\text{def}}{=} 1 + A \times \text{List}^A \stackrel{\text{def}}{=} S^{\text{List}^\bullet,A}$ where $S^{P^\bullet,A} \stackrel{\text{def}}{=} 1 + A \times P^A$

The Church encoding of P^A is $\text{Ch}^{P^\bullet,A} = \forall F^\bullet.(S^{F^\bullet} \rightsquigarrow F^\bullet) \to F^A$

The Church encoding of `List[_]` is non-recursive

Details: Why Church encoding of a free semigroup is a semigroup - it's not obvious

FS = forall S. (Z => S) × (S × S => S) => S is a semigroup. We need to define the binary operation |+| on values of type FS. A value f of type FS is a function with a type parameter, that we can use as f[S](e, c) to compute a value of any given type S from arguments e : Z => S and c: S × S => S. Scala code for f will be

```
def f[S](empty: Z => S, combine: (S, S) => S): S = ???
```

So, given f and g of this type, we need to somehow define a new function h = f |+| g also of the same type. Begin to write code for that function:

```
def h[S](empty: Z => S, combine: (S, S) => S): S = ???
```

The free semigroup in the tree encoding defines the binary operation as a formal operation that does not compute anything. In the Church encoding, however, we have the binary operation as the argument "combine" of h, and so we should call that function. So we use it:

def h[S](empty: Z => S, combine: (S, S) => S): S = combine(???, ???)

We need to fill the typed holes ??? of type S. It is clear that we should use f and g somehow. We can use f and g simply by calling those functions on the arguments 'empty' and 'combine'. Since f and g both have a universally quantified type parameter, we can just use the given type S for them.

def h[S](empty: Z => S, combine: (S, S) => S): S = combine(f[S](empty, combine), g[S](empty, combine))

The types match, and we have used both functions f and g in a way that is intuitively correct. We have preserved information. So, this is likely the correct implementation. It remains to verify the associativity law. To do that, we need to assume that 'combine' is associative for the actual type S on which we use the Church encoding (i.e. a non-free, lawful semigroup)

To show equivalence between FSZ and Chz, write code for the type ChZ and the two directions of the isomorphism,

```
trait FSCh[Z] { def run[S](empty: Z => S, combine: (S, S) => S): S }
def fsz2ch[Z](fsz: NEList[Z]): FSCh[Z] = ???
def ch2fsz[Z](ch: FSCh[Z]): NEList[Z] = ???
```

Church encoding V: Type classes Look at the Church encoding of the free semigroup:

$$\text{ChFS}^Z \stackrel{\text{def}}{=} \forall X.\,(Z \to X) \times (X \times X \to X) \to X$$

If X is constrained to the `Semigroup` typeclass, we will already have a value $X \times X \to X$, so we can omit it: $\text{ChFS}^Z = \forall X^{:\text{Semigroup}}.\,(Z \to X) \to X$

The "induction functor" for "semigroup over Z" is $\text{SemiG}^X \stackrel{\text{def}}{=} Z + X \times X$

So the Church encoding is $\forall X.(\text{SemiG}^X \to X) \to X$

Generalize to arbitrary type classes:

Type class C is defined by its operations $C^X \to X$ (with a suitable $C^\bullet$)

call $C^\bullet$ the **structure functor** of the inductive typeclass C

Tree encoding of "free C over Z" is recursive, $\text{FreeC}^Z \stackrel{\text{def}}{=} Z + C^{\text{FreeC}^Z}$

Church encoding is $\text{FreeC}^Z \stackrel{\text{def}}{=} \forall X.\,(Z + C^X \to X) \to X$

Equivalently, $\text{FreeC}^Z \stackrel{\text{def}}{=} \forall X^{:C}.\,(Z \to X) \to X$

Laws of the typeclass are satisfied automatically after "running"

Works similarly for type constructors: operations $C^{P^\bullet, A} \to P^A$

Free typeclass C over $F^\bullet$ is $\text{FreeC}^{F^\bullet, A} \stackrel{\text{def}}{=} \forall P^{\bullet:C}.\,(F^\bullet \rightsquigarrow P^\bullet) \to P^A$

Properties of free type constructions Generalizing from our examples so far:

We "enriched" Z to a monoid FM^Z, and F^A to a monad $\text{DSL}^{F,A}$

The "enrichment" adds case classes representing the needed operations

Works for a generating type Z and for a generating type constructor F^A

Obtain a **free type construction**, which performs no computations

FM^Z wraps Z in "just enough" stuff to make it look like a monoid

$\text{FreeF}^{F^\bullet, A}$ wraps F^A in "just enough" stuff to make it look like a functor

A value of a free construction can be "run" to yield non-free values

Questions:

Can we construct a free typeclass C over any type constructor F^A?

Yes, with typeclasses: (contra)functor, filterable, monad, applicative

Which of the possible encodings to use?

Tree encoding, reduced encodings, Church encoding

What are the laws for the $\text{FreeC}^{F,A}$ – "free instance of C over F"?

For all $F^\bullet$, must have `wrap[A]` : $F^A \to \text{FreeC}^{F,A}$ or $F^\bullet \rightsquigarrow \text{FreeC}^{F,\bullet}$

For all $M^\bullet : C$, must have `run` : $(F^\bullet \rightsquigarrow M^\bullet) \to \text{FreeC}^{F,\bullet} \rightsquigarrow M^\bullet$

The laws of typeclass C must hold after interpreting into an $M^\bullet : C$

Given any `t`: $F^\bullet \rightsquigarrow G^\bullet$, must have `fmap(t)`: $\text{FreeC}^{F,\bullet} \rightsquigarrow \text{FreeC}^{G,\bullet}$

Recipes for encoding free typeclass instances Build a free instance of typeclass C over $F^\bullet$, as a type constructor $P^\bullet$

The typeclass C can be functor, contrafunctor, monad, etc.

Assume that C has methods m_1, m_2, ..., with type signatures $m_1 : Q_1^{P^\bullet,A} \to P^A$, $m_2 : Q_2^{P^\bullet,A} \to P^A$, etc., where $Q_i^{P^\bullet,A}$ are covariant in $P^\bullet$

Inductive typeclass is defined via a methods functor, $S^{P^\bullet} \rightsquigarrow P^\bullet$

The tree encoded FC^A is a disjunction defined recursively by

$$\text{FC}^A \stackrel{\text{def}}{=} F^A + Q_1^{\text{FC}^\bullet,A} + Q_2^{\text{FC}^\bullet,A} + ...$$

```
sealed trait FC[A]; case class Wrap[A](fa:  F[A]) extends FC[A]
case class Q1[A](...)  extends FC[A]
case class Q2[A](...)  extends FC[A]; ...
```

Any type parameters within Q_i are then existentially quantified

`run()` maps $F^\bullet \rightsquigarrow M^\bullet$ in the disjunction and recursively for other parts

Derive a reduced encoding via reasoning about possible values of FC^A and by taking into account the laws of the typeclass C

A Church encoding can use the tree encoding or the reduced encoding

Church encoding is "automatically reduced", but performance may differ

Properties of inductive typeclasses If a typeclass C is inductive with methods $C^X \to X$ then:

A free instance of C over Z can be tree-encoded as $\text{FreeC}^Z \stackrel{\text{def}}{=} Z + C^{\text{FreeC}^Z}$

All inductive typeclasses have free instances, FreeC^Z

If $P^{:C}$ and $Q^{:C}$ then $P \times Q$ and $Z \to P$ also belong to typeclass C

but not necessarily $P + Q$ or $Z \times P$

Proof: can implement $(C^P \to P) \times (C^Q \to Q) \to C^{P\times Q} \to P \times Q$ and $(C^P \to P) \to C^{Z\to P} \to Z \to P$, but cannot implement $(...) \to P + Q$

Analogous properties hold for type constructor typeclasses

Methods described as $C^{F^\bullet,A} \to F^A$ with type constructor parameter $F^\bullet$

What typeclasses *cannot* be tree-encoded (or have no "free" instances)?

Any typeclass with a method *not ultimately returning* a value of P^A

Example: a typeclass with methods $\text{pt} : A \to P^A$ and $\text{ex} : P^A \to A$

Such typeclasses are not inductive

Typeclasses with methods of the form $P^A \to ...$ are **co-inductive**

Worked example IV: Free contrafunctor Method $\text{contramap} : C^A \times (B \to A) \to C^B$

Tree encoding: $\text{FreeCF}^{F^\bullet,B} \stackrel{\text{def}}{=} F^B + \exists A.\text{FreeCF}^{F^\bullet,A} \times (B \to A)$

Reduced encoding: $\text{FreeCF}^{F^\bullet,B} \stackrel{\text{def}}{=} \exists A.F^A \times (B \to A)$

A value of type $\text{FreeCF}^{F^\bullet,B}$ must be of the form

$$\exists Z_1.\exists Z_2...\exists Z_n.F^{Z_1} \times (B \to Z_n) \times (Z_n \to Z_{n-1}) \times ... \times (Z_2 \to Z_1)$$

The functions $B \to Z_n$, $Z_n \to Z_{n-1}$, etc., are composed associatively

The equivalent type is $\exists Z_1.F^{Z_1} \times (B \to Z_1)$

The reduced encoding is non-recursive

Example: $F^A \stackrel{\text{def}}{=} A$, "interpret" into the contrafunctor $C^A \stackrel{\text{def}}{=} A \to \text{String}$

```
def prefixLog[A](p:  A): A → String = a → p.toString + a.toString
```

If $F^\bullet$ is already a contrafunctor then $\text{FreeCF}^{F^\bullet,A} \cong F^A$

Worked example V: Free pointed functor Over an arbitrary type constructor $F^\bullet$:

Pointed functor methods $\text{pt} : A \to P^A$ and $\text{map} : P^A \times (A \to B) \to P^B$

Tree encoding: $\text{FreeP}^{F^\bullet,A} \stackrel{\text{def}}{=} A + F^A + \exists Z.\text{FreeP}^{F^\bullet,Z} \times (Z \to A)$

Derivation of the reduced encoding:

The tree encoding of a value $\text{FreeP}^{F^\bullet,A}$ is either

$$\exists Z_1.\exists Z_2...\exists Z_n.F^{Z_n} \times (Z_n \to Z_{n-1}) \times ... \times (Z_2 \to Z_1) \times (Z_1 \to A)$$

or

$$\exists Z_1.\exists Z_2...\exists Z_n.Z_n \times (Z_n \to Z_{n-1}) \times ... \times (Z_2 \to Z_1) \times (Z_1 \to A)$$

Compose all functions by associativity; one function $Z_n \to A$ remains
The case $\exists Z_n . Z_n \times (Z_n \to A)$ is equivalent to just A
Reduced encoding: $\text{FreeP}^{F^\bullet, A} \stackrel{\text{def}}{=} A + \exists Z . F^Z \times (Z \to A)$, non-recursive
This reuses the free functor as $\text{FreeP}^{F^\bullet, A} = A + \text{FreeF}^{F^\bullet, A}$
If the type constructor $F^\bullet$ is *already* a functor, $\text{FreeF}^{F^\bullet, A} \cong F^A$ and so:
Free pointed functor over a functor $F^\bullet$ is simplified: $A + F^A$
If $F^\bullet$ is already a pointed functor, need not use the free construction
If we do, we will have $\text{FreeP}^{F^\bullet, A} \ncong F^A$
only functors and contrafunctors do not change under "free"

Worked example VI: Free filterable functor (See Chapter 6.) Methods:

$$\text{map} : F^A \to (A \to B) \to F^B$$
$$\text{mapOpt} : F^A \to (A \to 1 + B) \to F^B$$

We can recover `map` from `mapOpt`, so we keep only `mapOpt`
Tree encoding: $\text{FreeFi}^{F^\bullet, A} \stackrel{\text{def}}{=} F^A + \exists Z . \text{FreeFi}^{F^\bullet, Z} \times (Z \to 1 + A)$
If $F^\bullet$ is already a functor, can simplify the tree encoding using the identity $\exists Z . P^Z \times (Z \to 1 + A) \cong P^A$ and obtain $\text{FreeFi}^{F^\bullet, A} \stackrel{\text{def}}{=} F^A + \text{FreeFi}^{F^\bullet, 1+A}$, which is equivalent to $\text{FreeFi}^{F^\bullet, A} = F^A + F^{1+A} + F^{1+1+A} + \ldots$
Reduced encoding: $\text{FreeFi}^{F^\bullet, A} \stackrel{\text{def}}{=} \exists Z . F^Z \times (Z \to 1 + A)$, non-recursive
Derivation: $\exists Z_1 \ldots \exists Z_n . F^{Z_n} \times (Z_n \to 1 + Z_{n-1}) \times \ldots \times (Z_1 \to 1 + A)$ is simplified using the laws of `mapOpt` and Kleisli composition, and yields $\exists Z_n . F^{Z_n} \times (Z_n \to 1 + A)$. Encode F^A as $\exists Z . F^Z \times (Z \to 0 + Z)$.
If $F^\bullet$ is already a functor, the reduced encoding is $\text{FreeFi}^{F^\bullet, A} = F^{1+A}$
Free filterable over a filterable functor $F^\bullet$ is not equivalent to $F^\bullet$
Free filterable contrafunctor is constructed in a similar way

Worked example VII: Free monad Methods:

$$\text{pure} : A \to F^A$$
$$\text{flatMap} : F^A \to (A \to F^B) \to F^B$$

Can recover `map` from `flatMap` and `pure`, so we keep only `flatMap`
Tree encoding: $\text{FreeM}^{F^\bullet, A} \stackrel{\text{def}}{=} F^A + A + \exists Z . \text{FreeM}^{F^\bullet, Z} \times (Z \to \text{FreeM}^{F^\bullet, A})$
Derivation of reduced encoding:
can simplify $A \times (A \to \text{FreeM}^{F^\bullet, B}) \cong \text{FreeM}^{F^\bullet, B}$
use associativity to replace $\text{FreeM}^A \times (A \to \text{FreeM}^B) \times (B \to \text{FreeM}^C)$ by $\text{FreeM}^A \times (A \to \text{FreeM}^B \times (B \to \text{FreeM}^C))$
therefore we can replace $\exists Z . \text{FreeM}^{F^\bullet, Z} \times \ldots$ by $\exists Z . F^Z \times \ldots$
Reduced encoding: $\text{FreeM}^{F^\bullet, A} \stackrel{\text{def}}{=} A + \exists Z . F^Z \times (Z \to \text{FreeM}^{F^\bullet, A})$
"**Final Tagless** style" means "Church encoding of free monad over $F^\bullet$"
Free monad over a functor $F^\bullet$ is $\text{FreeM}^{F^\bullet, A} \stackrel{\text{def}}{=} A + F^{\text{FreeM}^{F^\bullet, A}}$
Free monad $\text{FreeM}^{M^\bullet, \bullet}$ over a monad $M^\bullet$ is not equivalent to $M^\bullet$
Free monad over a pointed functor $F^\bullet$ is $\text{FreeM}^{F^\bullet, A} \stackrel{\text{def}}{=} F^A + F^{\text{FreeM}^{F^\bullet, A}}$
start from half-reduced encoding $F^A + \exists Z . F^Z \times (Z \to \text{FreeM}^{F^\bullet, A})$
replace the existential type by an equivalent type $F^{\text{FreeM}^{F^\bullet, A}}$

Worked example VIII: Free applicative functor Methods:

$$\text{pure} : A \to F^A$$
$$\text{ap} : F^A \to F^{A \to B} \to F^B$$

We can recover `map` from `ap` and `pure`, so we omit `map`
Tree encoding: $\text{FreeAp}^{F^\bullet, A} \stackrel{\text{def}}{=} F^A + A + \exists Z . \text{FreeAp}^{F^\bullet, Z} \times \text{FreeAp}^{F^\bullet, Z \to A}$

Reduced encoding: $\text{FreeAp}^{F^\bullet, A} \stackrel{\text{def}}{=} A + \exists Z. F^Z \times \text{FreeAp}^{F^\bullet, Z \to A}$

Derivation: a FreeAp^A is either $\exists Z_1 ... \exists Z_n. Z_1 \times \text{FreeAp}^{Z_1 \to Z_2} \times ...$ or $\exists Z_1 ... \exists Z_n. F^{Z_1} \times \text{FreeAp}^{Z_1 \to Z_2} \times ...$; encode $Z_1 \times \text{FreeAp}^{Z_1 \to Z_2}$ equivalently as $\text{FreeAp}^{Z_1 \to Z_2} \times ((Z_1 \to Z_2) \to Z_2)$ using the identity law; so the first FreeAp^Z is always F^A, or we have a pure value

Free applicative over a functor $F^\bullet$:

$$\text{FreeAp}^{F^\bullet, A} \stackrel{\text{def}}{=} A + \text{FreeZ}^{F^\bullet, A}$$
$$\text{FreeZ}^{F^\bullet, A} \stackrel{\text{def}}{=} F^A + \exists Z. F^Z \times \text{FreeZ}^{F^\bullet, Z \to A}$$

$\text{FreeZ}^{F^\bullet, \bullet}$ is the reduced encoding of "free zippable" (no `pure`)

$\text{FreeAp}^{F^\bullet, \bullet}$ over an applicative functor $F^\bullet$ is not equivalent to $F^\bullet$

Laws for free typeclass constructions Consider an inductive typeclass C with methods $C^A \to A$

Define a free instance of C over Z recursively, $\text{FreeC}^Z \stackrel{\text{def}}{=} Z + C^{\text{FreeC}^Z}$

FreeC^Z has an instance of C, i.e. we can implement $C^{\text{FreeC}^Z} \to \text{FreeC}^Z$

FreeC^Z is a C over Z; we can implement the functions $\text{fmap}^{\text{FreeC}} : (Y \to Z) \to \text{FreeC}^Y \to \text{FreeC}^Z$

$$\text{run}^P : (Z \to P) \to \text{FreeC}^Z \to P$$
$$\text{wrap} : Z \to \text{FreeC}^Z$$

$$\begin{tikzcd} \text{FreeC}^Y \arrow[d, "\text{fmap}\, f^{:Y \to Z}"'] \arrow[dr, "\text{run}(f \,⨾\, g)"] & \\ \text{FreeC}^Z \arrow[r, "\text{run}(g^{:Z \to P})"] & P \end{tikzcd}$$

Law 1: $\text{run}\,(\text{wrap}) = \text{id}$; law 2: $\text{fmap}\, f \,⨾\, \text{run}\, g = \text{run}\,(f \,⨾\, g)$ (naturality of `run`)

For any $P^{:C}, Q^{:C}, g^{:Z \to P}$, and a typeclass-preserving $f^{:P \to Q}$, we have

$$\text{run}^P\,(g) \,⨾\, f = \text{run}^Q\,(g \,⨾\, f) \qquad \text{– "universal property" of run}$$

$$\begin{tikzcd} \text{FreeC}^Z \arrow[d, "\text{run}^P\,(g^{:Z \to P})"'] \arrow[dr, "\text{run}^Q\,(g \,⨾\, f)"] & \\ P \arrow[r, "f^{:P \to Q}"] & Q \end{tikzcd} \qquad \begin{tikzcd} C^P \arrow[r, "\text{ops}_P"] \arrow[d, "\text{fmap}_S\, f"'] & P \arrow[d, "f"] \\ C^Q \arrow[r, "\text{ops}_Q"] & Q \end{tikzcd}$$

$f^{:P \to Q}$ **preserves typeclass** C if the diagram on the right commutes

Combining the generating constructors in a free typeclass Consider FreeC^Z for an inductive typeclass C with methods $C^X \to X$

We would like to combine generating constructors Z_1, Z_2, etc.

In a monadic DSL – combine different operations defined separately

Note: monads do not compose in general

To combine generators, use $\text{FreeC}^{Z_1 + Z_2}$; an "instance over Z_1 and Z_2"

but need to inject parts into disjunction, which is cumbersome

Church encoding makes this easier to manage:

$\text{FreeC}^Z \stackrel{\text{def}}{=} \forall X. (Z \to X) \times (C^X \to X) \to X$ and then

$$\text{FreeC}^{Z_1 + Z_2} \stackrel{\text{def}}{=} \forall X. (Z_1 \to X) \times (Z_2 \to X) \times (C^X \to X) \to X$$

Encode the functions $Z_i \to X$ via typeclasses `ExZ1`, `ExZ2`, etc., where typeclass `ExZ1` has method $Z_1 \to X$, etc.

Then

$$\text{FreeC}^{Z_1 + Z_2} = \forall X^{:E_{Z_1} :E_{Z_2}}. (C^X \to X) \to X$$

or equivalently

$$\text{FreeC}^{Z_1 + Z_2} = \forall X^{:C\ :E_{Z_1} :E_{Z_2}}. X$$

The code is easier to maintain

This works for all typeclasses C and any number of generators Z_i

Combining different free typeclasses To combine free instances of different typeclasses C_1 and C_2:

Option 1: use functor composition, $\text{FreeC}_{12}^{Z} \stackrel{\text{def}}{=} \text{FreeC}_1^{\text{FreeC}_2^{Z}}$
Order of composition matters!
Operations of C_2 need to be lifted into C_1
Works only for inductive typeclasses
Encodes $C_1^{C_2}$ but not $C_2^{C_1}$
Option 2: use disjunction of method functors, $C^X \stackrel{\text{def}}{=} C_1^X + C_2^X$, and build the free typeclass instance using C^X
Church encoding: $\text{FreeC}_{12}^{Z} \stackrel{\text{def}}{=} \forall X.\, (Z \to X) \times \left(C_1^X + C_2^X \to X\right) \to X$
Example 1: C_1 is functor, C_2 is contrafunctor
Interpret a free functor/contrafunctor into a profunctor
Example 2: C_1 is monad, C_2 is applicative functor
Interpret into a monad that has a non-standard `zip` implementation
Example: interpret into `Future` and convert `zip` into parallel execution
Each `zip` creates parallel branch, each `flatMap` creates sequential chain

13.5.1 Exercises

Exercise 13.5.1.1 The **interactive input-output** monad is defined recursively by:

```
sealed trait TIO[A]
final case class Pure[A](a: A) extends TIO[A]
final case class Read[A](read: P => TIO[A]) extends TIO[A]
final case class Write[A](output: Q, next: TIO[A]) extends TIO[A]
```

In the type notation, this is written as:

$$\text{TIO}^A \stackrel{\text{def}}{=} A + (P \to \text{TIO}^A) + Q \times \text{TIO}^A \quad ,$$

Here P and Q are fixed types. The monad `TIO` represents computations that may consume an input value of type P or produce an output value of type Q. Use the free monad construction to show that `TIO` is a lawful monad. Implement a monad instance for `TIO`.

Exercise 13.5.1.2 Implement a free semigroup on a type Z in the tree encoding and in the reduced encoding. Show that the semigroup laws hold for the reduced encoding but *not* for the tree encoding. Show that the laws hold for the tree encoding after interpreting into a lawful semigroup S.

Exercise 13.5.1.3 For a fixed monoid L, define a typeclass Mod_L (called "L-module"). Type P is an L-module if the monoid L "acts" on P via a function act: $L \to P \to P$, with laws $\text{act}\, x \,⨾\, \text{act}\, y = \text{act}\,(x \,⨾\, y)$ and $\text{act}\,(e_L) = \text{id}$. Show that Mod_L is an inductive typeclass. Implement a free L-module on a type Z.

Exercise 13.5.1.4 **(a)** Implement a monadic DSL with operations `put: A => Unit` and `get: Unit => A`. These operations should store and retrieve a state value of type `A`. Test on some example programs written in that DSL.

(b) Implement a monadic DSL with operations `put: A => Unit`, `get: Unit => Option[A]`, and `clear: Unit => Unit`. These operations should store and retrieve a state value of type `A`. Running `clear` should delete the state value. When there is no state value, `get` should return `None`. Test on some example programs.

Exercise 13.5.1.5 Implement the Church encoding of the type constructor $P^A \stackrel{\text{def}}{=} \text{Int} + A \times A$. For the resulting type constructor, implement a `Functor` instance.

Exercise 13.5.1.6 Describe the monoid type class via a method functor $C^\bullet$ (such that the monoid's operations are combined into the type $S^M \to M$). Using $S^\bullet$, implement the free monoid on a type Z in the Church encoding.

Exercise 13.5.1.7 Assuming that $F^\bullet$ is a functor, define $Q^A \stackrel{\text{def}}{=} \exists Z. F^Z \times (Z \to A)$ and implement `f2q`: $F^A \to Q^A$ and `q2f`: $Q^A \to F^A$. Show that these functions are natural transformations, and that they are inverses of each other "observationally", i.e., after applying `q2f` in order to compare values of Q^A.

Exercise 13.5.1.8 Prove the following type equivalences involving quantified types:

(a) $\forall A. A \times A \cong \mathbb{0}$.

(b) $\forall A. (A \times A \times A \to A) \cong \mathbb{1} + \mathbb{1} + \mathbb{1}$.

(c) $\exists Z. Z \cong \mathbb{1}$.

(d) $\exists Z. Z \times (A \to Z) \times (Z \to B) \cong B \times (A \to B)$.

Exercise 13.5.1.9* Prove the following type equivalences involving quantified type constructors (the types A, B, C, D are fixed):

(a) $\forall F^\bullet. F^A \cong \mathbb{0}$.

(b) $\forall F^\bullet. F^A \to B \cong B$.

(c) $\forall F^\bullet. F^A \to F^B \cong A \to B$.

(d) $\forall F^\bullet. \left(A \to F^B\right) \to C + F^D \cong C + A \times (B \to D)$.

(e) $\forall F^\bullet. \left(A \to F^B\right) \to C \to F^D \cong (C \to A) \times (C \times B \to D)$.

Exercise 13.5.1.10 Derive a reduced encoding for a free applicative functor on a pointed functor.

Exercise 13.5.1.11 Implement a "free pointed filterable" typeclass (combining pointed and filterable) on a type constructor $F^\bullet$ in the tree encoding. Derive a reduced encoding. Simplify these encodings when $F^\bullet$ is already a functor.

Corrections The slides say that the "universal property" of the runner is $\text{run}^P g \fatsemi f = \text{run}^Q (g \fatsemi f)$, however, this is not true; it is the right naturality property of $\text{run}^P : (Z \to P) \to \text{FreeC}^Z \to P$ with respect to the type parameter P. The universal property is $f = \text{wrap} \fatsemi \text{run}^P f$ for any $f : Z \to P$ and any type P that belongs to the typeclass C.

The "logarithm" $\text{Lg}(F^\bullet) \stackrel{\text{def}}{=} \forall A. F^A \to A$ is an operation with bizarre properties. Examples: $\forall A. (Z \to A) \to A \cong Z$, so $\text{Lg}(Z \to \bullet) = Z$. This might motivate the name "logarithm". But $\text{Lg}(F^\bullet + G^\bullet) = \text{Lg}(F) \times \text{Lg}(G)$, which resembles the distributive law for the *exponential* function rather than for the logarithm. Also, $\forall A. (Z \times A \times A) \to A \cong Z \times \mathbb{2}$, so $\text{Lg}(Z \times (\mathbb{2} \to \bullet)) = Z \times \mathbb{2}$. However, for a constant functor, $\text{Lg}(Z) = \mathbb{0}$. This shows that $\text{Lg}(F^\bullet \times G^\bullet) \not\cong \text{Lg}(F) \times \text{Lg}(G)$. We also have $\text{Lg}(\text{Opt}) = \mathbb{0}$.

13.6 Properties of free constructions

13.6.1 Free monad

The free monad on a functor F is defined by

$$\text{Free}^{F,A} \stackrel{\text{def}}{=} A + F^{\text{Free}^{F,A}} \quad .$$

It was shown in Statement 10.2.8.10 that $\text{Free}^{F,A}$ is a lawful monad for any functor F. We will now derive some further properties of the free monad construction.

The next statement shows that one can change the underlying functor F while preserving the free monad operations.

Statement 13.6.1.1 For any functor G and any natural transformation $\phi : F^A \to G^A$, the corresponding transformation $\psi(\phi) : \text{Free}^{F,A} \to \text{Free}^{G,A}$ defined by:

$$\psi(\phi) : \text{Free}^{F,A} \to \text{Free}^{G,A} \quad , \qquad \psi \stackrel{\text{def}}{=} \left\| \begin{array}{c|cc} & A & G^{\text{Free}^{G,A}} \\ \hline A & \text{id} & \mathbb{0} \\ F^{\text{Free}^{F,A}} & \mathbb{0} & \overline{\psi}^{\uparrow F} \fatsemi \phi \end{array} \right\|$$

is a monad morphism. In other words, the free monad Free^F is natural in the functor F.

Proof Since ϕ is fixed, we can write $\psi(\phi)$ as simply ψ for brevity. Denote $P \stackrel{\text{def}}{=} \text{Free}^F$ and $Q \stackrel{\text{def}}{=} \text{Free}^G$; we need to show that $\psi : P \rightsquigarrow Q$ is a monad morphism.

To verify the identity law:

$$\text{expect to equal } \text{pu}_Q : \quad \text{pu}_P \fatsemi \psi = \begin{array}{c|c|cc} & & A & F^{P^A} \\ \hline & A & \text{id} & \mathbb{0} \end{array} \fatsemi \begin{array}{c|c|cc} & & A & G^{Q^A} \\ \hline & A & \text{id} & \mathbb{0} \\ & F^{P^A} & \mathbb{0} & \overline{\psi}^{\uparrow F} \fatsemi \phi \end{array}$$

$$= \begin{array}{c|c|cc} & & A & G^{Q^A} \\ \hline & A & \text{id} & \mathbb{0} \end{array} = \text{pu}_Q \quad .$$

To verify the composition law, write the two sides separately:

$$\text{left-hand side :} \quad \text{ftn}_P \fatsemi \psi = \begin{array}{c|c|cc} & & A & F^{P^A} \\ \hline & A & \text{id} & \mathbb{0} \\ & F^{P^A} & \mathbb{0} & \text{id} \\ & F^{P^{P^A}} & \mathbb{0} & \overline{\text{ftn}}_P^{\uparrow F} \end{array} \fatsemi \begin{array}{c|c|cc} & & A & G^{Q^A} \\ \hline & A & \text{id} & \mathbb{0} \\ & F^{P^A} & \mathbb{0} & \overline{\psi}^{\uparrow F} \fatsemi \phi \end{array}$$

$$= \begin{array}{c|c|cc} & & A & G^{Q^A} \\ \hline & A & \text{id} & \mathbb{0} \\ & F^{P^A} & \mathbb{0} & \overline{\psi}^{\uparrow F} \fatsemi \phi \\ & F^{P^{P^A}} & \mathbb{0} & \overline{\text{ftn}}_P^{\uparrow F} \fatsemi \overline{\psi}^{\uparrow F} \fatsemi \phi \end{array} \quad ,$$

$$\text{right-hand side :} \quad \psi^{\uparrow P} \fatsemi \psi \fatsemi \text{ftn}_Q = \begin{array}{c|c|ccc} & & A & G^{Q^A} & F^{P^{Q^A}} \\ \hline & A & \text{id} & \mathbb{0} & \mathbb{0} \\ & F^{P^A} & \mathbb{0} & \psi^{\uparrow F} \fatsemi \phi & \mathbb{0} \\ & F^{P^{P^A}} & \mathbb{0} & \mathbb{0} & \overline{\psi}^{\uparrow P \uparrow F} \end{array} \fatsemi \begin{array}{c|c|ccc} & & A & G^{Q^A} & G^{Q^{Q^A}} \\ \hline & A & \text{id} & \mathbb{0} & \mathbb{0} \\ & G^{Q^A} & \mathbb{0} & \text{id} & \mathbb{0} \\ & F^{P^{Q^A}} & \mathbb{0} & \mathbb{0} & \overline{\psi}^{\uparrow F} \fatsemi \phi \end{array} \fatsemi \text{ftn}_Q$$

$$= \begin{array}{c|c|ccc} & & A & G^{Q^A} & G^{Q^{Q^A}} \\ \hline & A & \text{id} & \mathbb{0} & \mathbb{0} \\ & F^{P^A} & \mathbb{0} & \psi^{\uparrow F} \fatsemi \phi & \mathbb{0} \\ & F^{P^{P^A}} & \mathbb{0} & \mathbb{0} & \overline{\psi}^{\uparrow P \uparrow F} \fatsemi \overline{\psi}^{\uparrow F} \fatsemi \phi \end{array} \fatsemi \begin{array}{c|c|cc} & & A & G^{Q^A} \\ \hline & A & \text{id} & \mathbb{0} \\ & G^{Q^A} & \mathbb{0} & \text{id} \\ & G^{Q^{Q^A}} & \mathbb{0} & \overline{\text{ftn}}_Q^{\uparrow G} \end{array}$$

$$= \begin{array}{c|c|cc} & & A & G^{Q^A} \\ \hline & A & \text{id} & \mathbb{0} \\ & F^{P^A} & \mathbb{0} & \psi^{\uparrow F} \fatsemi \phi \\ & F^{P^{P^A}} & \mathbb{0} & \overline{\psi}^{\uparrow P \uparrow F} \fatsemi \overline{\psi}^{\uparrow F} \fatsemi \phi \fatsemi \overline{\text{ftn}}_Q^{\uparrow G} \end{array} \quad .$$

The remaining difference is between the last rows of the matrices:

$$\overline{\text{ftn}}_P^{\uparrow F} \fatsemi \overline{\psi}^{\uparrow F} \fatsemi \phi \stackrel{?}{=} \overline{\psi}^{\uparrow P \uparrow F} \fatsemi \overline{\psi}^{\uparrow F} \fatsemi \phi \fatsemi \overline{\text{ftn}}_Q^{\uparrow G} \quad .$$

By the inductive assumption, the law already holds for recursive calls of $\overline{\psi}$:

$$\text{ftn}_P \fatsemi \overline{\psi} = \overline{\psi}^{\uparrow P} \fatsemi \overline{\psi} \fatsemi \text{ftn}_Q \quad .$$

So, it remains to show that

$$(\overline{\psi}^{\uparrow P} \fatsemi \overline{\psi} \fatsemi \mathrm{ftn}_Q)^{\uparrow F} \fatsemi \phi \stackrel{?}{=} \overline{\psi}^{\uparrow P \uparrow F} \fatsemi \overline{\psi}^{\uparrow F} \fatsemi \phi \fatsemi \overline{\mathrm{ftn}_Q}^{\uparrow G} \quad .$$

This holds due to the naturality law of ϕ, in the form $\phi \fatsemi f^{\uparrow G} = f^{\uparrow F} \fatsemi \phi$. □

Heuristically, a free monad on a functor F will wrap F in a more complicated type constructor such that the resulting type has the required monad operations. If F is already a monad, constructing the free monad on F is unnecessary. Indeed, a value of type $\mathrm{Free}^{F,A}$ can be always mapped back to F^A while preserving the monad operations:

Statement 13.6.1.2 Assume that F is itself a monad, and denote $T \stackrel{\text{def}}{=} \mathrm{Free}^F$ for brevity.

(a) There is a monad morphism $p : T^A \to F^A$ defined by

$$p \stackrel{\text{def}}{=} \left| \begin{array}{c|c} & F^A \\ \hline A & \mathrm{pu}_F \\ F^{T^A} & \overline{p}^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} \right| \quad .$$

(b) The function $q : F^A \to T^A$ defined by $q(f) \stackrel{\text{def}}{=} \mathbb{0} + f \triangleright (a^{:A} \to a + \mathbb{0})^{\uparrow F}$ is *not* a monad morphism.
*** but $q \fatsemi f = \mathrm{id}$?

Proof **(a)** To verify the identity law of p:

$$\text{expect to equal } \mathrm{pu}_F : \quad \mathrm{pu}_T \fatsemi p = \left| \begin{array}{c|cc} & A & F^{T^A} \\ \hline A & \mathrm{id} & \mathbb{0} \end{array} \right| \fatsemi \left| \begin{array}{c|c} & F^A \\ \hline A & \mathrm{pu}_F \\ F^{T^A} & \overline{p}^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} \right| = \mathrm{pu}_F \quad .$$

To verify the composition law, write its two sides separately:

$$\text{left-hand side :} \quad \mathrm{ftn}_T \fatsemi p = \left| \begin{array}{c|cc} & A & F^{T^A} \\ \hline A & \mathrm{id} & \mathbb{0} \\ F^{T^A} & \mathbb{0} & \mathrm{id} \\ F^{T^{T^A}} & \mathbb{0} & \overline{\mathrm{ftn}_T}^{\uparrow F} \end{array} \right| \fatsemi \left| \begin{array}{c|c} & F^A \\ \hline A & \mathrm{pu}_F \\ F^{T^A} & \overline{p}^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} \right| = \left| \begin{array}{c|c} & F^A \\ \hline A & \mathrm{pu}_F \\ F^{T^A} & \overline{p}^{\uparrow F} \fatsemi \mathrm{ftn}_F \\ F^{T^{T^A}} & \overline{\mathrm{ftn}_T}^{\uparrow F} \fatsemi \overline{p}^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} \right| ,$$

$$\text{right-hand side :} \quad p^{\uparrow T} \fatsemi p \fatsemi \mathrm{ftn}_F = \left| \begin{array}{c|cc} & F^A & F^{T^{F^A}} \\ \hline A & \mathrm{pu}_F & \mathbb{0} \\ F^{T^A} & \overline{p}^{\uparrow F} \fatsemi \mathrm{ftn}_F & \mathbb{0} \\ F^{T^{T^A}} & \mathbb{0} & \overline{p}^{\uparrow T \uparrow F} \end{array} \right| \fatsemi \left| \begin{array}{c|c} & F^{F^A} \\ \hline F^A & \mathrm{pu}_F \\ F^{T^{F^A}} & \overline{p}^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} \right| \fatsemi \mathrm{ftn}_F$$

$$= \left| \begin{array}{c|c} & F^A \\ \hline A & \mathrm{pu}_F \fatsemi \underline{\mathrm{pu}_F \fatsemi \mathrm{ftn}_F} \\ F^{T^A} & \overline{p}^{\uparrow F} \fatsemi \mathrm{ftn}_F \fatsemi \underline{\mathrm{pu}_F \fatsemi \mathrm{ftn}_F} \\ F^{T^{T^A}} & \overline{p}^{\uparrow T \uparrow F} \fatsemi \overline{p}^{\uparrow F} \fatsemi \underline{\mathrm{ftn}_F \fatsemi \mathrm{ftn}_F} \end{array} \right| = \left| \begin{array}{c|c} & F^A \\ \hline A & \mathrm{pu}_F \\ F^{T^A} & \overline{p}^{\uparrow F} \fatsemi \mathrm{ftn}_F \\ F^{T^{T^A}} & \overline{p}^{\uparrow T \uparrow F} \fatsemi \overline{p}^{\uparrow F} \fatsemi \mathrm{ftn}_F^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} \right| .$$

The last two matrices differ only in the last rows, and the difference is

$$\overline{\mathrm{ftn}_T}^{\uparrow F} \fatsemi \overline{p}^{\uparrow F} \stackrel{?}{=} \overline{p}^{\uparrow T \uparrow F} \fatsemi \overline{p}^{\uparrow F} \fatsemi \mathrm{ftn}_F^{\uparrow F} \quad .$$

Omitting the lifting to F, we get:

$$\overline{\mathrm{ftn}_T} \fatsemi \overline{p} \stackrel{?}{=} \overline{p}^{\uparrow T} \fatsemi \overline{p} \fatsemi \mathrm{ftn}_F \quad .$$

This holds by the inductive assumption that the recursive calls to $\overline{p}$ already obey the composition law.

(b) The function $q \stackrel{\text{def}}{=} f \to \mathbb{0} + f \triangleright (x \to x + \mathbb{0})^{\uparrow F}$ fails the monad morphism identity law. Given $f \stackrel{\text{def}}{=} \text{pu}_F(a)$, we compute:

$$q(f) = \mathbb{0} + a \triangleright \text{pu}_F \triangleright (x \to x + \mathbb{0})^{\uparrow F} = \mathbb{0} + a \triangleright (x \to x + \mathbb{0}) \triangleright \text{pu}_F = \mathbb{0} + \text{pu}_F(a + \mathbb{0}) \quad .$$

However, the expected value is $\text{pu}_T(a) = a + \mathbb{0}$, which cannot equal $\mathbb{0} + \text{pu}_F(a + \mathbb{0})$.

13.7 Working with quantified types

***Move all this to an appendix?

In the notation used in this book, there is a key difference between the quantified type $\forall X. F^X$ and the type expression F^X that contains the type parameter X. In both cases, X is a completely unknown type parameter, and an example of Scala code implementing those types would be:

```
def f[X]: F[X] = ???
```

However, the type quantifier $\forall X$ implies that all values of type $\forall X. F^X$ must be implemented via fully parametric code. The code of the function `f[X]` may not make decisions based on the actual type passed at run time as the type parameter `X` into the function. The assumption of full parametricity enables us to reason about quantified types in a special way. This section explores the techniques of this reasoning.

13.7.1 The Yoneda identities

The following statement shows that the equivalence between the type $\forall X. (A \to X) \to X$ and the type A can be generalized to involve an arbitrary functor F.

Statement 13.7.1.1 (covariant Yoneda identity) If A is any fixed type and $F^\bullet$ is any functor, the type F^A is equivalent to the function type $\forall X. (A \to X) \to F^X$, where the functions of that type are additionally required to be natural transformations between the functors $A \to \bullet$ and $F^\bullet$. (The naturality law will be satisfied automatically if those functions are implemented using fully parametric code.)

Proof The isomorphism between the two types is implemented via two functions `toC` and `fromC`:

$$\text{toC} : F^A \to \forall X. (A \to X) \to F^X \quad , \qquad \text{toC} \stackrel{\text{def}}{=} p^{:F^A} \to \forall X. g^{:A \to X} \to p \triangleright g^{\uparrow F} \quad ,$$
$$\text{fromC} : (\forall X. (A \to X) \to F^X) \to F^A \quad , \qquad \text{fromC} \stackrel{\text{def}}{=} q^{:\forall X. (A \to X) \to F^X} \to q^A(\text{id}^{:A \to A}) \quad .$$

In the last line, the function q is required to be a natural transformation between functors $A \to \bullet$ and $F^\bullet$, where X is a type parameter of q. To apply q to an argument id of type $A \to A$ in the last line, we set q's type parameter X as $X = A$. This is indicated by the superscript A in q^A.

We need to show that $\text{fromC} \fatsemi \text{toC} = \text{id}$ and $\text{toC} \fatsemi \text{fromC} = \text{id}$. To verify that $\text{fromC} \fatsemi \text{toC} = \text{id}$, apply both sides to an arbitrary function $q^{:\forall X. (A \to X) \to F^X}$:

$$\text{expect to equal } q : \quad \underline{q^{:\forall X. (A \to X) \to F^X} \triangleright \text{fromC}} \fatsemi \text{toC} = q^A(\text{id}) \triangleright \text{toC} = \forall X. g^{:A \to X} \to q^A(\text{id}) \triangleright g^{\uparrow F} \quad .$$

To proceed, we use the assumption that q satisfies a naturality law: for any $r^{:A \to B}$ and $f^{:B \to C}$,

$$q^B(r) \triangleright f^{\uparrow F} = q^C(r \fatsemi f) \quad . \tag{13.1}$$

Applying this law with $B = A$, $C = X$, $f = g$, and $r = \text{id}$, we get

$$q^A(\text{id}) \triangleright g^{\uparrow F} = q^X(\text{id} \fatsemi g) = q^X(g) \quad .$$

We can now show that the function $q \triangleright \text{fromC} \fatsemi \text{toC}$ is the same as q:

$$q \triangleright \text{fromC} \fatsemi \text{toC} = \forall X.\, g^{:A\to X} \to \underline{q^A(\text{id}) \triangleright g^{\uparrow F}} = \forall X.\, \underline{g^{:A\to X} \to g} \triangleright q^X = \forall X.\, q^X = q \quad .$$

To verify that $\text{toC} \fatsemi \text{fromC} = \text{id}$, apply both sides to an arbitrary $p^{:F^A}$:

$$\text{expect to equal } p: \quad \underline{p^{:F^A} \triangleright \text{toC}} \fatsemi \text{fromC} = (\forall X.\, g^{:A\to X} \to p \triangleright g^{\uparrow F}) \triangleright \text{fromC}$$
$$= (g^{:A\to A} \to p \triangleright g^{\uparrow F})(\text{id}^{:A\to A}) = p \triangleright \text{id}^{\uparrow F} = p \quad .$$

It remains to check that the function $q^X \stackrel{\text{def}}{=} g^{:A\to X} \to p \triangleright g^{\uparrow F}$, used as an argument of `fromC`, is always a natural transformation. To verify the naturality law (13.1):

$$\text{expect to equal } q^C(r \fatsemi f): \quad q^B(r) \triangleright f^{\uparrow F} = p \triangleright r^{\uparrow F} \triangleright f^{\uparrow F} = p \triangleright r^{\uparrow F} \fatsemi f^{\uparrow F} = p \triangleright (r \fatsemi f)^{\uparrow F} = q^C(r \fatsemi f) \quad .$$

This completes the proof of the covariant Yoneda identity. □

A similar type equivalence holds for contrafunctors:

Statement 13.7.1.2 (contravariant Yoneda identity) If A is a fixed type and $H^\bullet$ is any contrafunctor, the type H^A is equivalent to the function type $\forall X.\, (X \to A) \to H^X$, where the functions of that type are additionally required to be natural transformations between the contrafunctors $\bullet \to A$ and $H^\bullet$.

Proof The isomorphism between the two types is implemented via two functions `toC` and `fromC`:

$$\text{toC} : H^A \to \forall X.\, (X \to A) \to H^X \quad , \qquad \text{toC} \stackrel{\text{def}}{=} p^{:H^A} \to \forall X.\, g^{:X\to A} \to p \triangleright g^{\downarrow H} \quad ,$$
$$\text{fromC} : (\forall X.\, (X \to A) \to H^X) \to H^A \quad , \qquad \text{fromC} \stackrel{\text{def}}{=} q^{:\forall X.\,(X\to A)\to H^X} \to q^A(\text{id}^{:A\to A}) \quad .$$

In the last line, the function q is required to be a natural transformation between contrafunctors $X \to A$ and F^X, where X is a type parameter of q.

We need to show that $\text{fromC} \fatsemi \text{toC} = \text{id}$ and $\text{toC} \fatsemi \text{fromC} = \text{id}$. To verify that $\text{fromC} \fatsemi \text{toC} = \text{id}$, apply both sides to an arbitrary function $q^{:\forall X.\,(X\to A)\to H^X}$:

$$\text{expect to equal } q: \quad \underline{q^{:\forall X.\,(X\to A)\to H^X} \triangleright \text{fromC}} \fatsemi \text{toC} = q^A(\text{id}) \triangleright \text{toC} = \forall X.\, g^{:X\to A} \to q^A(\text{id}) \triangleright g^{\downarrow H} \quad .$$

By assumption, q satisfies a naturality law: for any $r^{:B\to A}$ and $f^{:C\to B}$,

$$q^B(r) \triangleright f^{\downarrow H} = q^C(f \fatsemi r) \quad . \tag{13.2}$$

Applying this law with $B = A$, $C = X$, $f = g$, and $r = \text{id}$, we get

$$q^A(\text{id}) \triangleright g^{\downarrow H} = q^X(g \fatsemi \text{id}) = q^X(g) \quad .$$

We can now show that the function $q \triangleright \text{fromC} \fatsemi \text{toC}$ is the same as q:

$$q \triangleright \text{fromC} \fatsemi \text{toC} = \forall X.\, g^{:X\to A} \to q^X(g) = \forall X.\, q^X = q \quad .$$

To verify that $\text{toC} \fatsemi \text{fromC} = \text{id}$, apply both sides to an arbitrary $p^{:H^A}$:

$$\text{expect to equal } p: \quad \underline{p^{:H^A} \triangleright \text{toC}} \fatsemi \text{fromC} = (\forall X.\, g^{:X\to A} \to p \triangleright g^{\downarrow H}) \triangleright \text{fromC}$$
$$= (g^{:A\to A} \to p \triangleright g^{\downarrow H})(\text{id}^{:A\to A}) = p \triangleright \text{id}^{\downarrow H} = p \quad .$$

Finally, we check that the function $q^X \stackrel{\text{def}}{=} g^{:X\to A} \to p \triangleright g^{\downarrow H}$, used as an argument of `fromC`, is always a natural transformation. To verify the naturality law (13.2):

$$\text{expect to equal } q^C(f \fatsemi r): \quad q^B(r) \triangleright f^{\downarrow H} = p \triangleright r^{\downarrow H} \fatsemi f^{\downarrow H} = p \triangleright (f \fatsemi r)^{\downarrow H} = q^C(f \fatsemi r) \quad .$$

This completes the proof of the contravariant Yoneda identity. □

The Yoneda identities can be extended to many other contexts. For instance, a Yoneda identity holds for types parameterized by a type constructor:

Statement 13.7.1.3 (covariant Yoneda identity for functors) Assume that $P^\bullet$ is any type constructor and S^F is a higher-order functor, i.e., S depends covariantly on an arbitrary type constructor $F^\bullet$. (An example of such S is $S^F \stackrel{\text{def}}{=} F^{\text{Int}} \times F^{\text{String}}$.) Then the type S^P is equivalent to the function type $\forall F^\bullet. (P^\bullet \rightsquigarrow F^\bullet) \to S^F$, where the function is required to be natural in the parameter $F^\bullet$ (while its argument, a function of type $P^\bullet \rightsquigarrow F^\bullet$, is *not* required to be a natural transformation because P and F are not required to be functors). The corresponding naturality law for functions σ of type $\forall F^\bullet. (P^\bullet \rightsquigarrow F^\bullet) \to S^F$ involves arbitrary type constructors $Q^\bullet$, $R^\bullet$, and arbitrary functions $f : P^\bullet \rightsquigarrow Q^\bullet$ and $g : Q^\bullet \rightsquigarrow R^\bullet$, and may be written as

$$\sigma^Q(f) \fatsemi g^{\uparrow S} = \sigma^R(f \fatsemi g) \quad . \tag{13.3}$$

Here, $g^{\uparrow S}$ has type $S^Q \to S^R$ and is a lifting of the function $g : Q^\bullet \rightsquigarrow R^\bullet$ to the higher-order functor S.

Proof The isomorphism is implemented via two functions `toC` and `fromC`,

$$\text{toC} : S^P \to \forall F. (P \rightsquigarrow F) \to S^F \quad , \qquad \text{toC} \stackrel{\text{def}}{=} s^{:S^P} \to \forall F. g^{:P\rightsquigarrow F} \to s \triangleright g^{\uparrow S} \quad ,$$
$$\text{fromC} : (\forall F. (P \rightsquigarrow F) \to S^F) \to S^P \quad , \qquad \text{fromC} \stackrel{\text{def}}{=} \sigma^{:\forall F.(P\rightsquigarrow F)\to S^F} \to \sigma^P(\text{id}^{:P\rightsquigarrow P}) \quad .$$

In the last line, the function σ is required to be natural in its type parameter Q.

We need to show that $\text{fromC} \fatsemi \text{toC} = \text{id}$ and $\text{toC} \fatsemi \text{fromC} = \text{id}$. To verify that $\text{fromC} \fatsemi \text{toC} = \text{id}$, apply both sides to an arbitrary function $\sigma^{:\forall F.(P\rightsquigarrow F)\to S^F}$:

$$\text{expect to equal } \sigma : \quad \sigma^{:\forall F.(P\rightsquigarrow F)\to S^F} \triangleright \text{fromC} \fatsemi \text{toC} = \sigma^P(\text{id}) \triangleright \text{toC} = \forall F. g^{:P\rightsquigarrow F} \to \sigma^P(\text{id}) \triangleright g^{\uparrow S} \quad .$$

Since by assumption σ satisfies the naturality law (13.3), we may apply that law with $Q = P$, $R = F$, and $f = \text{id}$:

$$\sigma^P(\text{id}) \fatsemi g^{\uparrow S} = \sigma^F(\text{id} \fatsemi g) = F^F(g) \quad .$$

It follows that the function $\sigma \triangleright \text{fromC} \fatsemi \text{toC}$ is the same as σ:

$$\sigma \triangleright \text{fromC} \fatsemi \text{toC} = \forall F. g^{:P\rightsquigarrow F} \to \sigma^F(g) = \forall F. \sigma^F = \sigma \quad .$$

To verify that $\text{toC} \fatsemi \text{fromC} = \text{id}$, apply both sides to an arbitrary $s^{:S^P}$:

$$\text{expect to equal } s : \quad s^{:S^P} \triangleright \text{toC} \fatsemi \text{fromC} = s \triangleright \text{toC} \triangleright \text{fromC} = (\forall F. g^{:P\rightsquigarrow F} \to s \triangleright g^{\uparrow S}) \triangleright \text{fromC}$$
$$= (g^{:P\rightsquigarrow P} \to s \triangleright g^{\uparrow S})(\text{id}^{:P\rightsquigarrow P}) = s \triangleright \text{id}^{\uparrow S} = s \quad .$$

It remains to check that the function $\sigma^F \stackrel{\text{def}}{=} g^{:P\rightsquigarrow F} \to s \triangleright g^{\uparrow S}$, used as an argument of `fromC`, is natural in F. To verify the naturality law (13.3):

$$\text{expect to equal } \sigma^R(f \fatsemi g) : \quad \sigma^Q(f) \fatsemi g^{\uparrow S} = s \triangleright f^{\uparrow S} \fatsemi g^{\uparrow S} = s \triangleright (f \fatsemi g)^{\uparrow S} = \sigma^R(f \fatsemi g) \quad .$$

13.7.2 Recursive type equations with different fixpoints

A recursive type is usually defined as a **fixpoint** of a functor; that is, a solution of a type equation of the form $T \cong F^T$, where F is a "structure functor" that specifies the details of the type recursion. A solution of the type equation $T \cong F^T$ is a type T that is equivalent to F^T via two isomorphisms:

$$\text{fix} : F^T \to T \quad , \qquad \text{unfix} : T \to F^T \quad , \qquad \text{fix} \fatsemi \text{unfix} = \text{id} \quad , \quad \text{unfix} \fatsemi \text{fix} = \text{id} \quad .$$

Section 8.5.1 gave a condition for implementability of such types T. We will now consider the question of whether there can be several fixpoints T.

If a functor has several *inequivalent* fixpoints T_1, T_2, etc., it means that each T_i separately satisfies the fixpoint equation $T \cong F^T$. An example is the fixpoint equation for the "lazy list":

$$L^A \cong \mathbb{1} + (\mathbb{1} \to A \times L^A) \quad . \tag{13.4}$$

We may also write the same equation using a structure functor F:

$$L^A \cong F^{A,L^A} \quad , \qquad F^{A,R} \stackrel{\text{def}}{=} \mathbb{1} + (\mathbb{1} \to A \times R) \quad .$$

A solution of this fixpoint equation can be visualized (non-rigorously) as the type

$$L^A = F^{A,F^{A,F^{A,\ldots}}} = \mathbb{1} + (\mathbb{1} \to A \times (\mathbb{1} + (\mathbb{1} \to A \times (\ldots)))) \quad ,$$

representing an "on-call" list of values of type A. To get the next value of type A, one must run a function call. In this way, the elements of the on-call list are computed only when needed. It could happen that a value of type L^A will *never* stop yielding new values of type A if we keep requesting the next elements of the list.

To see that the fixpoint equation (13.4) has (at least) three inequivalent solutions, consider a function `toList` that converts a value of type L^A into a sequence of type `List[A]`, whose elements are eagerly evaluated. The function `toList` keeps recursively requesting new elements of the on-call list and accumulates the resulting values of type A:

$$\text{toList} : L^A \to \text{List}^A \quad , \qquad \text{toList} \stackrel{\text{def}}{=} \left| \begin{array}{c||cc} & \mathbb{1} & \mathbb{1} + A \times \text{List}^A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ \mathbb{1} \to A \times L^A & \mathbb{0} & p \to p(1) \triangleright (\text{id} \boxtimes \overline{\text{toList}}) \end{array} \right| \quad .$$

Does this function terminate? It is clear that `toList` will terminate only if the on-call list eventually stops yielding new values of type A. On the other hand, if the on-call list never stops yielding new values of type A, the function `toList` will not terminate. So, `toList` must be a partial function.

Let us denote by L^A_{fin} the type corresponding to "finite" on-call lists, i.e., on-call lists that eventually stop yielding new values of type A. Then the function

$$\text{toList} : L^A_{\text{fin}} \to \text{List}^A$$

is total. On-call lists of type L^A_{fin} contain a finite number of values and thus are equivalent to eager lists. To make this equivalence formal, we may define isomorphism functions $\text{toList} : L^A_{\text{fin}} \to \text{List}^A$ and $\text{fromList} : \text{List}^A \to L^A_{\text{fin}}$.

Let us also denote by L^A_{inf} the type corresponding to "infinite" on-call lists, i.e., those that *never* stop yielding new values of type A. The function `toList` does not terminate for arguments of type L^A_{inf}, because a list (of type List^A) cannot contain infinitely many values. So, the types L^A_{fin} and L^A_{inf} are *not* equivalent. Were they equivalent, we would have an isomorphism $q : L^A_{\text{inf}} \to L^A_{\text{fin}}$, and then we could compose q with `toList` to obtain a function $\text{toList} : L^A_{\text{inf}} \to \text{List}^A$ that terminates, which is impossible.

To show that both types (L^A_{fin} and L^A_{inf}) satisfy the fixpoint type equation (13.4), we can implement the corresponding isomorphisms `fix` and `unfix`. Each of these functions will either add or remove one element at the beginning of the list. These operations keep finite lists finite and infinite lists infinite. So, a composition (such as, unfix ⨾ fix) of these isomorphisms will act as an identity function on L^A_{fin} or on L^A_{inf}.

The type L^A is an on-call list that may or may not terminate. So, L^A is equivalent to a disjunction $L^A_{\text{fin}} + L^A_{\text{inf}}$. We see that the type equation (13.4) has three inequivalent solutions: L^A_{fin}, L^A_{inf}, and L^A.

Some fixpoints represent "larger" types than other fixpoints. For instance, L^A is "larger" than either of L^A_{fin} and L^A_{inf}. To see this formally, we consider the functions $f_1 : L^A_{\text{fin}} \to L^A$ and $f_2 : L^A_{\text{inf}} \to L^A$. The function f_1 embeds values of type L^A_{fin} (finite lists) in the type L^A that includes both finite and infinite lists. It is clear that f_1 is injective; the same holds also for f_2.

Additionally, we can show that the functions f_1 and f_2 preserve the fixpoint structure of the types. ***

13.7.3 The Church encoding of recursive types

Any given type can be represented in a **Church encoding**, which is a function type with a universally quantified type parameter. A simple Church encoding is given by the type equivalence

$$T \cong \forall X.\, (T \to X) \to X \quad ,$$

which follows from the covariant Yoneda identity (Statement 13.7.1.1 with the functor $F \stackrel{\text{def}}{=} \text{Id}$).

There is rarely an advantage in replacing a simple type T by a more complicated function type, $\forall X.\, (T \to X) \to X$. However, the Church encoding has a different form when T is a recursive type.

Consider a recursive type T defined by a fixpoint equation $T \stackrel{\text{def}}{=} F^T$ with a given structure functor F. We could write a Church encoding for T as $\forall X.\, (T \to X) \to X$ or as $\forall X.\, (F^T \to X) \to X$, but these encodings give no advantages. It turns out that another, more useful Church encoding for T is:

$$T \cong \forall X.\, (F^X \to X) \to X \quad \text{if the type } T \text{ is defined by } T \stackrel{\text{def}}{=} F^T \quad . \tag{13.5}$$

The Scala code for the type $\forall X.\, (F^X \to X) \to X$ is:

```
trait TC[F[_]] { def run[X](fold: F[X] => X): X }
```

In this section, we will study the type equivalence (13.5).

Note that the Yoneda lemma cannot be used to prove Eq. (13.5). The Yoneda lemma only applies to types of the form $\forall X.\, (A \to X) \to F^X$, where the type A cannot depend on the quantified type X.

The following statement[6] shows that the Church encoding (13.5) is a fixpoint:

Statement 13.7.3.1 For a given structure functor F, define the type T as

$$T \stackrel{\text{def}}{=} \forall X.\, (F^X \to X) \to X \quad .$$

Additionally, we require all values t of type T to satisfy the strong dinaturality law, which for the given type T has the following form: for any $r^{:F^A \to A}$, $s^{:F^B \to B}$, and $f^{:A \to B}$,

$$\text{if} \quad r \fatsemi f = f^{\uparrow F} \fatsemi s \quad \text{then} \quad r \triangleright t^A \triangleright f = s \triangleright t^B \quad . \tag{13.6}$$

(Strong dinaturality holds for all fully parametric functions of type T. This follows from Example D.3.6.13(c) with $G^A \stackrel{\text{def}}{=} A$ and $H^A \stackrel{\text{def}}{=} A$.) Defined in this way, the type T is a solution of the fixpoint equation $T \cong F^T$.

Proof We will use typed holes to derive the code for the isomorphisms

$$\text{fix} : F^T \to T \quad , \qquad \text{unfix} : T \to F^T \quad .$$

Begin by writing out the type signature of `fix` and an implementation with a typed hole:

$$\text{fix} : F^{\forall X.\, (F^X \to X) \to X} \to \forall Y.\, (F^Y \to Y) \to Y \quad ,$$
$$\text{fix} \stackrel{\text{def}}{=} f^{:F^{\forall X.\, (F^X \to X) \to X}} \to \forall Y.\, q^{:F^Y \to Y} \to ???^{:Y} \quad .$$

The only way of computing a value of type Y is to apply q to an argument of type F^Y:

$$\text{fix} = f^{:F^{\forall X.\, (F^X \to X) \to X}} \to \forall Y.\, q^{:F^Y \to Y} \to \left(???^{:F^Y}\right) \triangleright q \quad .$$

The functor F is arbitrary, so the only way of computing a value of type F^Y is to use the given value f with a `map` method (the only method we can use with any functor F):

$$???^{:F^Y} = f \triangleright \left(p^{:\forall X.\, (F^X \to X) \to X} \to ???^{:Y}\right)^{\uparrow F} \quad .$$

[6] See also the papers "A note on strong dinaturality" (http://www.cs.ioc.ee/~tarmo/papers/fics10.pdf) and "Build, augment, and destroy universally" (https://kodu.ut.ee/~varmo/papers/aplas04.ps.gz).

Since the type parameter X is universally quantified inside p, we may set X to any type as needed. So, we set $X = Y$ and use the given value $q^{:F^Y \to Y}$ to fill the typed hole:

$$???^{:F^Y} = f \triangleright (p^{:\forall X.\,(F^X \to X) \to X} \to q \triangleright p^Y)^{\uparrow F} \quad .$$

This allows us to complete the code of `fix`:

$$\text{fix} \stackrel{\text{def}}{=} f^{:F^{\forall X.\,(F^X \to X) \to X}} \to \forall Y.\, q^{:F^Y \to Y} \to f \triangleright \left(p^{:\forall X.\,(F^X \to X) \to X} \to q \triangleright p^Y\right)^{\uparrow F} \triangleright q \quad .$$

We turn to implementing `unfix`:

$$\begin{aligned} &\text{unfix} : \left(\forall Y.\,(F^Y \to Y) \to Y\right) \to F^{\forall X.\,(F^X \to X) \to X}\right) \quad , \\ &\text{unfix} \stackrel{\text{def}}{=} t^{:\forall Y.\,(F^Y \to Y) \to Y} \to ???^{:F^T} \quad . \end{aligned}$$

The only way of filling the typed hole $???^{:F^T}$ is to apply t^Y while setting the type parameter Y to F^T:

$$\text{unfix} = t^{:\forall Y.\,(F^Y \to Y) \to Y} \to t^{F^T}\left(???^{:F^{F^T} \to F^T}\right) \quad .$$

We can now fill the typed hole $???^{:F^{F^T} \to F^T}$ by lifting `fix` to the functor F:

$$\text{unfix} \stackrel{\text{def}}{=} t^{:\forall Y.\,(F^Y \to Y) \to Y} \to t^{F^T}\left(\text{fix}^{\uparrow F}\right) \quad .$$

It remains to show that `fix` and `unfix` are inverses. Start with one direction:

$$\begin{aligned} \text{expect to equal } t : \quad & t^{:T} \triangleright \text{unfix} \fatsemi \text{fix} = t^{F^T}\left(\text{fix}^{\uparrow F}\right) \triangleright \text{fix} \\ & = \forall Y.\, q^{:F^Y \to Y} \to \text{fix}^{\uparrow F} \triangleright t^{F^T} \triangleright \left(p^{:\forall X.\,(F^X \to X) \to X} \to q \triangleright p^Y\right)^{\uparrow F} \triangleright q \quad . \end{aligned}$$

The resulting function will be equal to t if we show that it gives the same result when applied to an arbitrary argument $q^{:F^Y \to Y}$ (where the type parameter Y is free):

$$\text{fix}^{\uparrow F} \triangleright t^{F^T} \triangleright \left(p^{:\forall X.\,(F^X \to X) \to X} \to p^Y(q)\right)^{\uparrow F} \triangleright q \stackrel{?}{=} t^Y\left(q^{:F^Y \to Y}\right) \quad . \tag{13.7}$$

To proceed, we need to use the strong dinaturality law of t with suitable r, s, and f:

$$r^{:F^A \to A} \triangleright t^A \triangleright f^{:A \to B} = s^{:F^B \to B} \triangleright t^B \quad . \tag{13.8}$$

This law will reproduce Eq. (13.7) if we choose r, s, and f such that

$$r = \text{fix}^{\uparrow F} \quad , \qquad f = \left(p^{:\forall X.\,(F^X \to X) \to X} \to q \triangleright p^Y\right)^{\uparrow F} \fatsemi q \quad , \qquad s = q \quad .$$

For these equations to hold, the type parameters A and B must be set appropriately in Eq. (13.8). The type of $\text{fix}^{\uparrow F}$ is $F^{F^T} \to F^T$, which means that $A = F^T$. The type of q is $F^Y \to Y$, so $B = Y$. This agrees with the type of $f^{:A \to B}$. So, the strong dinaturality law (13.8) will yield Eq. (13.7) as long as the chosen variables satisfy the condition

$$r \fatsemi f \stackrel{?}{=} f^{\uparrow F} \fatsemi s \quad .$$

It remains to verify that the last line holds. Substituting the values of the variables, we get:

$$\begin{aligned} & \text{fix}^{\uparrow F} \fatsemi f \stackrel{?}{=} f^{\uparrow F} \fatsemi q \quad , \\ \text{or equivalently :} \quad & \text{fix}^{\uparrow F} \fatsemi \left(p^{:\forall X.\,(F^X \to X) \to X} \to q \triangleright p^Y\right)^{\uparrow F} \fatsemi q \stackrel{?}{=} \left(p^{:\forall X.\,(F^X \to X) \to X} \to q \triangleright p^Y\right)^{\uparrow F \uparrow F} \fatsemi q^{\uparrow F} \fatsemi q \quad . \end{aligned}$$

Evaluate the function composition in the left-hand side, and make both sides equal:

$$\begin{aligned}
& \underline{\text{fix}}^{\uparrow F} \fatsemi \left(p^{:\forall X.\,(F^X \to X) \to X} \to q \triangleright p^Y\right)^{\uparrow F} \fatsemi q \\
\text{expand function}: \quad & = \left(\left(f^{:F^T} \to f \triangleright \text{fix}\right) \fatsemi \left(p \to q \triangleright p^Y\right)\right)^{\uparrow F} \fatsemi q = \left(f^{:F^T} \to q \triangleright \left(f \triangleright \underline{\text{fix}}\right)^Y\right)^{\uparrow F} \fatsemi q \\
\text{definition of fix}: \quad & = \left(\underline{f^{:F^T} \to f \triangleright} \left(p \to q \triangleright p^Y\right)^{\uparrow F} \fatsemi q\right)^{\uparrow F} \fatsemi q = \left(\left(p \to q \triangleright p^Y\right)^{\uparrow F} \fatsemi q\right)^{\uparrow F} \fatsemi q \quad .
\end{aligned}$$

The last expression is equal to the right-hand side.

This proves one direction of the isomorphism between T and F^T ($\text{unfix} \fatsemi \text{fix} = \text{id}$).

To verify the opposite direction of the isomorphism, we write:

$$\begin{aligned}
\text{expect to equal } f: \quad & f^{:F^T} \triangleright \text{fix} \triangleright \text{unfix} = \left(\forall Y.\, q^{:F^Y \to Y} \to f \triangleright \left(p^{:\forall X.\,(F^X \to X) \to X} \to q \triangleright p^Y\right)^{\uparrow F} \triangleright q\right) \triangleright \text{unfix} \\
& = f \triangleright \left(p^{:\forall X.\,(F^X \to X) \to X} \to p^{F^T}\left(\text{fix}^{\uparrow F}\right)\right)^{\uparrow F} \triangleright \text{fix}^{\uparrow F} \\
& = f \triangleright \left(p^{:\forall X.\,(F^X \to X) \to X} \to p^{F^T}\left(\text{fix}^{\uparrow F}\right) \triangleright \text{fix}\right)^{\uparrow F} \quad .
\end{aligned}$$

The last value will be equal to f if the function under $(...)^{\uparrow F}$ is an identity function:

$$p^{F^T}\left(\text{fix}^{\uparrow F}\right) \triangleright \text{fix} \stackrel{?}{=} p \quad .$$

Since p is of function type, both sides must be equal when applied to an arbitrary $s^{:F^B \to B}$:

$$s \triangleright \left(p^{F^T}\left(\text{fix}^{\uparrow F}\right) \triangleright \text{fix}\right) \stackrel{?}{=} s \triangleright p \quad . \tag{13.9}$$

To prove the last equation, we use the assumption that all values of type T satisfy the strong dinaturality law. So, the law must apply to the value p:

$$r \triangleright p \triangleright f = s \triangleright p \quad .$$

This law reproduces Eq. (13.9) if we define r and f by

$$r \stackrel{\text{def}}{=} \text{fix}^{\uparrow F} \quad , \qquad f \stackrel{\text{def}}{=} u^{:F^T} \to s \triangleright \left(u \triangleright \text{fix}\right) \quad .$$

It remains to verify that the assumption of the strong dinaturality law holds:

$$\begin{aligned}
& r \fatsemi f \stackrel{?}{=} f^{\uparrow F} \fatsemi s \quad , \\
\text{or equivalently}: \quad & \text{fix}^{\uparrow F} \fatsemi \left(u \to s \triangleright \left(u \triangleright \text{fix}\right)\right) \stackrel{?}{=} \left(u^{:F^T} \to s \triangleright \left(u \triangleright \text{fix}\right)\right)^{\uparrow F} \fatsemi s \quad .
\end{aligned}$$

Rewrite the left-hand side above until it becomes equal to the right-hand side:

$$\begin{aligned}
& \text{fix}^{\uparrow F} \fatsemi \left(u \to s \triangleright \left(u \triangleright \text{fix}\right)\right) = \left(u^{:F^{F^T}} \to u \triangleright \text{fix}^{\uparrow F}\right) \fatsemi \left(u \to s \triangleright \left(u \triangleright \text{fix}\right)\right) \\
\text{compute composition}: \quad & = u \to s \triangleright \left(u \triangleright \text{fix}^{\uparrow F} \triangleright \underline{\text{fix}}\right) = \underline{u \to u \triangleright} \text{fix}^{\uparrow F} \triangleright \left(q \to s \triangleright q\right)^{\uparrow F} \fatsemi s \\
\text{unexpand function}: \quad & = \left(\text{fix} \fatsemi \left(q \to s \triangleright q\right)\right)^{\uparrow F} \fatsemi s = \left(\left(u^{:F^T} \to u \triangleright \text{fix}\right) \fatsemi \left(q \to s \triangleright q\right)\right)^{\uparrow F} \fatsemi s \\
\text{compute composition}: \quad & = \left(u^{:F^T} \to s \triangleright \left(u \triangleright \text{fix}\right)\right)^{\uparrow F} \fatsemi s \quad .
\end{aligned}$$

The two sides are now equal. □

A curious property of the type T is that it is a function with argument of type $F^X \to X$, where X can be any type, including T itself. But we already have a function of type $F^T \to T$; it is the function `fix`. So, applying a value t of type T to the function `fix` yields again a value of type T. As the next statement shows, that value is the same as t:

Statement 13.7.3.2 Consider the Church-encoded type T and the function `fix` defined in Statement 13.7.3.1. It follows from the strong dinaturality law[7] for any value $t^{:T}$ that

$$t^T(\text{fix}) = t \quad . \tag{13.10}$$

Proof We use the strong dinaturality law (13.6):

$$\text{if} \quad r \fatsemi f = f^{\uparrow F} \fatsemi s \quad \text{then} \quad r \triangleright t^A \triangleright f = s \triangleright t^B \quad .$$

It remains to choose suitable values $r^{:F^A \to A}$, $s^{:F^B \to B}$, and $f^{:A \to B}$ so that the law (13.6) reproduces Eq. (13.10). Since the law always involves applying the function t to some arguments, while the right-hand side of Eq. (13.10) contains just t, let us apply both sides of Eq. (13.10) to an arbitrary value $s^{:F^B \to B}$, where the type B is also arbitrary:

$$s \triangleright (\text{fix} \triangleright t^T) \stackrel{?}{=} s \triangleright t^B \quad .$$

The left-hand side will have the form $r \triangleright t^A \triangleright f$ if we set $A = T$, $r = \text{fix}$, and f a function that applies its argument to s:

$$f^{:T \to B} \stackrel{\text{def}}{=} u^{:T} \to s \triangleright u^B \quad .$$

It remains to verify the assumption of the strong dinaturality law (13.6):

$$r \fatsemi f \stackrel{?}{=} f^{\uparrow F} \fatsemi s \quad ,$$

$$\text{or equivalently :} \quad \text{fix} \fatsemi (u^{:T} \to s \triangleright u^B) \stackrel{?}{=} (u^{:T} \to s \triangleright u^B)^{\uparrow F} \fatsemi s \quad .$$

Rewrite the left-hand side of the last line above:

$$\begin{aligned} & \text{fix} \fatsemi (u^{:T} \to s \triangleright u^B) = f^{:F^T} \to s \triangleright (f \triangleright \text{fix})^B \\ \text{definition of fix :} \quad & = \underline{f^{:F^T} \to f \triangleright} (q \to s \triangleright q)^{\uparrow F} \fatsemi s \\ \text{unexpand function :} \quad & = (q \to s \triangleright q)^{\uparrow F} \fatsemi s \quad . \end{aligned}$$

This is equal to the left-hand side after renaming s to u. □

The Church encoding $T \stackrel{\text{def}}{=} \forall X. (F^X \to X) \to X$ of the fixpoint has a special property: for any other fixpoint R, there is a unique fixpoint-preserving map $T \to R$ called a **catamorphism**. To define that map, assume that the fixpoint R has a known function $\text{fix}_R : F^R \to R$ and write:

$$\text{cata} : T \to R \quad , \qquad \text{cata} \stackrel{\text{def}}{=} t^{:\forall X. (F^X \to X) \to X} \to \text{fix}_R \triangleright t^R \quad .$$

Statement 13.7.3.3 **(a)** The function `cata` (defined above) is a fixpoint-preserving map.

(b) Any other fixpoint-preserving function of type $T \to R$ is equal to `cata`.

Proof ***

13.7.4 The co-Yoneda identities

The Yoneda identities allow us in many cases to simplify type expressions with universal quantifiers. Similar identities hold for existentially quantified types.

Statement 13.7.4.1 For any functor F and any contrafunctor H, the following identities hold:

$$\begin{aligned} \textbf{(a)}\ \text{covariant co-Yoneda identity :} \quad & \exists R. F^R \times (R \to A) \cong F^A \quad , \\ \textbf{(b)}\ \text{contravariant co-Yoneda identity :} \quad & \exists R. H^R \times (A \to R) \cong H^A \quad . \end{aligned}$$

(The equivalence is observational.)

[7] Dan Doel gave a proof using relational parametricity: see https://cs.stackexchange.com/questions/131901/

Proof ***

□

Note that the Scala type `Any` closely corresponds to the type $\exists X.\, X$, which is observationally equivalent to `Unit` . The advantage of using `Any` instead of `Unit` is that `Any` is a supertype of *all* Scala types. Indeed, for any type T there is an injective function $T \to \exists X.\, X$, This function corresponds to a function of type `T => Any` in Scala:

```
def toAny[T](t: T): Any = t
```

This is just an identity function that relabels the types; so, this function establishes the subtyping relation `T <: Any`.

13.8 Discussion

Free constructions in mathematics: Example I Consider the Cyrillic letter ц (tsè) and the Chinese word 水 (shuǐ)

We want to *multiply* ц by 水. Multiply how?

Say, we want an associative (but noncommutative) product of them

So we want to define a *semigroup* that *contains* ц and 水 as elements

while we still know nothing about ц and 水

Consider the set of all *unevaluated expressions* such as ц·水·水·ц·水

Here ц·水 is different from 水·ц but $(a \cdot b) \cdot c = a \cdot (b \cdot c)$

All these expressions form a **free semigroup** generated by ц and 水

This is the most unrestricted semigroup that contains ц and 水

Example calculation: (水·水)·(ц·水)·ц = 水·水·ц·水·ц

How to represent this as a data type:

Tree encoding: the full expression tree: (((水,水),(ц,水)),ц)

Implement the operation $a \cdot b$ as pair constructor (easy)

Reduced encoding, as a "smart" structure: List(水,水,ц,水,ц)

Implement $a \cdot b$ by concatenating the lists (more expensive)

Free constructions in mathematics: Example II Want to define a product operation for n-dimensional vectors: $\mathbf{v}_1 \otimes \mathbf{v}_2$

The $\otimes$ must be linear and distributive (but not commutative):

$$\begin{aligned}
\mathbf{u}_1 \otimes \mathbf{v}_1 + (\mathbf{u}_2 \otimes \mathbf{v}_2 + \mathbf{u}_3 \otimes \mathbf{v}_3) &= (\mathbf{u}_1 \otimes \mathbf{v}_1 + \mathbf{u}_2 \otimes \mathbf{v}_2) + \mathbf{u}_3 \otimes \mathbf{v}_3 \\
\mathbf{u} \otimes (a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2) &= a_1 (\mathbf{u} \otimes \mathbf{v}_1) + a_2 (\mathbf{u} \otimes \mathbf{v}_2) \\
(a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2) \otimes \mathbf{u} &= a_1 (\mathbf{v}_1 \otimes \mathbf{u}) + a_2 (\mathbf{v}_2 \otimes \mathbf{u})
\end{aligned}$$

We have such a product for 3-dimensional vectors; but it cannot be made to work for 2 or 4-dimensional vectors

Consider *unevaluated expressions* of the form $\mathbf{u}_1 \otimes \mathbf{v}_1 + \mathbf{u}_2 \otimes \mathbf{v}_2 + ...$

A free vector space generated by pairs of vectors

Impose the equivalence relationships shown above

The result is known as the **tensor product**

Tree encoding: full unevaluated expression tree

A list of any number of vector pairs $\sum_i \mathbf{u}_i \otimes \mathbf{v}_i$

Reduced encoding: an $n \times n$ matrix

Reduced encoding requires proofs and more complex operations

13.8.1 Beyond Yoneda: using parametricity to simplify quantified types

The covariant Yoneda identity,

$$\forall R.\, (A \to R) \to F^R \cong F^A \quad ,$$

and other similar identities hold for fully parametric code. If we assume parametricity, we can also simplify certain expressions containing quantified types where the Yoneda identities cannot be applied.

Statement 13.8.1.1 For any exponential-polynomial functor F and for any profunctor $P^{X,Y}$, the types $\forall A.\, F^{P^{A,A}}$ and $F^{\forall A.\, P^{A,A}}$ are equivalent when restricted to fully parametric implementations. The same holds when F is an exponential-polynomial contrafunctor.

Proof ***

Statement 13.8.1.2 **(a)** For any profunctor $P^{X,Y}$, the types $\exists A.\, P^{A,A}$ and $\forall B.\, (\forall A.\, (P^{A,A} \to B)) \to B$ are equivalent when restricted to fully parametric implementations.

(b) Without the outer quantifier ($\forall B$), this property does not hold: the types $(\exists A.\, P^{A,A}) \to B$ and $\forall A.\, (P^{A,A} \to B)$ are *not* equivalent, at least for some profunctors P.

Proof ***

14 Computations in functor blocks. III. Monad transformers

14.1 Practical use

14.1.1 Combining monadic effects via functor composition

Monads describe effects that depend on previously computed values. It is often necessary to combine effects of several monads in one. For example, the effect of `Future` is to compute values concurrently, while `Option` represents a possibly missing value. We would like to describe a possibly missing value that is computed concurrently. The type constructors `Future` and `Option` are not compatible (not subtypes of each other), so we cannot simply use them together in one functor block:

```
for {
  x <- Future(1)
  y <- Option(x + 1) // Type error.
```

One solution is to combine the effects of `Future` and `Option` in a single "larger" monad. Perhaps the type constructor of that monad is `Future[Option[A]]`? However, we run into a problem if we want to use this nested type directly in a functor block. For instance, this code will not compile:

```
for {
  xOpt <- Future(Option(1)) // We have xOpt: Option[Int] here.
  x    <- xOpt              // Type error.
```

We cannot mix source lines of `Future` and `Option` types because the source types must be compatible throughout the entire functor block. We could work with the nested type using code like this:

```
val result: Future[Option[Int]] = for {
  xOpt <- Future(Option(1))  // xOpt: Option[Int]
  tOpt <- xOpt match {       // yOpt: Option[Int]
    case None      => Future.successful(None)     // This expression must be a Future[Option[Int]].
    case Some(x)   => for {
        yOpt <- Future(Option(x + 1))
        zOpt <- yOpt match { // zOpt: Option[Int]
          case None      => Future.successful(None) // Must return a Future[Option[Int]] everywhere.
          case Some(y)   => Future(Option(y + 3))
        }
      } yield zOpt
  }
} yield tOpt
```

The type constructor forces us to use pattern matching with nested functor blocks, since that is the only way of getting access to values of type `A` within `Future[Option[A]]`. The code is repetitive and deeply nested, which makes it hard to read and to change.

The first step towards solving the problem is to rewrite this monadic program as a direct chain of computations depending on the results of previous ones:

```
val result: Future[Option[Int]] = for { // This will not compile!
  x <- Future(Option(1))                // The type is x: Option[Int], although we wanted x: Int.
  y <- Future(Option(x + 1))            // Type error: x + 1 is undefined.
  z <- Future(Option(y + 3))
} yield z
```

To be able to write code in this way, we need to implement the `flatMap` operation directly on the type `Future[Option[A]]` with respect to the type parameter `A`. To achieve that, let us wrap the type `Future[Option[A]]` into a new class `FutOpt` and implement the `map` and `flatMap` methods for that class:

```
final case class FutOpt[A](nested: Future[Option[A]]) {
  def map[B](f: A => B)(implicit ec: ExecutionContext): FutOpt[B] = FutOpt(nested.map(_.map(f)))
  def flatMap[B](f: A => FutOpt[B])(implicit ec: ExecutionContext): FutOpt[B] =
    FutOpt(nested.flatMap {      // Provide a function Option[A] => Future[Option[B]].
      case None    => Future.successful(None) // Must return a Future[Option[B]] here.
      case Some(x) => f(x).nested
    })
}

import scala.concurrent.ExecutionContext.Implicits.global

val result: FutOpt[Int] = for {      // This code compiles now.
  x <- FutOpt(Future(Option(1)))     // x: Int
  y <- FutOpt(Future(Option(x + 1))) // y: Int
  z <- FutOpt(Future(Option(y + 3))) // z: Int
} yield z
```

To shorten the code, we define conversions ("lifts") from `Future` to `FutOpt` and from `Option` to `FutOpt`:

```
implicit class FutOptLift1[A](fut: Future[A])(implicit ec: ExecutionContext) {
  def up: FutOpt[A] = FutOpt(fut.map(x => Some(x)))
}
implicit class FutOptLift2[A](opt: Option[A]) {
  def up: FutOpt[A] = FutOpt(Future.successful(opt))
}
```

We can now rewrite the previous code in a more readable form:

```
val result: FutOpt[Int] = for {
  x <- Option(1).up     // x: Int
  y <- Future(x + 1).up // y: Int
  z <- Option(y + 3).up // z: Int
} yield z  // This eventually evaluates to ‘FutOpt(Future.Successful(Some(5)))‘.
```

We may hope to apply the same technique when mixing effects of arbitrary monads, such as:

```
val result: SomeBigMonad[Int] = for {  // After some definitions, this code should work:
  x <- (1 to n).up                     // x: Int
  y <- Future { computation(x) }.up    // y: Int
  z <- Try { maybeError(y) }.up        // z: Int
} yield z
```

This code will work if we somehow create a new monad that combines `Seq`, `Future`, and `Try`. But it turns out that a simple composition of these type constructors in a certain chosen order (say, as `Seq[Future[Try[A]]]`, or as `Future[Try[Seq[A]]]`, etc.) cannot be made into a lawful monad.

No simple recipe is known for combining two or more arbitrary monads into a "larger" monad. Certain techniques are known to work for specific kinds of monads. This chapter focuses on developing these techniques.

We have seen that `Future[A]` and `Option[A]` are combined in a new monad as `Future[Option[A]]`. Does functor composition always work? For some monads M_1 and M_2, one of the compositions ($M_1 \circ M_2$ or $M_2 \circ M_1$) may turn out to be a monad. However, `Option[Future[A]]` is not a monad. By trial and error, we find further examples where the functor composition works and where it does not (see Table 14.1; incorrect choices are marked with a strike-through line).

It turns out that some monads do not compose in either order, while others have a "preferred" order. For instance, the `Reader` monad ($M^A \stackrel{\text{def}}{=} R \to A$) always composes on the outside (as Statement 10.2.8.5 shows) but not necessarily on the inside. On the other hand, the `Either` monad ($M^A \stackrel{\text{def}}{=} Z + A$) always composes on the inside (we will prove that in Section 14.4), but not necessarily on the outside.

Example 14.1.1.1 Show that a `Reader` monad does not compose inside of an `Either` monad.

Solution The `flatMap` method's type signature cannot be implemented for the type constructor $F^A \stackrel{\text{def}}{=} Z + (R \to A)$, where R and Z are fixed (but otherwise unknown) types. Begin writing the code:

```
1 def flatMap[A, B](fa: Either[Z, R => A])(f: A => Either[Z, R => B]): Either[Z, R => B] = fa match {
2   case Left(z)     => Left(z)      // No other choice here.
3   case Right(h)    => Right(r =>   // Cannot return a Left() here, since we have no values of type Z.
4       val x: Either[Z, R => B] = f(ra(r))      // Can compute a value ‘x‘ of type Either[Z, R => B].
5       ???                  )       // Need to return a value of type B here, using only r, ra, and f.
6 }
```

In line 3, we must decide whether to return a `Left(...)` or a `Right(...)`, given only values $h^{:R\to A}$ and $f^{:A\to Z+(R\to B)}$. The decision must be hard-coded since at that point we have no values of type R or A, so we cannot apply the functions f and h to any arguments. The only possible decision is to return a `Right(...)`. But then we are not able to compute a value of type B in the scope of lines 3–5 because we cannot guarantee that the value $x^{:Z+(R\to B)}$ contains a function $R \to B$ rather than a value of type Z. So, the typed hole $???^{:B}$ in line 5 cannot be filled. □

When the monad composition works both ways, the two resulting monads are usually not equivalent: the effects combined in different order behave differently. For example, the `Writer` monad $W \times A$ ("appending a log message") and the `Either` monad $Z + A$ (allowing failure) can be composed in two ways. The monad $M^A \stackrel{\text{def}}{=} W \times (Z + A)$ describes a computation that may fail but always appends a message. The monad $M^A \stackrel{\text{def}}{=} Z + W \times A$ will not append any message on failure.

Throughout this chapter, we will see other examples of monads whose effects can be combined in different order. The programmer needs to choose carefully the order of composition, according to the desired behavior of the effects.

Monad M_1^A	**Monad** M_2^A	$M_1 \circ M_2$	$M_2 \circ M_1$
Future^A	$Z + A$	Future^{Z+A}	$\cancel{Z + \text{Future}^A}$
`Future[A]`	`State[S, A]`	~~`Future[State[S, A]]`~~	~~`State[S, Future[A]]`~~
`State[S, A]`	$Z \to A$	~~`State[S, Z => A]`~~	`Z => State[S, A]`
`State[S, A]`	$Z + A$	`State[S, Either[Z, A]]`	~~`Either[Z, State[S, A]]`~~
List^A	$Z \to A$	$\cancel{\text{List}^{Z\to A}}$	$Z \to \text{List}^A$
List^A	$Z + A$	List^{Z+A}	$\cancel{Z + \text{List}^A}$
$\mathbb{1} + A$	$Z \to A$	$\cancel{\mathbb{1} + (Z \to A)}$	$Z \to \mathbb{1} + A$
$W \times A$	$Z + A$	$W \times (Z + A)$	$Z + W \times A$

Table 14.1: Correct and incorrect compositions of monads.

When functor composition works, implementing the `up` methods is straightforward:

Example 14.1.1.2 Implement a composition of `List` and `Try` monads for use in functor blocks.

Solution Since `Try` is a pass/fail monad, we must compose it *inside* other monads. So, we define the combined monad type as `List[Try[A]]`. We wrap the type `List[Try[A]]` into a class, implement `map` and `flatMap` as methods, and then define the two `up` functions (the "lifts") as extension methods:

```
final case class ListTry[A](value: List[Try[A]]) {
  def map[B](f: A => B): ListTry[B] = ListTry(value.map(_.map(f)))
  def flatMap[B](f: A => ListTry[B]): ListTry[B] = ListTry(
    value.flatMap {        // Need to provide a function Try[A] => List[Try[B]] here.
      case Failure(t)    => List(Failure(t))
      case Success(a)    => f(a).value
    })
}
def pure[A](a: A): ListTry[A] = ListTry(List(Success(a)))
```

```
implicit class ListTryLift1[A](l: List[A]) { def up: ListTry[A] = ListTry(l.map(Success(_))) }
implicit class ListTryLift2[A](t: Try[A]) { def up: ListTry[A] = ListTry(List(t)) }
```

After these definitions, we can write functor blocks that use `List` and/or `Try` as needed:

```
scala> for {
         x <- List(1, 2, 3).up
         y <- Try(x + 100).up
       } yield y
res0: ListTry[Int] = ListTry(List(Success(101), Success(102), Success(103)))
```

The following examples show that the functor composition does not always work. For instance, the `State` monad does not always compose with other monads in either order.

Example 14.1.1.3 Show that the `State` monad fails to compose inside the `Option` monad.

Solution There is no lawful `flatMap` function for the type constructor `Option[State[S, A]]`:

```
type OpSt[A] = Option[S => (A, S)]                // Here, the fixed type S must be already defined.
def flatMap[A, B](fa: OpSt[A])(f: A => OpSt[B]): OpSt[B] = fa match {
  case None      => None         // No other choice here.
  case Some(sta) => Some( s =>   // If we return ‘None‘ here, we will violate the monad identity law.
        // We need to return a value of type (S, B) in this scope, using given values f, sta, and s.
    val (s1, a) = sta(s) // We can compute another value of type s. But can we get a value of type B?
    val b: B = f(a) match {     // The only way of possibly getting ‘b: B‘ is to apply ‘f‘ to an ‘A‘.
        case Some(stb)   => stb(s2)._2                          // Here, we can get a value of type B.
        case None        => ??? // We are stuck: there is no way of returning a value of type B here.
    }
    ???                 )
}
```

Example 14.1.1.4 Show that the `State` monad fails to compose outside the `Reader` monad.

Solution Begin by implementing `pure` and `flatMap` for the type constructor `State[S, R => A]`:

```
type StR[A] = S => (R => A, S) // Here, the fixed types R and S must be already defined.
def pure[A](a: A): StR[A] = { s => (_ => a, s) }
def flatMap[A, B](fa: StR[A])(f: A => StR[B]): StR[B] = { s =>  // Need a value of type (R => B, S).
  val (s1, ra): (S, R => A) = fa(s)                  // We now have two values (s and s1) of type S.
  val rb: R => B = { r =>  // We can compute a value of type R => B. Trying to preserve information.
    val a: A = ra(r)
    val (s2, rb): (S, R => B) = f(a)(s1)
    rb(r)   // We must discard s2 here. This loses information since we could not use ‘f‘ elsewhere.
  }
  (rb, s1)  // We can only return (rb, s1) or (rb, s) here, since ‘s2‘ is not visible in this scope!
}
```

There is a problem with this implementation: The new state value s_1 is computed without using the function argument f. The only usage of f (producing s_2) is hidden within the scope of the nameless function of type $R \to B$. So, even though we have tried to preserve information as much as possible, the implementation of `flatMap` must discard the second monad's effect. This will violate the identity law `pure(x).flatMap(f) == f(x)`. That law cannot possibly hold if the implementation of `flatMap` ignores the function f when computing the new state:

```
pure(x).flatMap(f)
  == flatMap { s => (_ => x, s) } (f)
  == { s =>
  val (ra, s) = (_ => x, s)
  val rb = ...
  (rb, s)
}
```

The result always has the initial unmodified `s` as the second element of the pair. However, f is an arbitrary function of type `A => StR[B]`, so `f(x): StR[B]` could equal the function `{ s => (... , p(x, s)) }`, where the value `p(x, s)` is not necessarily equal to `s`. Therefore, `pure(x).flatMap(f)` is not equal to `f(x)` for arbitrary functions `f`.

14.1.2 Combining monads via monad transformers

Not all monads combine with others via functor composition. But even for those that do, the method of Example 14.1.1.2 requires us to write custom wrapper classes for each *pair* of monads. The length

of the required code is then quadratic in the number of supported monads. "Monad transformers" is the name of an approach for organizing the monad-combining code where the code size is only linear in the number of monads.

In that approach, we fix the first monad L (the "**base**" monad) and write code that combines L with an *arbitrary* second monad M (the "**foreign**" monad). This implements a new monad denoted by T_L^M. It turns out that the code of T_L^M can be made fully parametric in the monad M: the `pure` and `flatMap` methods of T_L^M will use only M's monad methods (pu_M and flm_M) and will work in the same way for all foreign monads M. The monad T_L^M is called "the monad L's transformer applied to M". So, the **monad** L's **transformer** is the type constructor $T_L^{M,A}$ viewed as a function of its type parameters M and A. We say *the* transformer T_L^M (rather than *a* transformer) because, as a rule, there is only one way of defining T_L^M (but see Section 14.8.5 below for some exceptions to that rule).

To illustrate this approach, let us implement the transformer for the `Try` monad. We begin with Example 14.1.1.2 that combines `Try` with `List`, but we now view `Try` as the base monad and `List` as the foreign monad. The goal is to replace `List` by an arbitrary foreign monad M, so that the resulting code works in the same way for all monads M. Begin by commenting the code of Example 14.1.1.2 to indicate the usage of the `List` monad's methods `pure` and `flatMap`:

```
final case class ListTry[A](value: List[Try[A]]) { // We will use List.map, List.pure, List.flatMap.
  def map[B](f: A => B): ListTry[B] = ListTry(value.map(_.map(f)))           // Using List.map here.
  def flatMap[B](f: A => ListTry[B]): ListTry[B] = ListTry(
    value.flatMap {         // Using List.flatMap here.
      case Failure(t)   => List(Failure(t)) // This is List.pure(Failure(t)).
      case Success(a)   => f(a).value
    })
}
def pure[A](a: A): ListTry[A] = ListTry(List(Success(a)))          // This is List.pure(Success(a)).
```

It is now straightforward to generalize that code to an arbitrary foreign monad M:

```
final case class TryT[M[_]: Monad : Functor, A](value: M[Try[A]]) {
  def map[B](f: A => B): TryT[M, B] = TryT(value.map(_.map(f)))         // Using M's ‘map‘ here.
  def flatMap[B](f: A => TryT[M, B]): TryT[M, B] = TryT(
    value.flatMap {                                          // Using M's ‘flatMap‘ here.
      case Failure(t)   => Monad[M].pure(Failure(t)) // Use Monad[M].pure(x) instead of List(x).
      case Success(a)   => f(a).value
    })
}
def pure[A](a: A): ListTry[A] = ListTry(Monad[M].pure(Success(a)))     // Using M's ‘pure‘ here.
```

The "foreign lift" and the "base lift" also use the methods of the foreign monad M:

```
implicit class TryTLift[M[_]: Monad : Functor, A](m: M[A]) {
  def up: TryT[M, A] = TryT(m.map(Success(_)))
}
implicit class TryTBaseLift[M[_]: Monad : Functor, A](t: Try[A]) {
  def up: TryT[M, A] = TryT(Monad[M].pure(t))
}
```

After these definitions, we are able to combine `Try` with any other monad `M`, as long as an implicit values of type `Monad[M]` and `Functor[M]` are in scope. The code will look like this:

```
implicit val functorList: Functor[List] = ... // Create a Functor typeclass instance for List.
implicit val monadList: Monad[List] = ...     // Create a Monad typeclass instance for List.

scala> val result = for {
         x <- List(1, 2, 3).up
         y <- Try(x + 100).up
       } yield y
res0: TryT[List, Int] = TryT(List(Success(101), Success(102), Success(103)))
```

This example defines a type constructor denoted by $T_{\text{Try}}^{M,\bullet}$ that works parametrically in the foreign monad M. We can use $T_{\text{Try}}^{M,A}$ as a monad with respect to the type parameter A. (The foreign monad

M is the first type parameter in the transformer.) We will later implement transformers for other monads, such as T^M_{Reader}, T^M_{Writer}, T^M_{State}, etc. The advantage of the transformer approach is that the code for T^M_L needs to be written once for every base monad L, not for every pair of monads L, M.

14.1.3 Monad transformers for standard monads

While the code of `TryT` works, it leads to more questions. The code of `TryT` is specific to the `Try` monad and uses its internal structure (the `Failure` and `Success` subtypes). It is not obvious how to implement a monad transformer T^M_L for an arbitrary given base monad L. We expect the code for T^M_L's methods to be parametric in M but not in L. This is shown by the notation T^M_L where L is a *subscript*. Changing the base monad L to another monad will require a complete rewrite of T^M_L's code.

There is no general recipe for implementing a monad transformer T^M_L given the code for a monad L. Nevertheless, monad transformers are known for every monad defined by fully parametric code. Transformers for some monads are obtained via functor composition; for other monads, transformers need to be defined in special ways. Table 14.2 lists the types of some known monad transformers. Depending on the monad, the corresponding transformer is defined by composing inside the foreign monad (`Either`, `Writer`); by composing outside the foreign monad (`Reader`, `Sel`); by wrapping a recursive definition inside the foreign monad (`List`); and by mixing the foreign monad in a special way with the base monad (`Eval`, `State`, `Cont`).

Name	Base monad	Transformer type
`Reader`	$L^A \stackrel{\text{def}}{=} R \to A$	$T^{M,A}_L \stackrel{\text{def}}{=} R \to M^A$
`Either`	$L^A \stackrel{\text{def}}{=} Z + A$	$T^{M,A}_L \stackrel{\text{def}}{=} M^{Z+A}$
`Writer`	$L^A \stackrel{\text{def}}{=} A \times W$	$T^{M,A}_L \stackrel{\text{def}}{=} M^{A\times W}$
`List`	$L^A \stackrel{\text{def}}{=} \mathbb{1} + A \times L^A$	$T^{M,A}_L \stackrel{\text{def}}{=} M^{\mathbb{1}+A\times T^{M,A}_L}$
`Eval`	$L^A \stackrel{\text{def}}{=} A + (\mathbb{1} \to A)$	$T^{M,A}_L \stackrel{\text{def}}{=} M^{A+(\mathbb{1}\to M^A)}$
`State`	$L^A \stackrel{\text{def}}{=} S \to A \times S$	$T^{M,A}_L \stackrel{\text{def}}{=} S \to M^{A\times S}$
`Cont`	$L^A \stackrel{\text{def}}{=} (A \to R) \to R$	$T^{M,A}_L \stackrel{\text{def}}{=} (A \to M^R) \to M^R$
`Sel`	$L^A \stackrel{\text{def}}{=} (A \to R) \to A$	$T^{M,A}_L \stackrel{\text{def}}{=} (M^A \to R) \to M^A$

Table 14.2: Known monad transformers for some monads.

It is not easy to determine the type of a monad transformer.[1] A justification for the types shown in Table 14.2 is that they work and obey the required laws, while no simpler transformers are known. Later in this chapter, we will prove that those and other monad transformers are correct. For now, we accept the results of Table 14.2 and turn to the practical use of monad transformers.

The first task is to obtain the code of monad methods and lifts for the transformers in Table 14.2.

The type constructors for monad transformers are conventionally named `ReaderT`, `EitherT`, etc.

The `ReaderT` transformer The corresponding monad was already derived in Statement 10.2.8.5:

```
final case class ReaderT[M[_]: Monad : Functor, R, A](run: R => M[A]) {
  def map[B](f: A => B): ReaderT[M, R, B] = ReaderT(r => run(r).map(f))
  def flatMap[B](f: A => ReaderT[M, R, B]): ReaderT[M, R, B] = ReaderT {
    r => run(r)            // Type is M[A].
           .flatMap(f)     // Type is M[ReaderT[M, R, B]].
           .map(_.run(r))  // Type is M[B].
  }
}
def pure[M[_]: Monad, A](a: A): ReaderT[M, A] = ReaderT(_ => Monad[M].pure(a))
```

The "foreign lift" and the "base lift" for `ReaderT` are implemented by:

```
implicit class ReaderTLift[M[_]: Monad : Functor, R, A](m: M[A]) {
  def up: ReaderT[M, R, A] = ReaderT(_ => m) // This is the ‘pure‘ method of the Reader monad.
}
implicit class ReaderTBaseLift[M[_]: Monad : Functor, R, A](t: R => A) {
```

[1] The difficulty of that task is illustrated by the history of the `ListT` transformer in the Haskell standard library. The transformer `ListT` was implemented incorrectly in 2002; the code violated the monad laws. The faulty transformer remained in the library until 2007. See the discussion in https://wiki.haskell.org/index.php?title=ListT_done_right

```
  def up: ReaderT[M, R, A] = ReaderT(r => Monad[M].pure(t(r)))
}
```

The lifts are written in the code notation as:

$$\text{foreign lift}: \quad \text{flift} : M^A \to T^{M,A}_{\text{Reader}} \quad , \qquad \text{flift}\,(m^{:M^A}) \stackrel{\text{def}}{=} _^{:R} \to m = \text{pu}_{\text{Reader}}(m) \quad ,$$

$$\text{base lift}: \quad \text{blift} : (R \to A) \to T^{M,A}_{\text{Reader}} \quad , \qquad \text{blift}\,(t^{:R\to A}) \stackrel{\text{def}}{=} r^{:R} \to \text{pu}_M\,(t(r)) = t \,⨾\, \text{pu}_M \quad .$$

We have seen in Section 10.1.5 that getting data out of the `Reader` monad requires a runner (θ_{Reader}), which is a function that calls `run` on some value of type R (i.e., injects `Reader`'s dependency value). In later sections 10.1.7 and 10.1.9, we have seen other monads that need runners. Generally, a runner for a monad M is a function of type $M^A \to A$. So, we may expect that the foreign monad M could have its own runner ($\theta_M : M^A \to A$). How can we combine M's runner (θ_M) with `Reader`'s runner? Since the type of T^M_{Reader} is a functor composition of `Reader` and M, the runners can be used independently of each other. We can first run the effect of M and then run the effect of `Reader`:

$$(\theta_M^{\uparrow\text{Reader}} \,⨾\, \theta_{\text{Reader}}) : (R \to M^A) \to A \quad .$$

Alternatively, we can run `Reader`'s effect first (injecting the dependency) and then run M's effect:

$$(\theta_{\text{Reader}} \,⨾\, \theta_M) : (R \to M^A) \to A \quad .$$

These runners commute because of the naturality law of θ_{Reader}, which holds for any $f^{:A\to B}$:

$$f^{\uparrow\text{Reader}} \,⨾\, \theta_{\text{Reader}} = \theta_{\text{Reader}} \,⨾\, f \quad , \quad \text{so} \quad \theta_M^{\uparrow\text{Reader}} \,⨾\, \theta_{\text{Reader}} = \theta_{\text{Reader}} \,⨾\, \theta_M \quad .$$

The `EitherT` transformer is similar to `TryT` since the type `Try[A]` is equivalent to `Either[Throwable, A]`.

The `WriterT` transformer It is easier to begin with the `flatten` method for the transformed monad $T^A \stackrel{\text{def}}{=} M^{A\times W}$, which has type signature:

$$\text{ftn}_T : M^{M^{A\times W}\times W} \to M^{A\times W} \quad , \qquad \text{ftn}_T\left(t^{:M^{M^{A\times W}\times W}}\right) = ???^{:M^{A\times W}} \quad .$$

Since M is an unknown, arbitrary monad, the only way of computing a value of type $M^{A\times W}$ is by using the given value t somehow. The only way to get a value of type A wrapped in $M^{A\times W}$ is by extracting the type A from inside $M^{M^{A\ldots}\ldots}$. So, we need to flatten the two layers of M that are present in the type of t. However, we cannot immediately apply M's `flatten` method to t because t's type is not of the form M^{M^X} with some X. To bring it to that form, we use M's `map` method:

$$t \triangleright (m^{:M^{A\times W}} \times w^{:W} \to m \triangleright (p^{:A\times W} \to p \times w)^{\uparrow M})^{\uparrow M} : M^{M^{A\times W\times W}} \quad .$$

Now the type is well adapted to using both M's and `Writer`'s flatten methods:

$$\begin{aligned}
\text{ftn}_T\left(t^{:M^{M^{A\times W}\times W}}\right) &= t \triangleright (m^{:M^{A\times W}} \times w^{:W} \to m \triangleright (p^{:A\times W} \to p \times w)^{\uparrow M})^{\uparrow M} \triangleright \text{ftn}_M \triangleright (\text{ftn}_{\text{Writer}})^{\uparrow M} \quad , \\
\text{ftn}_T &= (m^{:M^{A\times W}} \times w^{:W} \to m \triangleright (p^{:A\times W} \to p \times w)^{\uparrow M})^{\uparrow M} \,⨾\, \text{ftn}_M \,⨾\, (\text{ftn}_{\text{Writer}})^{\uparrow M} \\
&= \text{flm}_M\,(m \times w \to m \triangleright (p \to p \times w)^{\uparrow M} \,⨾\, \text{ftn}_{\text{Writer}}^{\uparrow M}) \\
&= \text{flm}_M\,(m \times w \to m \triangleright (a \times w_2 \to a \times (w \oplus w_2))) \quad .
\end{aligned}$$

Translating this formula to Scala, we obtain the code of `flatMap`:

```
final case class WriterT[M[_]: Monad : Functor, W: Monoid, A](t: M[(A, W)]) {
  def map[B](f: A => B): WriterT[M, W, B] = WriterT(t.map { case (a, w) => (f(a), w) })
  def flatMap[B](f: A => WriterT[M, W, B]): WriterT[M, W, B] = WriterT(
    t.flatMap { case (a, w) => f(a).t.map { case (b, w2) => (b, w |+| w2) }
  })
}
```

The code of `pure` is straightforward:

```
def pure[M[_]: Monad, W: Monoid, A](a: A): WriterT[M, W, A] =
  WriterT(Monad[M].pure((a, Monoid[W].empty))
```

The "foreign lift" and the "base lift" for `WriterT` are:

```
def flift[A]: M[A] => M[(A, W)] = _.map { a => (a, Monoid[W].empty) }
def blift[A]: ((A, W)) => M[(A, W)] = { t => Monad[M].pure(t) }
```

$$\text{foreign lift}: \quad \text{flift} : M^A \to T^{M,A}_{\text{Writer}} \quad , \qquad m^{:M^A} \triangleright \text{flift} \stackrel{\text{def}}{=} m \triangleright (a \to a \times e_w)^{\uparrow M} \quad ,$$

$$\text{base lift}: \quad \text{blift} : \text{Writer}^{W,A} \to T^{M,A}_{\text{Writer}} \quad , \qquad \text{blift} \stackrel{\text{def}}{=} \text{pu}_M \quad .$$

Comparing the lifts for `ReaderT` and `WriterT`, we notice some common features of lifts for compositional transformers (`ReaderT`, `EitherT`, `WriterT`): one of the lifts is equal to a `pure` method, and the other is equal to a lifted `pure` method. We will see later that this pattern applies to all monad transformers defined via functor composition.

The `ListT` transformer is more complicated because `List` is a recursive type. To shorten the code, we use the `Option` type to represent $\mathbb{1} + A \times T^{M,A}_{\text{List}}$ in the recursive type $T^{M,A}_{\text{List}} \stackrel{\text{def}}{=} M^{\mathbb{1}+A\times T^{M,A}_{\text{List}}}$:

```
final case class ListT[M[_]: Monad : Functor, A](value: M[Option[(A, ListT[M, A])]])
```

The `List` monad was implemented in Example 10.2.3.3 using the list concatenation function `concat`; "flattening" a `List[List[A]]` means concatenating all nested lists. If the ordinary `List` were defined via `Option`, the code for `concat` would be:

```
final case class ListO[A](value: Option[(A, ListO[A])])
def concat[A](t1: ListO[A], t2: ListO[A]): ListO[A] = t1.value match {
  case None => t2
  case Some((a, tail)) => Some((a, concat(tail, t2))) // Recursive call.
}
```

We need to implement the analogous concatenation function for `ListT`. To visualize the type of `ListT`, consider an example where the foreign monad is $M^A \stackrel{\text{def}}{=} R \to A$. Expand a few steps of recursion:

$$T^{M,A}_{\text{List}} = R \to \mathbb{1} + A \times (R \to \mathbb{1} + A \times (R \to ...(R \to \mathbb{1} + A \times T^{M,A}_{\text{List}})...)) \quad .$$

This type describes an M-effect wrapped around each next value of type A in the list. To extract further values in the list, we need to run the nested effects.

This understanding helps us implement the concatenation function for `ListT`, called `comb`:

```
def comb[M[_]: Monad : Functor, A](t1: ListT[M, A], t2: ListT[M, A]): ListT[M, A] = ListT( for {
  first <- t1.value            // Run the first effect and get ‘first: Option[(A, ListT[M, A])]‘.
  rest  = first match {        // Compute ‘rest: M[Option[(A, ListT[M, A])]]‘.
          case None => t2.value
          case Some((a, tail)) => Monad[M].pure(Some((a, comb(tail, t2))))   // Recursive call.
  y     <- rest
} yield y)
```

We are now ready to write the code of `ListT`'s `flatMap` method:

```
final case class ListT[M[_]: Monad : Functor, A](value: M[Option[(A, ListT[M, A])]]) {
  def map[B](f: A => B): ListT[M, B] = ListT(t.map(_.map { // Use the map methods of M and Option.
    case (head, tail) => (f(head), tail.map(f))
  }))
  def flatMap[B](f: A => ListT[M, B]): ListT[M, B] = ListT(
    t.flatMap {    // Here we need a function of type Option[(A, ListT)] => M[Option[(A, ListT)]].
      case None => Monad[M].pure(None)
      case Some((head, tail)) => comb(f(head), tail).value       // Type is M[Option[(A, ListT)]].
  })
}
```

The code of `pure` creates a single-element list, which uses a pair of an $a^{:A}$ and an empty list:

```
def pure[M[_]: Monad, A](a: A): ListT[M, A] =
  ListT(Monad[M].pure(Some((a, ListT(Monad[M].pure(None))))))
```

The lifts for `ListT` are:

```
def flift[M[_]: Monad, A]: M[A] => ListT[M, A] = ListT(a => m.flatMap(ListT.pure(a).value))
def blift[M[_]: Monad, A]: List[A] => ListT[M, A] =
  _.foldRight(Monad[M].pure(None)){ case (a, tail) => Monad[M].pure(Some((a, blift(tail)))) }
```

$$\text{foreign lift}: \quad \text{flift} : M^A \to T^{M,A}_{\text{List}} \quad , \qquad m^{:M^A} \triangleright \text{flift} \stackrel{\text{def}}{=} m \triangleright (a \to \mathbb{0} + a \times \text{pu}_M(1+\mathbb{0}))^{\uparrow M} \quad ,$$

$$\text{base lift}: \quad \text{blift} : \text{List}^A \to T^{M,A}_{\text{List}} \quad , \qquad \text{blift} \stackrel{\text{def}}{=} \left| \begin{array}{c|c} & M^{\mathbb{1}+A\times T^{M,A}_{\text{List}}} \\ \hline \mathbb{1} & 1 \to \text{pu}_M(1+\mathbb{0}) \\ A \times \text{List}^A & a \times t \to \text{pu}_M(\mathbb{0} + a \times \overline{\text{blift}}(t)) \end{array} \right| \quad .$$

Finally, we turn to runners for `ListT`. List-like monads usually represent *several* useful results rather than a single result. So, they may not need runners of type $\text{List}^A \to A$. Also, a list may be empty, which makes a fully parametric function of type $\text{List}^A \to A$ impossible. However, the foreign monad M may come with its runner θ_M. We can use θ_M to run the M's effects and extract values from a `ListT` into an ordinary `List`. This is accomplished by a runner (call it `runListT`) that applies M's runner (θ_M) to a value of type `ListT[M, A]` and obtains a `List[A]`. The code may be written as:

```
def runListT[A](runner: M[A] => A)(listT: ListT[M, A]): List[A] = runner(listT.value) match {
  case None                 => List()
  case Some((head, tail))   => head :: run(runner)(tail)
}
```

However, this code will not always terminate. The reason is that the type of `ListT` contains a layer of M in every recursive step. So, extracting a value of type `List[A]` from a value of type `ListT[M, A]` involves applying θ_M once per list value. When M is of function type, a value of type `ListT[M, A]` is an "on-call" sequence whose elements are not available in advance but will be evaluated on demand when θ_M is applied. So, the total number of elements in `ListT` may not be known in advance and may be infinite. Here is an example of an "infinite" collection that uses the `Reader` monad as M:

```
def ascend(n: Int): ListT[Reader[Int, *], Int] = ListT(Lazy(k => Some((n, ascend(n + k)))))
```

Evaluating `ascend(n)` gives a well-defined value representing the sequence $[n, n+k, n+2k, ...]$. The parameter k is passed via the `Reader` monad's dependency injection. It is clear that `runListT` will not terminate when applied to such a sequence.

This example shows how to use `ListT` for creating effectful streams of unbounded length. If we need to run the effects of M for an unbounded-length stream, the target should be a stream type (i.e., an on-call or a lazy list) rather than an eagerly evaluated `List` type.

Another way of running the effects of a `ListT[M, A]` is to assume that `A` is a monoid type. For monoid types `A`, we can implement the type signature $\text{List}^A \to A$ as a standard `reduce` operation:

```
def reduce[A: Monoid]: List[A] => A = {
  case Nil            => Monoid[A].empty
  case head :: tail   => head |+| reduce(tail)
}
```

Similar code gives a runner `brun` that transforms `ListT[M, A]` into `M[A]` for monoid types `A`:

```
def brun[M[_]: Monad, A: Monoid](listT: ListT[M, A]): M[A] = listT.value.flatMap {
  case None                  => Monad[M].pure(Monoid[A].empty)
  case Some((head, tail))    => Monad[M].pure(head) |+| brun(tail) // Monoid M[A].
}
```

For this code to work, we need a monoid instance for M^A (see Exercise 10.2.9.13):

```
implicit def monoidMA[M[_]: Functor : Monad, A: Monoid]: Monoid[M[A]] = ...
```

However, it remains unknown[2] whether this special version of `brun` is a lawful monad morphism $T^M_{\text{List}} \rightsquigarrow M$ or even just a monoid morphism $T^{M,A}_{\text{List}} \to M^A$.

The `StateT` transformer The formula for that transformer's type, `StateT[M, S, A] = S => M[(S, A)]`, puts the foreign monad M at a specific place within the type of the `State` monad. Let us implement the monad methods for `StateT`:

```
final case class StateT[M[_]: Monad : Functor, S, A](run: S => M[(A, S)]) {
  def map[B](f: A => B): StateT[M, S, B] = StateT(s1 => run(s1).map { // Use the map method of M.
    case (a, s2) => (f(a), s2)
  })
  def flatMap[B](f: A => StateT[M, S, B]): StateT[M, S, B] = StateT(s1 => run(s1).flatMap {
      case (a, s2) => f(a).run(s2)                // Use the updated intermediate state s2.
  })
}
def pure[M[_]: Monad, S, A](a: A): StateT[M, S, A] = StateT(s => Monad[M].pure((a, s)))
```

$$T^{M,A}_{\text{State}} \stackrel{\text{def}}{=} T^A \stackrel{\text{def}}{=} S \to M^{A\times S} \quad , \qquad \text{pu}_T \stackrel{\text{def}}{=} a^{:A} \to s^{:S} \to \text{pu}_M(a \times s) \quad , \tag{14.1}$$

$$\text{flm}_T(f^{:A\to T^B}) \stackrel{\text{def}}{=} t^{:T^A} \to s_1^{:S} \to t(s_1) \triangleright \text{flm}_M(a \times s_2 \to f(a)(s_2)) \quad . \tag{14.2}$$

The implementations of the lift functions for `StateT` are:

```
def flift[M[_]: Functor, S, A](m: M[A]): StateT[M, S, A] = StateT(s => m.map { a => (a, s) })
def blift[M[_]: Monad, S, A](p: State[S, A]): StateT[M, S, A] = StateT(s => Monad[M].pure(p.run(s)))
```

$$\text{foreign lift}: \quad \text{flift} : M^A \to T^{M,A}_{\text{State}} \quad , \qquad m^{:M^A} \triangleright \text{flift} \stackrel{\text{def}}{=} s \to m \triangleright (a \to a \times s)^{\uparrow M} \quad ,$$

$$\text{base lift}: \quad \text{blift} : (S \to A \times S) \to T^{M,A}_{\text{State}} \quad , \qquad \text{blift} \stackrel{\text{def}}{=} p^{:S\to A\times S} \to p \,⨾\, \text{pu}_M \quad .$$

The `State` monad's runner (θ_{State}) substitutes a given initial state (`init`) and extracts the final value:

```
def runState[S, A](init: S)(p: State[S, A]): A = p.run(init)._1
```

$$\theta_{\text{State}}(i^{:S}) \stackrel{\text{def}}{=} p^{:S\to A\times S} \to i \triangleright p \triangleright \pi_1 \quad .$$

The transformer `StateT` does not have a general base runner of the form brun (θ_{State}): we cannot convert an *arbitrary* runner $\theta_{\text{State}} : \text{State}^{S,A} \to A$ into a function of type $T^{M,A}_{\text{State}} \to M^A$. Instead, we need to implement a specialized base runner that converts $T^{M,A}_{\text{State}}$ into M^A by running the `State` monad's effects while keeping the effects of the foreign monad M:

```
def brunStateT[M[_]: Functor, S, A](init: S)(t: StateT[M, S, A]): M[A] = t.run(init).map(_._1)
```

$$\text{brun}_{\text{State}}(i^{:S}) \stackrel{\text{def}}{=} t^{S\to M^{A\times S}} \to i \triangleright t \triangleright \pi_1^{\uparrow M} \quad .$$

If we need to run only the effects of M but keep the `State` monad's effect, we need the following code:

```
def frunStateT[M[_], S, A](runner: M[A] => A)(t: StateT[M, S, A]): State[S, A] = State { s =>
  val m: M[(A, S)] = t.run(s)
  runner(m)
}
```

$$\text{frun}_{\text{State}}(\theta^{:M^A\to A}) \stackrel{\text{def}}{=} t^{:S\to M^{A\times S}} \to (s \to s \triangleright t \,⨾\, \theta) = t^{:S\to M^{A\times S}} \to t \,⨾\, \theta \quad .$$

The `ContT` transformer For the continuation monad (`Cont[R, A]`), the transformer's type constructor `ContT[M, R, A]` has a peculiar formula where the foreign monad's constructor M wraps the type R but is not applied to any values of the main value type A. The code for the monad methods is:

```
final case class ContT[M[_], R, A](run: (A => M[R]) => M[R]) {
  def map[B](f: A => B): ContT[M, R, B] = ContT { (k: B => M[R]) => run(f andThen k) }
  def flatMap[B](f: A => ContT[M, R, B]): ContT[M, R, B] = ContT { (k: B => M[R]) =>
```

[2]This book does not have a proof. See Problem 15.3.2.11 for more details.

```
    val g: A => M[R] = a => f(a).run(k)
    run(g)
  }
}
def pure[M[_], R, A](a: A): ContT[M, R, A] = ContT(k => k(a))
```

Turning now to the code for lifts, we can easily write the "foreign lift":

```
def flift[M[_]: Monad, R, A](m: M[A]): ContT[M, R, A] = ContT { (k: A => M[R]) => m.flatMap(k) }
```

$$\text{flift} : M^A \to (A \to M^R) \to M^R = \text{flm}_M \quad .$$

However, implementing the "base lift" is impossible: the required type signature

$$\text{blift} : ((A \to R) \to R) \to (A \to M^R) \to M^R$$

has no implementation parametric in the monad M. The reason is that we cannot produce a value of type R out of M^R parametrically (some monads have no runners), and so we cannot convert $A \to M^R$ into $A \to R$. This prevents us from using the first argument of `blift` (a function of type $(A \to R) \to R$). Also, creating a value of type $(A \to M^R) \to M^R$ from scratch is impossible.

To produce an explicit example where `blift` cannot be implemented, we use the `curryhoward` library and choose a monad M^R that has no fully parametric runner, e.g., $M^R \stackrel{\text{def}}{=} Z + R$:

```
import io.chymyst.ch._

scala> def blift[R, A]: Cont[R, A] => ContT[Either[Int, *], R, A] = implement
Error:(43, 65) type ((A => R) => R) => (A => Either[<c>Int,R]) => Either[<c>Int,R] cannot be
    implemented
```

The absence of "base lift" for the continuation monad transformer means that we cannot use the continuation monad in a functor block as freely as other monads. We cannot lift an arbitrary given value of type `Cont[R, A]` into the transformed monad `ContT[M, R, A]`. Values of type `Cont[R, A]` may come from third-party libraries implemented in non-fully-parametric code. It may be impossible to rewrite that code to return `ContT[M, R, A]` with an arbitrary monad `M`. This deficiency of the continuation monad makes it unsuitable for certain applications.

The continuation monad also does not support foreign runners or base runners. The type signature of the method frun : $T^{M,A}_{\text{Cont}} \to T^{N,A}_{\text{Cont}}$ cannot be implemented with arbitrary monads M, N because the type $T^{M,A}_{\text{Cont}} \stackrel{\text{def}}{=} (A \to M^R) \to M^R$ is not covariant in the type parameter M. The base runner's type signature, brun : $((A \to M^R) \to M^R) \to M^A$, cannot be implemented because there is no way of constructing a value of type M^A out of a value of type M^R (the type A is arbitrary).

The `SelT` transformer Although the selector monad, $\text{Sel}^{Z,A} \stackrel{\text{def}}{=} (A \to Z) \to A$, resembles the continuation monad, its transformer is simpler because it consists of composing outside the foreign monad M (i.e., we replace A by M^A in the type of `Sel`). Only one implementation fits the type signatures:

```
final case class SelT[M[_]: Monad : Functor, Z, A](run: (M[A] => Z) => M[A]) {
  def map[B](f: A => B): SelT[M,Z,B] = SelT { (mbz: M[B] => Z) => run(ma => mbz(ma.map(f))).map(f) }
  def flatMap[B](f: A => Sel[M, Z, B]): Sel[M, Z, B] = SelT { (mbz: M[B] => Z) =>
    val amb: A => M[B] = a => f(a).run(mbz)
    val maz: M[A] => Z = ma => mbz(ma.flatMap(amb))
    run(maz).flatMap(amb)
  }
}
def pure[M[_]: Monad, Z, A](a: A): SelT[M, Z, A] = SelT(_ => Monad[M].pure(a))
```

Since $T^M_{\text{Sel}} = \text{Sel} \circ M$ is a functor composition, the lifts and the runners work as in the `Reader` monad:

```
def flift[A](m: M[A]): SelT[M, Z, A] = SelT(_ => m)   // Using the ‘pure‘ method of the Sel monad.
def blift[A](sel: Sel[Z, A]): SelT[M, Z, A] = SelT { (k: M[A] => Z) =>
  val az: A => Z = Monad[M].pure andThen k
  Monad[M].pure(sel(az))
}
```

The `EvalT` transformer The evaluation monad $\text{Eval}^A \stackrel{\text{def}}{=} A + (\mathbb{1} \to A)$ described in Section 10.1.8 represents a choice of eager or lazy evaluation. The corresponding transformer is defined by:

$$T^{M,A}_{\text{Eval}} \stackrel{\text{def}}{=} M^{A+(\mathbb{1}\to M^A)} \quad .$$

Let us implement the `EvalT` monad using `Either` and write the monad methods. The code for the `Eval` monad in Section 10.1.8 used a helper function `get: Eval[A] => A`. We see that `get` is `Eval`'s runner. A similar function (`brun`) plays the role of the "base runner" for `EvalT`: it runs `Eval`'s effect (converting lazy values into eager ones) but keeps M's effects intact. We use `brun` to implement `flatMap`:

```
final case class EvalT[M[_]: Monad : Functor, A](value: M[Either[A, Unit => M[A]]]) {
  def map[B](f: A => B): EvalT[M, B] = EvalT(value.map {
    case Left(a)     => Left(f(a))
    case Right(g)    => Right(_ => g(()).map(f))
  })
  def flatMap[B](f: A => EvalT[M, B]): EvalT[M, B] = EvalT(value.flatMap {
    case Left(a)    => f(a).value
    case Right(g)   => Monad[M].pure(Right { _ => g(()).flatMap(a => EvalT.brun(f(a).value)) })
  })
}
object EvalT {
  def pure[M[_]: Monad, A](a: A): EvalT[M, A] = EvalT(Monad[M].pure(Left(a)))
  def brun[M[_]: Monad, A]: EvalT[M, A] => M[A] = _.value.flatMap { // Either[A, Unit=>M[A]] => M[A]
    case Left(a)    => Monad[M].pure(a)
    case Right(g)   => g(())
  }
}
```

The lifts are implemented straightforwardly:

```
def flift[A](m: M[A]): EvalT[M, A] = EvalT(m.map(a => Left(a))) // ‘Left(a)‘ means ‘Eval.pure(a)‘.
def blift[A](e: Either[A, Unit => A]): EvalT[M, A] = EvalT(Monad[M].pure(e match {
  case Left(a)    => Left(a)
  case Right(g)   => Right { _ => Monad[M].pure(g(())) }
}))
```

Given a runner for M, we can run M's effects while keeping `Eval`'s effects intact:

```
def runEvalT[A](runner: M[A] => A)(t: EvalT[M, A]): Eval[A] = runner(t) match {
  case Left(a)     => Eager(a)                      // An eager value is ready.
  case Right(g)    => Lazy(_ => runner(g()))        // A lazy value remains lazy.
}
```

14.1.4 Combining more than two monads: monad stacks

How can we combine *three or more* monads in a single functor block? For example, we may want somehow to make the following code work:

```
for {   // Could this code compile?
  i <- List(1, 2, 3).up
  j <- Writer(...).up
  x <- Try(i + j + 100).up
  y <- Future(x / 2).up
  z <- Reader(...).up
} yield z
```

A monad transformer combines a base monad L with any given foreign monad M. The result is a new, "larger" monad T^M_L that can describe the effects of both L and M. To combine the monad T^M_L with a third monad (say, K), we could use K's transformer and obtain the monad $T^{T^M_L}_K$ that describes the effects of K, L, and M. This technique is known as making a **stack** of monads.

As an example, let us build a stack of `Reader`, `Writer`, and `State` monads. We set $K^\bullet = \text{Reader}^{R,\bullet}$, $L^\bullet = \text{Writer}^{W,\bullet}$, and $M^\bullet = \text{State}^{S,\bullet}$, and begin with T^M_L. Since the `Writer`'s transformer composes inside the foreign monad, we write $T^{\text{State},A}_{\text{Writer}} = S \to A \times W \times S$. Now we apply the `Reader`'s transformer, which composes outside the foreign monad:

$$T^{T^{\text{State},A}_{\text{Writer}}}_{\text{Reader}} = R \to S \to A \times W \times S \quad .$$

Monad L^A	Monad M^A	Monad T_L^M	Monad T_M^L	Same?
`Reader[R, A]`	`Either[E, A]`	$T_L^{M,A} = R \to E + A$	$T_M^{L,A} = R \to E + A$	Yes
`Reader[R, A]`	`Writer[W, A]`	$T_L^{M,A} = R \to A \times W$	$T_M^{L,A} = R \to A \times W$	Yes
`Reader[R, A]`	`List[A]`	$T_L^{M,A} = R \to \text{List}^A$	$T_M^{L,A} \stackrel{\text{def}}{=} R \to \mathbb{1} + A \times T_M^{L,A}$	No
`Reader[R, A]`	`State[S, A]`	$T_L^{M,A} = R \to S \to A \times S$	$T_M^{L,A} = S \to R \to A \times S$	Yes
`Either[E, A]`	`State[S, A]`	$T_L^{M,A} = S \to (E + A) \times S$	$T_M^{L,A} = S \to E + A \times S$	No
`Either[E, A]`	`Cont[R, A]`	$T_L^{M,A} = (E + A \to R) \to R$	$T_M^{L,A} = (A \to E + R) \to E + R$	No

Table 14.3: Examples of monads stacked in different orders.

This monad has an injected dependency of type R, an internally maintained state of type S, and an output value of type W. A value of type $R \to S \to A \times W \times S$ describes all three effects at once.

If the transformers `ReaderT` and `WriterT` are already implemented, we may write the Scala type:

```
type RWS[R, W, S, A] = ReaderT[WriterT[State[S, *], W, *], R, A] // Equivalent to R => S => (A, W, S)
```

A nested transformer is substituted into the *first* argument of the outer transformer, which has the type constructor kind. We use the "kind projector" plugin for type constructors such as `State[S, *]`.

Note that the transformer composition does not necessarily represent functor composition. For instance, the plain functor composition `Reader[R, Writer[W, State[S, A]]]` is not a monad.

We could stack the monads K, L, and M in a different order, e.g., $T_L^{T_M^K}$ or $T_M^{T_K^L}$. In general, different orders will produce inequivalent monads, although in certain cases the stacking order does not matter. Table 14.3 and the following examples illustrate stacking monads in different orders.

Example 14.1.4.1 Show that $T_{\text{State}}^{\text{Reader}}$ is equivalent to that of $T_{\text{Reader}}^{\text{State}}$.

Solution The types of the transformers are equivalent: the only difference is the flipped order of curried arguments. We will verify that the functions `pure` and `flatMap` in these transformers also differ only by the order of curried arguments. Begin by writing `flatMap` for the `Reader` and `State` monads:

$$\text{flm}_{\text{Reader}}(g^{:A\to R\to B}) = p^{:R\to A} \to r^{:R} \to (p(r) \triangleright g)(r) \quad ,$$
$$\text{flm}_{\text{State}}(g^{:A\to \text{State}^{S,B}}) = p^{:S\to A\times S} \to s_1^{:S} \to p(s_1) \triangleright (a \times s_2 \to g(a)(s_2)) \quad .$$

Now we write the code for $T_{\text{Reader}}^{\text{State}}$'s methods:

$$T_{\text{Reader}}^{\text{State},A} \stackrel{\text{def}}{=} T_1^A = R \to S \to A \times S \quad ,$$
$$\text{pu}_{T_1} = a^{:A} \to _^{:R} \to \text{pu}_{\text{State}}(a) = a^{:A} \to _^{:R} \to s^{:S} \to a \times s \quad ,$$
$$\text{flm}_{T_1}(f^{:A\to T_1^B}) = t^{:T_1^A} \to r^{:R} \to t(r) \triangleright \text{flm}_{\text{State}}(a^{:A} \to f(a)(r))$$
$$= t \to r \to t(r) \triangleright \left(p^{:\text{State}^{S,A}} \to s_1^{:S} \to p(s_1) \triangleright (a \times s_2 \to f(a)(r)(s_2))\right)$$
$$= t \to r \to s_1 \to t(r)(s_1) \triangleright (a \times s_2 \to f(a)(r)(s_2)) \quad .$$

The code for $T_{\text{State}}^{\text{Reader}}$ is found by using Eq. (14.2):

$$T_{\text{State}}^{\text{Reader},A} \stackrel{\text{def}}{=} T_2^A = S \to R \to A \times S \quad ,$$
$$\text{pu}_{T_2} = a^{:A} \to s^{:S} \to \text{pu}_{\text{Reader}}(a \times s) = a^{:A} \to s^{:S} \to _^{:R} \to a \times s \quad ,$$
$$\text{flm}_{T_2}(f^{:A\to T_2^B}) = t^{:T^A} \to s_1^{:S} \to t(s_1) \triangleright \text{flm}_{\text{Reader}}(a \times s_2 \to f(a)(s_2))$$
$$= t \to s_1 \to t(s_1) \triangleright \left(p \to r \to (p(r) \triangleright (a \times s_2 \to f(a)(s_2)))(r)\right)$$
$$= t \to s_1 \to r \to (t(s_1)(r) \triangleright (a \times s_2 \to f(a)(s_2)))(r) \quad .$$

We see that the codes indeed differ only by flipping the curried arguments $r^{:R}$ and $s_1^{:S}$. □

Example 14.1.4.2 Show that the monads $T^{\text{List}}_{\text{Reader}}$ and $T^{\text{Reader}}_{\text{List}}$ are not equivalent.

Solution The difference becomes clear once we write out the types:

$$T^{\text{List},A}_{\text{Reader}} \stackrel{\text{def}}{=} R \to \text{List}^A \quad , \qquad T^{\text{Reader},A}_{\text{List}} \stackrel{\text{def}}{=} R \to \mathbb{1} + A \times T^{\text{Reader},A}_{\text{List}} \quad .$$

In a value of type $T^{\text{List}}_{\text{Reader}}$, the injected dependency (of type R) is used to produce an "eager" list. The length of the resulting list, as well as all its elements, are computed at once from the injected value of type R. In contrast, a data structure of type $T^{\text{Reader}}_{\text{List}}$ represents a "lazy" (on-call) sequence: getting the value of each next element requires calling a function of type $R \to \mathbb{1} + A \times T^{\text{Reader}}_{\text{List}}$ (possibly, a different function each time). The resulting on-call sequence may be unlimited in length. □

The next example shows that effects of the two monads are not always independent within the larger transformed monad. An effect of one monad will sometimes "cancel" an effect of the other.

Example 14.1.4.3 Show that the monads $T^{\text{Either}}_{\text{State}}$ and $T^{\text{State}}_{\text{Either}}$ are not equivalent.

Solution The effect of $\text{State}^{S,A} = S \to A \times S$ is to produce an updated internal state (of type S) together with a result value of type A. Write the types of the transformed monads:

$$T^{\text{Either},A}_{\text{State}} = S \to E + A \times S \quad , \qquad T^{\text{State},A}_{\text{Either}} = S \to (E + A) \times S \quad .$$

A computation of type $T^{\text{Either},A}_{\text{State}}$ may fail with an error (of type E) and produce *no* new state. So, the `Either` monad is able to "cancel" the state-updating effect of the `State` monad. But a computation of type $T^{\text{State},A}_{\text{Either}}$ never fails to update the state (although it could fail to give a result of type A). Because of this difference, the two transformed monads cannot be equivalent. To show this more rigorously, set all type parameters A, E, S to the `Boolean` type and observe that the types $S \to E + A \times S$ and $S \to (E + A) \times S$ have a different number of distinct values: $|2 \to 2 + 2 \times 2| = 36$ but $|2 \to (2 + 2) \times 2| = 64$.

Transformers for stacks The stack $T^{T^M_L}_K$ is a monad whose code is parametric in M. So, we can decide to use it as the monad transformer *for the monad* T^L_K:

$$T^M_{T^L_K} \stackrel{\text{def}}{=} T^{T^M_L}_K \quad . \tag{14.3}$$

To combine $T^{T^M_L}_K$ with a new monad N, we write the stack $T^{T^{T^N_M}_L}_K$ (where T_M is M's transformer). With this technique, we may combine any number of monads in a monad stack, as long as the transformers for all monads are known (except for the last one).

Monad stacking can be viewed as a binary operation (which we denote by $\triangleleft$) that combines monads K and L into a new monad $K \triangleleft L \stackrel{\text{def}}{=} T^L_K$. This operation is associative *by definition*: the transformer for the monad T^L_K is defined via Eq. (14.3), which we can rewrite as:

$$(K \triangleleft L) \triangleleft M \stackrel{\text{def}}{=} K \triangleleft (L \triangleleft M) \quad .$$

A monad stack is then denoted unambiguously by, e.g., $K \triangleleft L \triangleleft M \triangleleft N$, without need for parentheses.

Example 14.1.4.4 Define type constructors for these monad stacks and for their transformers:

(a) `State[S, *]`$\triangleleft$`Eval`.

(b) `List`$\triangleleft$`Option`$\triangleleft$`Try`.

Solution **(a)** To define the type constructor for `State[S, *]`$\triangleleft$`Eval`, set $M = \text{Eval}$ in the formula for `State` monad's transformer:

$$T^{M,A}_{\text{State}} = S \to M^{A \times S} = S \to \text{Eval}^{A \times S} \quad .$$

Now translate the type notation into code, assuming that the type `S` is defined previously:

```
type StateEval[A] = S => Either[(A, S), Unit => (A, S)]
```

The transformer for this monad is denoted by `State[S, *]`$\triangleleft$`Eval`$\triangleleft M$, where M is an arbitrary foreign monad. By definition,

$$\text{State}^{S,\bullet} \triangleleft \text{Eval}^\bullet \triangleleft M^\bullet \stackrel{\text{def}}{=} \text{State}^{S,\bullet} \triangleleft (\text{Eval}^\bullet \triangleleft M^\bullet) \quad ,$$

so we need to start with the monad stack `Eval`$\triangleleft M$:

```
type EvalM[A] = M[Either[A, Unit => M[A]]]
```

Now we apply `State`'s transformer to the foreign monad `EvalM` and find:

```
type StateEvalT[M[_], A] = S => M[Either[(A, S), Unit => M[(A, S)]]]
```

This type constructor is the monad transformer of the monad $\text{State}^S \triangleleft \text{Eval}$.

(b) The stack `List◁Option◁Try` is defined as the `List` monad's transformer applied to the stack `Option◁Try`. So, we first define the type constructor for `Option◁Try`, which is a functor composition:

```
type OptionTry[A] = Try[Option[A]]
```

We now apply `ListT` to the monad `OptionTry`, which gives a recursive type definition:

```
final case class ListOptionTry[A](value: Try[Option[Option[(A, ListOptionTry[A])]]])
```

To define the transformer for the monad `ListOptionTry`, we need to start with the innermost nested monad (`Try`) and apply its transformer (`TryT`) to the foreign monad M:

```
type TryM[A] = M[Try[A]]   // TryT[M, A] is defined as M[Try[A]].
```

The next step is to apply `Option`'s transformer to the foreign monad `TryM`:

```
type OptionTryM[A] = M[Try[Option[A]]]  // OptionT[TryM, A] is defined as TryM[Option[A]].
```

Finally, we apply the `ListT` transformer to the last monad:

```
final case class ListOptionTryT[M[_], A](value: M[Try[Option[Option[(A, ListOptionTryT[M, A])]]]])
```

□

Does the stacking operation ($\triangleleft$) have a unit? We notice from Table 14.2 that each transformer T_L^M somehow mixes the foreign monad M with the type of the base monad L. In every example, if we set M to the identity monad Id (defined by $\text{Id}^A \stackrel{\text{def}}{=} A$), the transformed type will become equivalent to the base monad L. So, we may expect this property to hold for every transformer:

$$T_L^{\text{Id}} \cong L \quad , \quad \text{equivalently written as:} \qquad L \triangleleft \text{Id} \cong L \quad .$$

Intuitively, the identity monad has a "trivial effect". Combining a trivial effect with another monad should not change that monad's effects. So, it is also natural to *define* the monad transformer for the identity monad as equal to the foreign monad:

$$T_{\text{Id}}^M \stackrel{\text{def}}{=} M \quad , \quad \text{equivalently written as:} \qquad \text{Id} \triangleleft M \stackrel{\text{def}}{=} M \quad .$$

Due to these properties of the identity monad, the base monad L can be recovered as T_L^{Id} if we know the code for the transformer T_L^M. So, if we already have an implementation for a monad L's transformer T_L, we do not need to define the monad L's type and methods separately. This saves time when implementing monad code in libraries (but may decrease performance).

With these definitions, we find that the monad stacking operation ($\triangleleft$) resembles the binary operation of a monoid: it is associative and has a unit (the monad `Id`). However, the monad $L \triangleleft M$ is defined only if the monad L has a transformer. It is unknown[3] whether a transformer exists for every monad relevant to functional programming. The question is open (Problem 15.3.2.5) even if restricted to monads whose methods are implemented by fully parametric code.

Transformers certainly cannot exist for monads defined using code that executes side effects. Examples of that kind are the `Future` monad and other "task-like" monads that create new threads. Transformers also cannot exist for external side-effect monads, such as `IO` from the `cats-effect` library or `ZIO` from the `zio` library. The reason is that a transformer must allow a foreign monad to cancel the effects of the base monad, as we have seen in Example 14.1.4.3. But side effects often cannot be undone once they are run, because changes in the external world may be irreversible.[4]

[3] For a discussion, see https://stackoverflow.com/questions/24515876/

[4] For a discussion, see https://stackoverflow.com/questions/11792275/

Throughout this chapter, we will build transformers for every exponential-polynomial monad considered in this book (although some transformers will only partially satisfy the required laws).

14.1.5 A typeclass for monad transformers

Transformers such as `EitherT` and `StateT` are type constructors that depend on an arbitrary foreign monad M and have the methods `flift`, `blift`, and `frun` with known type signatures. We may specify these properties via a typeclass `MTrans[T]` that describes what is required for T to be a monad transformer. A simple definition of the monad transformer typeclass can look like this:

```
type Id[A] = A        // The identity monad's typeclass instance must be defined elsewhere.
trait MTrans[T[_[_]]] {    // The type parameter T is a type constructor with 2 parameters.
  def monadT[M[_]: Monad]: Monad[T[M, *]]       // A monad instance must exist for T[M, *].
  def flift[M[_]: Monad, A]: M[A] => T[M, A]
  def blift[M[_]: Monad, A]: Base[A] => T[M, A] = frun[Id, M, A](Monad[M].pure)
  def frun[M[_]: Monad, N[_]: Monad, A](phi: M[A] => N[A]): T[M, A] => T[N, A]
}
```

The transformer's code is parametric in the foreign monad M (but not in the base monad L). Although `blift` can be implemented via `frun` as $\text{blift}_L^M \stackrel{\text{def}}{=} \text{frun}_L^{\text{Id},M}(\text{pu}_M)$, it is convenient to keep all three methods defined separately in the typeclass.

Here is a schematic example of defining an `MTrans` typeclass instance for `ReaderT`:

```
final case class Reader[A](run: R => A)     // The fixed type R must be already defined.
final case class ReaderT[M[_], A](run: Reader[M[A]])
implicit val mTransReaderT = new MTrans[ReaderT] {
  def monadT[M[_] : Monad]: Monad[ReaderT[M, *]] = ...
  def flift[M[_] : Monad, A]: M[A] => ReaderT[M, A] = { ma => ReaderT(Reader(_ => ma)) }
  def frun[M[_] : Monad, N[_] : Monad, A](phi: M[A] => N[A]): ReaderT[M, A] => ReaderT[N, A] =
    t => ReaderT(Reader(r => phi(t.run.run(r))))
}
```

14.1.6 Lifts and runners for monad stacks

Working with a monad stack, such as $K \triangleleft L \triangleleft M \triangleleft N$, requires using the effects of all the monads K, L, M, N. The code should appear to be able to compose any combination of Kleisli functions, e.g.:

$$p^{:A\to(K\triangleleft L\triangleleft M\triangleleft N)^F} \stackrel{\text{def}}{=} f_1^{:A\to L^B} \diamond f_2^{:B\to N^C} \diamond f_3^{:C\to(L\triangleleft M)^D} \diamond f_4^{:D\to(K\triangleleft L)^E} \diamond f_5^{:E\to K^F} \quad ,$$

where the "universal" composition operation $\diamond$ should somehow apply to Kleisli functions involving any of the given monads or sub-stacks. We also expect to be able to run the effects of any one of the monads while keeping the effects of others. To achieve all that, we need to define the lifts and the runners for the monad stack.

Lifts Any monad contained in a stack, such as K^A, L^A, M^A, N^A, $(K\triangleleft L)^A$, $(L\triangleleft M)^A$, etc., can be lifted to the stack type $(K\triangleleft L\triangleleft M\triangleleft N)^A$ by some combination of lifts. Assume that we have the base lifts, foreign lifts, and the runners of the relevant transformers, such as:

$$\text{flift}_K^P : P \rightsquigarrow (K\triangleleft P) \quad , \quad \text{blift}_L^P : L \rightsquigarrow (L\triangleleft P) \quad , \quad \text{frun}_M^{P,Q}(f^{:P\rightsquigarrow Q}) : M\triangleleft P \rightsquigarrow M\triangleleft Q \quad , \quad \text{etc.},$$

for any monad P. Keep in mind that the transformer for $K\triangleleft L$ is defined by stacking the transformers for K and L. As an example, let us compute the lifts $P \rightsquigarrow K\triangleleft L\triangleleft P$ and $L\triangleleft M \rightsquigarrow L\triangleleft M\triangleleft P$:

$$\text{flift}_{K\triangleleft L}^P : P \rightsquigarrow K\triangleleft L\triangleleft P = \text{flift}_L^P \,\text{⨾}\, \text{flift}_K^{L\triangleleft P} \quad , \quad \text{blift}_{L\triangleleft M}^P : L\triangleleft M \rightsquigarrow L\triangleleft M\triangleleft P = \text{frun}_L^{M,M\triangleleft P}(\text{blift}_M^P) \quad .$$

Lifting a monad value from a given monad from the middle of the stack into the full monad stack can require composing several lifts. For example, consider the task of lifting a value of type M^A to the stack $(K\triangleleft L\triangleleft M\triangleleft N)^A$. We may first lift M^A to $(M\triangleleft N)^A$ using blift_M^N, then lift the result to

$(L \triangleleft M \triangleleft N)^A$ using $\text{flift}_L^{M \triangleleft N}$, and finally lift the last result to $(K \triangleleft L \triangleleft M \triangleleft N)^A$ via $\text{flift}_K^{L \triangleleft M \triangleleft N}$. So, we obtain the following function composition:

$$\left(\text{blift}_M^N \fatsemi \text{flift}_L^{M \triangleleft N} \fatsemi \text{flift}_K^{L \triangleleft M \triangleleft N}\right) : M^A \to (K \triangleleft L \triangleleft M \triangleleft N)^A \quad .$$

Equivalently, we may first lift M^A to $(L \triangleleft M)^A$ via flift_L^M, then lift to $(K \triangleleft L \triangleleft M)^A$ via $\text{flift}_K^{L \triangleleft M}$, and finally use $\text{blift}_{K \triangleleft L \triangleleft M}^N$, which is defined as:

$$\text{blift}_{K \triangleleft L \triangleleft M}^N : K \triangleleft L \triangleleft M \rightsquigarrow K \triangleleft L \triangleleft M \triangleleft N \stackrel{\text{def}}{=} \text{frun}_K^{L \triangleleft M, L \triangleleft M \triangleleft N}(\text{frun}_L^{M, M \triangleleft N}(\text{blift}_M^N)) \quad .$$

With these techniques, we can construct all required liftings in any monad stack, as long as every monad transformer in the stack has the methods `flift`, `blift`, and `frun`.

Transformers for certain monads (e.g., the continuation and the codensity monads) have neither the `frun` nor the `blift` methods. Also, task-like monads (such as `Future`) and the external side-effect monads (such as `IO` and `ZIO`) have no transformers at all. Values of those monads can be lifted to a monad stack only when the problematic monad is located at the deep end of the stack, so that its transformer is not needed. For example, the monad N at the end of the stack $K \triangleleft L \triangleleft M \triangleleft N$ can be lifted to the stack without need for N's transformer:

$$\text{flift}_{K \triangleleft L \triangleleft M}^N : N^A \to (K \triangleleft L \triangleleft M \triangleleft N)^A \stackrel{\text{def}}{=} \text{flift}_M^N \fatsemi \text{flift}_L^{M \triangleleft N} \fatsemi \text{flift}_K^{L \triangleleft M \triangleleft N} \quad .$$

For this reason, the external side-effect monads and the task-like monads are usually placed at the deep end of the stack. This workaround limits the use of those monads in stacks; for instance, a stack cannot contain both an `IO` and a `Future` monad. To provide adequate functionality for applications, the `IO` and `ZIO` monads include special methods for running concurrent computations on multiple threads, which makes it unnecessary to add a task-like monad to the same monad stack.

While we solved the lifting problem in principle for most monad stacks, it is impractical for the programmer to have to write manually a composition of several lifts for every source line in a functor block. Even if we implement all the lift functions as extension methods and abbreviate the type annotations as much as possible, the code would have to be written similarly to the fragment shown at left. If a monad stack $K \triangleleft L \triangleleft M \triangleleft N$ needs to be changed to $M \triangleleft L \triangleleft N \triangleleft K \triangleleft P$ because of new program requirements, all source lines in all functor blocks will have to be modified to accommodate the required changes in the lifts. Productive practical use of deep monad stacks needs better techniques. One such technique, the "lift relation", is described in Section 14.1.7. Another technique is an "MTL-style" monad stack (Section 14.1.8). Both techniques are based on creating special typeclasses that produce the required lifts automatically.

```
for {    // Using a stack K◁L◁M◁N.
  x <- valueM.blift[N].flift[L].flift[K]
  y <- valueK.blift[TL[TM[N]]]
  z <- valueL.blift[TM[N]].flift[K]
} yield ...
```

Runners Having computed a value of type, say, $(K \triangleleft L \triangleleft M \triangleleft N)^A$, we may want to run the effects of a chosen monad (say M) using a known runner $\theta_M : M^A \to A$ but keep the effects of other monads in the stack. If all the monad transformers in the stack support the foreign and base runners, we can run the effects of any chosen monad, e.g., like this:

$$\text{frun}_K^{L \triangleleft M \triangleleft N}(\text{frun}_L^{M \triangleleft N}(\text{brun}_M^N(\theta_M))) : K \triangleleft L \triangleleft M \triangleleft N \rightsquigarrow K \triangleleft L \triangleleft N \quad .$$

As in the case of lifts, the implementations of the runners depend on the order of monads in the stack and will have to be changed if the stack changes. It does not seem possible to define a "stack runner" parametric in the chosen monad, allowing the programmer simply to indicate which monad's effects need to be run. To see why, consider a monad stack of type `EitherT[E, ReaderT[R, Option, *], A]`. If we wish to run the effects of the `Reader` monad alone, the result type must be `EitherT[E, Option, A]`. But there is no simple way of obtaining that result type from the type of the stack: the type `EitherT[E, Option, A]` cannot be expressed as a type constructor `Q[Reader, ...]` where `Reader` is just one of the type parameters. This is because the type of the stack is not parametric in the type of the base monad.

As we will see in this chapter, some monad transformers do not support base runners at all. Monad stacks containing such transformers are still usable, but their effects will have to be run in a specific order, so that the missing base runners are not needed.

14.1.7 Constructing lifts via type relations

The "lift relation" technique creates a type relation that describes all the possible liftings within a given monad stack. Two monads P and Q are **in a lift relation** if there exists a monad morphism $P \rightsquigarrow Q$. If Q is itself a monad stack containing the monad P, we can implement the lift relation $P \rightsquigarrow Q$ by using the monad transformers from that stack. For instance, the monads L and $K \triangleleft L \triangleleft M$ are in a lift relation because we can construct a monad morphism $L \rightsquigarrow K \triangleleft L \triangleleft M$ as:

$$(\text{blift}_L^M \fatsemi \text{flift}_K^{L \triangleleft M}) : L \rightsquigarrow K \triangleleft L \triangleleft M \quad .$$

At the same time, the monads M and $K \triangleleft L$ will *not* be in a lift relation (for arbitrary monads K, L, M) since, in general, there will not be any monad morphisms $M \rightsquigarrow K$ or $M \rightsquigarrow L$, so we will have no way of lifting the effects of the monad M into the monad $K \triangleleft L$.

The lift relation is implemented as a type relation `MM[P, Q]` for the monads' type constructors (`P`, `Q`). A value of type `MM[P, Q]` will contain the required monad morphism between `P` and `Q`:

```
trait MM[P[_], Q[_]] { def up[A]: P[A] => Q[A] } // This function must be a monad morphism.
```

We define the lift relation recursively. Any monad is always in a lift relation with itself:

```
implicit def liftableId[P[_]]: MM[P, P] = new MM[P, P] { def up[A] = identity }
```

A stack $L \triangleleft M$ has two lifts, $L \rightsquigarrow L \triangleleft M$ and $M \rightsquigarrow L \triangleleft M$. A relation instance for the foreign lift is:

```
implicit def foreignMonadToTransformer[KT[_[_], _], M[_] : Monad]
  (implicit kt: MTrans[KT]): MM[M, KT[M, *]] = new MM[M, KT[M, *]] {
    override def apply[A]: M[A] => KT[M, A] = kt.flift[M, A]
}
```

It requires more work to define a relation instance for the base lift, because the type constructor for the base monad cannot be automatically derived from the transformer's type. To make the base monad known to the Scala compiler, we define another type relation that designates base monads of monad transformers:

```
final case class BaseMonad[L[_], KT[_[_], _]](iso: MM[L, KT[Id, *]])
```

It is assumed that the function `iso` should be a monad isomorphism. Declaring a relation instance of `BaseMonad` looks like this example:

```
implicit val readerBase: BaseMonad[Reader, ReaderT] = BaseMonad(new MM[Reader, ReaderT[Id, *]] {
    def apply[A]: Reader[A] => ReaderT[Id, A] = ReaderT[Id, A](_)
})
```

Now we can implement a relation instance for the base lift:

```
implicit def baseMonadToTransformer[KT[_[_], _], M[_] : Monad, L[_] : Monad]
  (implicit kt: MTrans[KT], isBase: BaseMonad[L, KT]): MM[L, KT[M, *]] = new MM[L, KT[M, *]] {
    override def apply[A]: L[A] => KT[M, A] = isBase.iso.apply andThen kt.blift[M, A]
}
```

The typeclass constraint `BaseMonad[L, KT]` permits the Scala compiler to direct the search for implicits towards finding the base monad of a given transformer.

Finally, given a stack $K \triangleleft M$ and a monad L, we may lift $L \rightsquigarrow K \triangleleft M$ if there is already a lift $L \rightsquigarrow K$:

```
implicit def monadToTransformer[KT[_[_], _], L[_] : Monad, M[_] : Monad]
  (implicit kt: MTrans[KT], lm: MM[L, M]): MM[L, KT[M, *]] = new MM[L, KT[M, *]] {
    override def apply[A]: L[A] => KT[M, A] = lm.apply andThen kt.flift
  }
```

This relation instance needs to have lower priority than the foreign and base liftings: the Scala compiler should look for simpler liftings before trying to find a relation instance recursively.

With appropriate syntax definitions and typeclass instances, we may now write code like this:

```
type Stack3[A] = EitherT[ReaderT[Option, *], A]            // A stack having 3 monads.
implicit val mTrEitherT: MTrans[EitherT] = ...
implicit val baseEither: BaseMonad[Either[E, *], EitherT] = ...
implicit val mTrReaderT: MTrans[ReaderT] = ...
implicit val baseReader: BaseMonad[Reader, ReaderT] = ...
def toEither(x: Int, r: Int): Either[E, Int] = ...

val resultStack3: Stack3[Int] = for { // Lift each value to Stack3 with a single ‘up‘.
  x <- Option(5).up[Stack3]
  r <- Reader(identity).up[Stack3]
  t <- toEither(x, r).up[Stack3]
} yield t
```

To implement the automatic lifts in this way, the programmer needs to create three implicit values for each monad: a `Monad` typeclass instance, a monad transformer (`MTrans`) typeclass instance, and an instance of the `BaseMonad` type relation. (So, the length of code is linear in the number of supported monad types.) Those instances can be defined as implicit values in a monad transformer library. Then the user may declare an arbitrary monad stack `MyStack` and use the `up[MyStack]` method as an automatic lift from any level in the stack to the top.

14.1.8 Constructing lifts via typeclasses for monad operations (“MTL-style”)

The MTL-style[5] monadic programming is another way of achieving automatic lifting of monadic values in arbitrary monad stacks. The MTL approach has two main ingredients:

- For a chosen set of “standard” monads (such as the `Reader` monad of type $\text{Reader}^{R,A} \stackrel{\text{def}}{=} R \to A$, the `State` monad of type $\text{State}^{S,A} \stackrel{\text{def}}{=} S \to A \times S$, etc.), define some typeclasses (`ReaderMonadOps`, `StateMonadOps`, etc.) encapsulating all the “effectful operations” specific to each monad. Refactor all monadic code to use only the operations from the provided typeclasses.

- Provide appropriate typeclass instances for all monad transformers and all their combinations. For instance, all monad stacks that contain the `State` monad will need to have a `StateMonadOps` typeclass instance. After that, all code written using `StateMonadOps`'s methods will automatically work under any monad stack that contains the `State` monad.

First, we need to define the “effectful operations” corresponding to each monad. As an example, consider the `State` monad ($\text{State}^{S,A}$). Its effect is to manipulate an internal state value (of type S): the state can be read, updated, and automatically passed on to the next computation. We may define the functions `get` and `set` that work with the internal state:

$$\text{get} : \text{State}^{S,S} \quad , \qquad \text{get} \stackrel{\text{def}}{=} s^{:S} \to s \times s \quad , \qquad \text{set} : S \to \text{State}^{S,1} \quad , \qquad \text{set} \stackrel{\text{def}}{=} s^{:S} \to _^{:S} \to 1 \times s \quad .$$

```
val program: State[S, A] = for {
  s1 <- get    // Get the initial state.
  s2  = f(s1)  // Do some computations.
  _ <- set(s2) // Assign the new state.
} yield g(s2)  // Compute a result value.
```

A `State`-monadic program can then be written as shown at left, without referring to the type signature $S \to A \times S$. The two functions `get` and `set` appear to be sufficient to manipulate the internal state in an arbitrary way. So, we regard these two functions as the “effectful operations” of the `State` monad. Any `State`-monadic program may be expressed via these two operations (together with the standard monad methods `map`, `flatMap`, and `pure`). To see this formally, consider an arbitrary monadic program $q : \text{State}^{S,A}$ and define functions f_1 and f_2 by:

$$f_1 : S \to A \stackrel{\text{def}}{=} q \,\text{⨾}\, \pi_1 \quad , \qquad f_2 : S \to S \stackrel{\text{def}}{=} q \,\text{⨾}\, \pi_2 \quad .$$

[5] The name MTL stands for “monad transformer library”.

These two functions compute the result value and update the state. Now we can express q equivalently as a monadic program involving only `map`, `flatMap`, and the operations `get` and `set`:

```
q == for {
  s1 <- get
  s2 = f2(s1)
  _ <- set(s2)
} yield f1(s1)
```

$$q = \text{get} \triangleright \text{flm}_{\text{State}}(s_1 \to f_2(s_1) \triangleright \text{set} \triangleright (_ \to f_1(s_1))^{\uparrow \text{State}}) \quad .$$

We may refactor any `State`-monadic code so that it does not directly refer to the type $S \to A \times S$ of the `State` monad but instead only uses the operations `get` and `set`. For instance, the refactored code will not apply a monadic value to an argument of type S.

Another example is the `Either` monad, for which we define an operation called `raise`:

$$\text{raise} : E \to E + A \quad , \qquad \text{raise} \stackrel{\text{def}}{=} e^{:E} \to e + \mathbb{0}^{:A} \quad .$$

Any `Either`-monadic program is a value of type $E + A$ and can always be expressed through `raise` and `pure`. We may then refactor any monadic code so that it uses only the `raise` operation but does not refer to the type $E + A$ of the `Either` monad. For instance, the refactored code will not apply pattern matching directly to monad values.

The second ingredient of the MTL-style programming involves "lifting" the monad operations to arbitrary monad stacks. For example, consider the `State` monad with the operations $\text{get} : \text{State}^{S,S}$ and $\text{set} : S \to \text{State}^{S,\mathbb{1}}$. For any monad stack P containing the monad $\text{State}^{S,\bullet}$, we need to define the "lifted" operations $\text{get}_P : P^S$ and $\text{set}_P : S \to P^{\mathbb{1}}$. By assumption, P contains $\text{State}^{S,\bullet}$, so a certain composition of lift functions will give a monad morphism $p : \text{State}^{S,\bullet} \rightsquigarrow P^\bullet$. Once p is known, the new operations get_P and set_P are expressed as:

$$\text{get}_P : P^S \stackrel{\text{def}}{=} \text{get} \triangleright p \quad , \qquad \text{set}_P : S \to P^{\mathbb{1}} \stackrel{\text{def}}{=} \text{set} \circ p \quad .$$

Similarly, the `raise` operation of the `Either` monad can be lifted to any monad stack P that contains the `Either` monad. For any such P, there exists a monad morphism $p : E + A \to P^A$, and the lifted operation (raise_P) is defined by:

$$\text{raise}_P : P^A \quad , \qquad \text{raise}_P \stackrel{\text{def}}{=} \text{raise} \triangleright p \quad .$$

Lifting operations from a monad to a stack usually requires having a lift function (p) that maps values of the specific monad to the stack. The lift function p will be different for each monad and each stack, and we would like to avoid implementing p by hand each time. In MTL-style programming, this is achieved by refactoring the code to use special typeclasses that encapsulate the effectful operations of all the required monads. For the `State` monad, the typeclass `StateMonadOps` is defined as

```
trait StateMonadOps[F[_], S] { // ‘F‘ is any monad that has ‘State‘-like operations.
  def get: F[S]
  def set: S => F[Unit]
}
```

We can now write `State`-monadic programs by using the `StateMonadOps` typeclass:

```
val ops = implicitly[StateMonadOps[F, S]] // Must be defined before.
val program: F[A] = for {
  s1 <- ops.get
  s2 = f2(s1)
  _ <- ops.set(s2)
} yield f1(s2)
```

This code does *not* assume that the monad `F` has the `State` monad's type signature, `S => (A, S)`. So, it will work unmodified with any monad `F` that has the required typeclass instance. The `State` monad itself, of course, admits an instance of `StateMonadOps`:

```
implicit val stateMonadOps: StateMonadOps[State[S, *], S] = new StateMonadOps[State[S, *], S] {
  def get: State[S, S] = State(s => (s, s))
  def set: S => State[S, Unit] = s => State(_ => ((), s))
}
```

For the `Either` monad, the typeclass `EitherMonadOps` is defined by:

```
trait EitherMonadOps[F[_], E] { // ‘F‘ is any monad that has ‘Either‘-like operations.
  def raise[A]: E => F[A]
}
```

An instance of this typeclass can be implemented for the `Either` monad:

```
implicit val eitherMonadOps: EitherMonadOps[Either[E, *], E] = new EitherMonadOps[Either[E, *], E] {
  def raise[A]: E => Either[E, A] = e => Left(e)
}
```

Once all monadic code is refactored to use the monad operations via typeclasses, it remains to define the relevant typeclass instances for monad stacks. After that, the refactored code will work unchanged under any monad stack; the effectful operations will be lifted automatically through appropriate typeclass functions. Let us see in more detail how this works for the `State` monad.

The goal is to implement a typeclass instance of `StateMonadOps` for any monad stack containing the `State` monad; a typeclass instance of `EitherMonadOps` for any stack containing the `Either` monad; and so on, for other monads. To proceed, we need a rigorous definition of what it means that a monad stack P "contains" some monad L. Intuitively, the stack P should have the form $P = M_1 \triangleleft M_2 \triangleleft ... \triangleleft L \triangleleft ... \triangleleft M_k$ for some monads $M_1, ..., M_k$. We can write a definition by induction:

- Base cases: P contains L when $P = L$ or $P = L \triangleleft Q$ for some other monad stack Q. (The stack $P = L$ can be seen as a special case of $P = L \triangleleft Q$ with $Q = \text{Id}$.) The lift function ($p_{L,P}$) is:

$$p_{L,P} : L \rightsquigarrow L \triangleleft Q \quad , \qquad p_{L,P} \stackrel{\text{def}}{=} \text{blift}_L^Q \quad .$$

- Inductive step: P contains L when $P = N \triangleleft Q$, where N is some base monad and Q is another monad stack that contains L. The lift function is:

$$p_{L,P} : L \rightsquigarrow N \triangleleft Q \quad , \qquad p_{L,P} \stackrel{\text{def}}{=} p_{L,Q} \fatsemi \text{flift}_N \quad ,$$

  where $p_{L,Q}$ is the lift function for the stack Q. (By the inductive assumption, $p_{L,Q}$ is known.)

As an example, we will implement a `StateMonadOps` instance for all monad stacks that contain a `State` monad. Begin with the base cases: The `State` monad itself already has a `StateMonadOps` instance. If we have a stack of the form `StateT[Q, S, *]` where `Q` is any monad, we use the base lift of the `StateT` transformer to implement the `StateMonadOps` typeclass:

```
implicit def stateTMonadOps[S, Q[_]: Monad] = new StateMonadOps[StateT[Q, S, *], S] {
  def get: StateT[Q, S, S] = StateT(s => Monad[Q].pure((s, s)))
  def set: S => StateT[Q, S, Unit] = s => StateT(_ => Monad[Q].pure(((), s)))
}
```

In the inductive step, the stack is of the form `OtherT[Q, *]` where "`OtherT`" is a monad transformer for the base monad N. By the inductive assumption, we already have a `StateMonadOps` typeclass instance for the monad `Q`. Using the `flift` method of the transformer `OtherT`, we write:

```
implicit def otherTMonadOps[OtherT[_[_], _], Q[_], S](implicit val qMonadOps: StateMonadOps[Q, S],
    otherT: MTrans[OtherT]): StateMonadOps[OtherT[Q, *]] = new StateMonadOps[OtherT[Q, *]] {
  def get: OtherT[Q, S] = otherT.flift(qMonadOps.get)
  def set: S => OtherT[Q, Unit] = qMonadOps.set andThen otherT]].flift
}
```

To use the MTL-style monad stacks in practice, one needs a library (such as `cats-mtl`[6]) that defines typeclasses with sufficiently many operations for each standard monad, so that any desired monadic program may be expressed through those operations, without using explicit monad types. The library should also implement instances of those typeclasses for all supported monad stacks.

While the MTL style does provide automatic liftings, some problems remain.

[6] See https://typelevel.org/blog/2018/10/06/intro-to-mtl.html

The first problem is the lack of general principles for the choice of effectful operations. We have seen that the operations `get` and `set` are sufficient to express any `State`-monadic programs. However, it may not be adequate to write code just with those two operations. As an example, look at *updating* the state using a given function of type $S \to S$. In many cases, updating is the same as `get`ting the current state value and immediately `set`ting a new value. But suppose that certain state updates must be handled in a special way: e.g., atomic updates in a concurrent program; updates wrapped in a database transaction; or updates that must be run on a designated processing thread (such as the thread that drives the GUI events). Special handling of updates cannot be implemented through `get` and `set`; a new effectful operation (`update`) must be provided to the programmer.

Monad	Operations
$\text{State}^{S,A}$	$\text{get} : \text{State}^{S,S}$ $\text{set} : S \to \text{State}^{S,\mathbb{1}}$ $\text{update} : (S \to S) \to \text{State}^{S,\mathbb{1}}$
$\text{Reader}^{R,A}$	$\text{ask} : \text{Reader}^{R,R}$ $\text{local} : R \times \text{Reader}^{R,A} \to \text{Reader}^{R,A}$
$\text{Either}^{E,A}$	$\text{raise} : E \to \text{Either}^{E,A}$ $\text{handle} : (E \to \text{Either}^{E,A}) \times \text{Either}^{E,A} \to \text{Either}^{E,A}$
List^{A}	$\text{empty} : \text{List}^{A}$ $\text{append} : \text{List}^{A} \times \text{List}^{A} \to \text{List}^{A}$ $\text{filter} : \text{List}^{A} \times (A \to \mathbb{2}) \to \text{List}^{A}$
$\text{Writer}^{W,A}$	$\text{log} : W \to \text{Writer}^{W,\mathbb{1}}$ $\text{collect} : \text{Writer}^{W,A} \to \text{Writer}^{W,A\times W}$ $\text{process} : \text{Writer}^{W,A\times(W\to W)} \to \text{Writer}^{W,A}$ $\text{clear} : \text{Writer}^{W,A} \to \text{Writer}^{W,A}$
$\text{Cont}^{R,A}$	$\text{abort} : R \to \text{Cont}^{R,A}$ $\text{callcc} : ((A \to \text{Cont}^{R,B}) \to \text{Cont}^{R,A}) \to \text{Cont}^{R,A}$

Table 14.4: Known operations for some monads.

Table 14.4 shows a number of operations for some monads.[7] The choice of operations seems to be heuristic and not following any principle or system. It is not easy to decide what operations will be useful or necessary in practical tasks involving a given monad. (Do we need a `filter` operation for the `Either` monad?)

Different libraries choose different sets of supported operations. However, adding a new operation to a monad in an MTL-style library means updating all typeclasses related to that monad. Users who cannot modify the library code will be unable to add new operations.

The second problem is the lack of a general method for lifting effectful operations to arbitrary monad transformers.[8] So far, the examples in this subsection were limited to operations with type signatures of the form L^A or $A \to L^B$. All operations of this form are lifted from the monad L to a monad stack P by composing with a lift function $p : L^A \to P^A$. However, some operations have significantly different type signatures and cannot be lifted to arbitrary monad stacks by composing with a lift. For instance, an operation called `local` (defined for the `Reader` monad) has the type signature $R \to \text{Reader}^{R,A} \to \text{Reader}^{R,A}$. In that type signature, some `Reader` values are used as arguments (that is, in a contravariant position). So, `local` cannot be converted to a function of type $R \to P^A \to P^A$ by composing with a lift function $p : \text{Reader}^{R,A} \to P^A$. Another example is the `Writer` monad's `clear` operation that has the type $\text{Writer}^{W,A} \to \text{Writer}^{W,A}$. Knowing only a lift function $p : \text{Writer}^{W,A} \to P^A$ is not sufficient for transforming `clear` into a function of type $P^A \to P^A$.

Lifting `clear` to a given monad stack P (say, $P = \texttt{State} \lhd \texttt{Writer}$ or $P = \texttt{Writer} \lhd \texttt{Cont}$) requires custom code. Let us see how to write that code. For the stack $P = \texttt{State} \lhd \texttt{Writer}$, we need to implement a function clear_P of type $P^A \to P^A$ given a function `clear` with type signature:

$$\text{clear} : \text{Writer}^{W,A} \to \text{Writer}^{W,A} = W \times A \to W \times A \quad .$$

[7] Operations were gathered from the `cats-mtl` documentation and the papers "*Monad transformers and modular interpreters*" (see `http://web.cecs.pdx.edu/~mpj/pubs/modinterp.html`) and "*Modular monad transformers*" (see `https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.219.5365`).

[8] M. Jaskelioff developed a technique that can lift effectful operations systematically to almost all monad stacks; see Section 14.9.4.

Since the stack type is $P^A \stackrel{\text{def}}{=} T^{\text{Writer},A}_{\text{State}} = S \to W \times A \times S$, the code for clear_P is:

$$\text{clear}_P : (S \to W \times A \times S) \to S \to W \times A \times S \quad , \qquad \text{clear}_P \stackrel{\text{def}}{=} p^{:S \to W \times A \times S} \to p \,\fatsemi\, (\text{clear} \boxtimes \text{id}^S) \quad .$$

For the stack $P = \texttt{Writer} \triangleleft \texttt{Cont}$, or more verbosely:

$$P^A \stackrel{\text{def}}{=} T^{\text{Cont},A}_{\text{Writer}} = (W \times A \to R) \to R \quad ,$$

the lifted operation clear_P of type $((W \times A \to R) \to R) \to (W \times A \to R) \to R$ is implemented as:

$$\text{clear}_P \stackrel{\text{def}}{=} p^{:(W \times A \to R) \to R} \to q^{:W \times A \to R} \to p(\text{clear} \,\fatsemi\, q) \quad .$$

These examples show that lifting an operation to a specific monad transformer will require custom code adapted to every combination of a monad and an operation. The length of required custom code is quadratic in the number of supported monads.

The third problem is the large number of typeclass instances that MTL libraries need to maintain. As we have seen in the example with `StateMonadOps`, just two implicit functions per monad type are in principle sufficient to define the operation typeclasses for all monad stacks. This would make the number of typeclass instances linear in the number of supported monads. However, to make the code run faster, libraries provide typeclass instances for every combination of monads (say, `Reader` with `State`, `Either` with `Reader`, and so on). This makes the required number of typeclass instances quadratic in the number of supported monads. Adding a new monad to an MTL library requires adding many new typeclass instances.

For these reasons, MTL libraries usually support only a limited number of monads and operations.

14.2 Laws of monad transformers

14.2.1 Motivation for the laws of lifts

A monad transformer creates a new monad that combines the effects of other monads. Our next goal is to derive the properties required for the new monad to work well in practice. So, let us look at the programmer's intuitions about monadic programs written using transformers.

To be specific, assume that we have combined two monads L and M into a transformed monad $T \stackrel{\text{def}}{=} L \triangleleft M \stackrel{\text{def}}{=} T^M_L$ and defined the necessary lifts, $\text{up}_L : L^A \to T^A$ and $\text{up}_M : M^A \to T^A$. After these definitions, a programmer should be able to write functor blocks with lifted values of L and M.

Programmers will certainly expect all previously accepted properties of functor blocks to remain valid after lifting all monadic values of types L and M to the transformed monad T. A monadic program in T could look like this:

```
for {
  i <- someValueL.up
  x <- someValueT(i)
  y <- anotherValueT(y)
  z <- someValueM(x).up
} yield z
```

This program combines values of L and M lifted to T with some directly available values of the monad T (say, obtained from other monadic programs). Refactoring such programs involves, as a special case, refactoring of functor blocks in the monad T alone. It follows that T must itself satisfy the monad laws, — the laws we derived in Chapter 10 by analyzing various functor block refactorings.

There are some additional code refactorings specific to lifted monadic values. Lifted values may occur before or after a source line with an arbitrary value of T. Similarly to the standard monadic refactorings, we consider three cases: a lifted `pure` method before a source line, a lifted `pure` method after a source line, and refactoring two adjacent lifted source lines into a separate functor block.

If L's `pure` method is used before a source line, we expect the following refactoring to work:

```
          // Anywhere inside a for/yield:
  y <- Monad[L].pure(x).up // Assume x: A
  z <- f(y)                // f: A => T[B]
```

```
// Must be equivalent to...
  y = x      // x: A
  z <- f(y) // f: A => T[B]
```

The corresponding law is `pure(x).up.flatMap(f) == f(x)` or:

$$\text{pu}_L \,⨾\, \text{blift} \,⨾\, \text{flm}_T(f) = f \quad .$$

If L's `pure` method is used after a source line, we expect the following code equivalence:

```
          // Anywhere inside a for/yield:
x <- t    // Assume t: T[A]
y <- Monad[L].pure(x).up
```

```
// Must be equivalent to...
x <- t    // Assume t: T[A]
y = x
```

The corresponding law is `t.flatMap(x => L.pure(x).up) == t` or:

$$\text{flm}_T(\text{pu}_L \,⨾\, \text{blift}) = \text{id} \quad .$$

The third law comes from refactoring a functor block when two adjacent source lines are lifted from L. We may equivalently put these lines in a functor block in L and then lift its result to T:

```
          // Anywhere inside a for/yield:
x <- p.up       // Assume p: L[A]
y <- q(x).up    // q: A => L[B]
```

```
// Must be equivalent to...
pq = for { x <- p; y <- q(x) } yield y
y <- pq.up // Lift a refactored block.
```

The corresponding law is `p.up.flatMap(x => q(x).up) == p.flatMap(q).up` or:

$$\text{blift} \,⨾\, \text{flm}_T(q \,⨾\, \text{blift}) = \text{flm}_L(q) \,⨾\, \text{blift} \quad .$$

The form of these three laws can be simplified if we rewrite them in terms of the Kleisli composition operations $\diamond_L$ and $\diamond_T$. Recall that `flatMap` is related to the Kleisli composition by:

$$f^{:A\to T^B} \diamond_T g^{:B\to T^C} = f \,⨾\, \text{flm}_T(g) \quad .$$

The first law is then written as:

$$\text{pu}_L \,⨾\, \text{blift} \,⨾\, \text{flm}_T(f) = (\text{pu}_L \,⨾\, \text{blift}) \diamond_T f \overset{!}{=} f \quad .$$

The second law is not of the form $f \,⨾\, \text{flm}\,(...)$, so let us pre-compose it with an arbitrary $f^{:A\to T^B}$:

$$f \,⨾\, \text{flm}_T(\text{pu}_L \,⨾\, \text{blift}) = f \diamond_T (\text{pu}_L \,⨾\, \text{blift}) \overset{!}{=} f \,⨾\, \text{id} = f \quad .$$

Pre-compose also the third law with an arbitrary function $p^{:A\to L^B}$:

$$p \,⨾\, \text{blift} \,⨾\, \text{flm}_T(q \,⨾\, \text{blift}) = (p \,⨾\, \text{blift}) \diamond_T (q \,⨾\, \text{blift}) \overset{!}{=} p \,⨾\, \text{flm}_L(q) \,⨾\, \text{blift} = (p \diamond_L q) \,⨾\, \text{blift} \quad .$$

The first two laws are now easier to interpret: they require the function $(\text{pu}_L \,⨾\, \text{blift})$ to be both a left and a right unit for the Kleisli composition $\diamond_T$. However, the monad laws of T already provide such a unit, namely pu_T. An associative operation can have only one two-sided unit, so $(\text{pu}_L \,⨾\, \text{blift})$ must be equal to pu_T. To see that, it is sufficient to write the equation:

$$(\text{pu}_L \,⨾\, \text{blift}) = (\text{pu}_L \,⨾\, \text{blift}) \diamond_T \text{pu}_T = \text{pu}_T \quad .$$

So, the first two laws are equivalent to a single **identity law**:

$$\text{pu}_L \,⨾\, \text{blift} = \text{pu}_T \quad .$$

The third law says that the post-composition with `blift` will map a Kleisli composition in L to a Kleisli composition in T. This is the **composition law** of lifts. This law is sometimes easier to verify when expressed through `flatten`, although the formulation of the law becomes less visually clear:

$$\text{blift} \fatsemi \text{blift}^{\uparrow T} \fatsemi \text{ftn}_T = \text{ftn}_L \fatsemi \text{blift} \quad .$$

$$\begin{tikzcd} L^{L^A} \arrow[r, "\text{blift}"'] \arrow[d, "\text{ftn}_L"'] & T^{L^A} \arrow[r, "\text{blift}^{\uparrow T}"'] & T^{T^A} \arrow[d, "\text{ftn}_T"] \\ L^A \arrow[rr, "\text{blift}"] & & T^A \end{tikzcd}$$

The identity and composition law enforce a certain compatibility of `blift` with the monad methods of L and T_L^M. Laws of this form were introduced in Section 10.3.4 where we considered monad morphisms $\phi : M \rightsquigarrow N$ between two monads M and N. So, we have now shown that the expected properties of `blift` in functor blocks are equivalent to the requirement that `blift` should be a monad morphism of type $L \rightsquigarrow T_L^M$.

Since we expect to be able to use the monads L and M equally well within a functor block, the same refactorings must be possible with values of type M instead of L. Thus, the laws of monad morphisms must hold also for the foreign lift function ($\text{flift}_L : M \rightsquigarrow T_L^M$).

Since monad morphisms are natural transformations (Statement 10.3.4.4), both `blift` and `flift` obey the corresponding naturality laws. The naturality laws can be also motivated by considering functor blocks where a source line with a lift occurs next to a non-source line. We will omit the details, keeping in mind that naturality laws always hold for fully parametric code.

14.2.2 Motivation for the laws of runners

A monad's runner ($\theta_M : M^A \to A$) extracts results of type A from monadic values of type M^A. The use of a runner will agree with programmer's intuition if the runner obeys the laws of monad morphisms (see Definition 10.3.4.1); a runner θ_M must be a monad morphism of type $M \rightsquigarrow \text{Id}$ (between M and Id, the identity monad). Heuristically, a runner "executes" or "runs" the effects of the monad M and delivers an effect-free value as a result.

How can we extract effect-free values from a monad stack that combines the effects of several monads? Let us try to create a runner $\theta_T : T_L^{M,A} \to A$ for the transformed monad T_L^M, provided that both L and M have known runners θ_L and θ_M.

As a first example, assume a compositional transformer such as $T_L^M = L \circ M$; this is the case for $L =$ `Reader` and some other monads. Given a runner θ_M, we may first run M's effects by using $\theta_M^{\uparrow L}$:

$$\theta_M^{\uparrow L} : L^{M^A} \to L^A \quad .$$

A value of type A can be now extracted by using L's runner. So, we may express θ_T as:

$$\theta_T \stackrel{\text{def}}{=} \theta_M^{\uparrow L} \fatsemi \theta_L \quad .$$

If we first run the effects of L and then those of M, the result will be the same: $\theta_L^{\uparrow L} \fatsemi \theta_M = \theta_M \fatsemi \theta_L$ by the (assumed) naturality of θ_L. So, the runners commute. This is intuitively to be expected because the two runners work on different effects that are separately present in a value of type T_L^M.

A similar situation occurs for pass/fail monads that have composed-inside transformers. We may run the effects of only one of the monads, or both if needed; the runners commute.

Now consider a non-compositional transformer such as `StateT`:

$$T_{\text{State}}^{M,A} = S \to M^{A \times S} \quad .$$

In this case, we cannot run the effects of `State` using θ_{State} or the effects of M using $\theta_M^{\uparrow \text{State}}$: the types do not match. Compare the types of `brun`, `frun`, θ_{State}, and $\theta_M^{\uparrow \text{State}}$:

$$\text{brun} : T_{\text{State}}^{M,A} \to M^A = (S \to M^{A \times S}) \to M^A \qquad \text{vs.} \quad \theta_{\text{State}} : (S \to A \times S) \to A \quad ,$$

$$\text{frun} : T_{\text{State}}^{M,A} \to \text{State}^{S,A} = (S \to M^{A \times S}) \to S \to A \times S \qquad \text{vs.} \quad \theta_M^{\uparrow \text{State}} : (S \to M^A \times S) \to S \to A \times S \quad .$$

We need to use new runners specially adapted to `StateT`, as shown in Section 14.1.3. The foreign runner (`frun`) is a function of θ_M and works for all foreign monads M. The base runner `brun`, however, is not a function of θ_{State} but has custom code. So, the commutativity of runners,

$$\text{frun}_{\text{State}}(\theta_M) \fatsemi \theta_{\text{State}}(i) \stackrel{?}{=} \text{brun}_{\text{State}}(i) \fatsemi \theta_M \quad ,$$

is no longer automatic and needs to be verified. Apply both sides to a value $t^{:T^{M,A}_{\text{State}}}$:

$$\begin{aligned}
\text{left-hand side}: \quad & t \triangleright \text{frun}_{\text{State}}(\theta_M) \fatsemi \theta_{\text{State}}(i) = t \triangleright (t \to t \fatsemi \theta_M) \fatsemi (t \to i \triangleright t \triangleright \pi_1) \\
\text{compute composition}: \quad & = i \triangleright (t \fatsemi \theta_M) \triangleright \pi_1 = i \triangleright t \fatsemi \theta_M \fatsemi \pi_1 \quad , \\
\text{left-hand side}: \quad & t \triangleright \text{brun}_{\text{State}}(i) \fatsemi \theta_M = t \triangleright (t \to i \triangleright t \triangleright \pi_1^{\uparrow M}) \triangleright \theta_M = i \triangleright t \fatsemi \underline{\pi_1^{\uparrow M} \fatsemi \theta_M} \\
\text{naturality law of } \theta_M: \quad & = i \triangleright t \fatsemi \theta_M \fatsemi \pi_1 \quad .
\end{aligned}$$

Both sides are now equal, which proves the commutativity of runners for `StateT`.

These examples motivate the requirement that a monad transformer should come equipped with a base runner $\text{brun}_L : T_L^M \rightsquigarrow M$ and a foreign runner $\text{frun}_L(\theta_M) : T_L^M \rightsquigarrow L$ as long as the monads L and M have runners θ_L and θ_M. The foreign runner must be a function of θ_M, but the base runner is not necessarily expressed as a function of θ_L. The effects of the base monad and of the foreign monad may be run in any order, and the results must be the same:

$$\text{brun}_L \fatsemi \theta_M = \text{frun}_L(\theta_M) \fatsemi \theta_L \quad .$$

Some monads M do not admit monad morphisms $M \rightsquigarrow \text{Id}$ but instead have useful monad morphisms $M \rightsquigarrow N$ with suitable target monads N. A monad morphism $M \rightsquigarrow N$ may partially run the effects of M and represent the results using N's effects. For instance, the continuation monad can be mapped to a task monad such as `Future`; the `Option` monad can be mapped to an `Either` monad.

When a foreign monad M requires a runner of the form $\phi : M \rightsquigarrow N$ with a target monad $N \not\cong \text{Id}$, we would like to be able to map the monad stacks accordingly as $T_L^M \rightsquigarrow T_L^N$, possibly running some of M's effects but keeping L's effects unchanged. It is natural to expect that the same foreign runner (frun_L) should be able to use ϕ instead of θ_M. We can then compute a map $\text{frun}_L(\phi) : T_L^M \rightsquigarrow T_L^N$ that runs the stacks as required.

When the base monad L requires a runner with a target monad K, we will similarly need a monad morphism $T_L^M \rightsquigarrow T_K^M$ that can partially run the effects of L. Since T_K^M depends on K non-parametrically, we cannot expect to have a "universal" base runner that uses $\phi : L \rightsquigarrow K$ as a parameter. The extended base runner $\text{brun}_{L,K} : T_L^M \rightsquigarrow T_K^M$ will work for all foreign monads M but needs to have custom code specially adapted to the monads L and K.

The pass/fail monads (such as `Option` and `Try`) show one example where a universal base runner exists. Recall that the monad transformers for pass/fail monads work by functor composition. If $L^A = \mathbb{1} + A$ and $K^A \stackrel{\text{def}}{=} E + A$ (where E is a fixed type representing error information), we can choose an arbitrary runner $\phi : L \rightsquigarrow K$ and compute the universal base runner as:

$$\text{brun}_{L,K}(\phi) : (T_L^M \rightsquigarrow T_K^M) \stackrel{\text{def}}{=} \phi^{\uparrow M} \quad .$$

Apart from the laws of monad morphisms, do `frun` and `brun` need additional properties? For intuition, recall the example with the continuation monad's runner (Section 10.1.9) that works by first mapping `Cont` $\rightsquigarrow$ `Future` and then `Future` $\rightsquigarrow$ `Id`. Denote the two runners by:

$$\phi_1 : \text{Cont}^{R,A} \to \text{Future}^A \quad , \qquad \phi_2 : \text{Future}^A \to A \quad .$$

The runner for `Cont` is then defined by $\theta_{\text{Cont}} \stackrel{\text{def}}{=} \phi_1 \fatsemi \phi_2$. Now consider how we would run a monad stack T_L^{Cont}. We may use the foreign runner with θ_{Cont} as:

$$\text{frun}_L(\theta_{\text{Cont}}) : T_L^{\text{Cont}} \rightsquigarrow L \quad .$$

We could also use those runners to map $T_L^{\text{Cont}} \rightsquigarrow T_L^{\text{Future}}$ and then $T_L^{\text{Future}} \rightsquigarrow T_L^{\text{Id}} \cong L$:

$$\text{frun}_L(\phi_1) : T_L^{\text{Cont}} \rightsquigarrow T_L^{\text{Future}} \quad , \qquad \text{frun}_L(\phi_2) : T_L^{\text{Future}} \rightsquigarrow L \quad .$$

Intuitively, we would expect that the composition of these two runners should be the same as when running T_L^{Cont} directly into L:

$$\text{frun}_L(\phi_1) \fatsemi \text{frun}_L(\phi_2) \stackrel{!}{=} \text{frun}_L(\theta_{\text{Cont}}) = \text{frun}_L(\phi_1 \fatsemi \phi_2) \quad .$$

The composition $\phi_1 ⨾ \phi_2$ of any two monad morphisms ϕ_1 and ϕ_2 is again a monad morphism (see Statement 10.3.4.5). This suggests the **composition law**:

$$\text{frun}_L(\phi_1) ⨾ \text{frun}_L(\phi_2) \overset{!}{=} \text{frun}_L(\phi_1 ⨾ \phi_2) \quad . \tag{14.4}$$

This law is similar to the composition law of `map`. By analogy, we also impose the **identity law**:

$$\text{frun}_L(\text{id}^{:M^A \to M^A}) \overset{!}{=} \text{id}^{:T_L^{M,A} \to T_L^{M,A}} \quad .$$

Finally, let us consider the usage of a monad stack T_L^M within a functor block. Typically, we would lift some values of types L^A or M^A to the stack; at the end of the computation, we may run some (or all) of the effects. There are two possible cases: we lift a monadic value and then run the effects of the same monad; or we run the effects of the other monad.

In the first case, we lift a value $m : M^A$ to the stack T_L^M and then run the effects of M using $\text{frun}_L(\theta_M)$, getting a value of type L^A. The result is a transformation:

$$\text{flift}_L ⨾ \text{frun}_L(\theta_M) : M^A \to L^A \quad .$$

Note that both `flift` and `frun` are monad morphisms, so their composition is again a monad morphism $M \rightsquigarrow L$. Since M is an arbitrary foreign monad, we cannot expect the monad L to describe any effects of the foreign monad M. So, the result value of type L^A must have an empty L-effect, that is, it must be created from a value $a^{:A}$ via pu_L. The only way to obtain a value $a^{:A}$ is by applying a runner θ_M to the value $m : M^A$. So, we derived a "purity" law:

$$m ▷ \text{flift}_L ▷ \text{frun}_L(\theta_M) = m ▷ \theta_M ▷ \text{pu}_L \quad , \quad \text{or equivalently:} \quad \text{flift}_L ⨾ \text{frun}_L(\theta_M) = \theta_M ⨾ \text{pu}_L \quad .$$

An analogous purity law for `blift` and `brun` is:

$$\text{blift} ⨾ \text{brun} = \theta_L ⨾ \text{pu}_M \quad .$$

In the second case, we lift a foreign value $m : M^A$ to the stack, obtaining a value $t \overset{\text{def}}{=} \text{flift}\,(m)$, and then apply the *base* runner to t (running just the effects of L). This yields a monad morphism $M^A \to M^A$. We expect this morphism to be an identity function[9] because the monad M is arbitrary, and the code of `flift` and `brun` must be parametric in M. This requirement is written as the law:

$$m ▷ \text{flift} ▷ \text{brun} \overset{!}{=} m \quad , \quad \text{or equivalently:} \quad \text{flift} ⨾ \text{brun} \overset{!}{=} \text{id} \quad ,$$

called the **nondegeneracy law** of `brun`. The law requires `flift` and `brun` to preserve all information about M-effects. The analogous nondegeneracy law for `blift` and `frun` is:

$$\text{blift} ⨾ \text{frun}_L(\theta_M) = \text{id} \quad .$$

14.2.3 Simplifying the laws of lifts and runners via category theory

We have derived many laws of lifts and runners from heuristic considerations. Let us turn to category theory for guidance about obtaining a simplified and coherent picture of those laws.

The composition law (14.4) suggests that `frun` is a lifting from monad morphisms $M \rightsquigarrow N$ to monad morphisms $T_L^M \rightsquigarrow T_L^N$. In the language of category theory, a monad morphism is a morphism in the category of monads. The category of monads has all possible monads (M, N, ...) as its objects and all possible monad morphisms (of type $M \rightsquigarrow N$) as its morphisms.

Note that `frun` obeys the laws of identity and composition; these are the laws of a (categorical) functor. Looking at the law (14.4), we find that `frun` maps morphisms between monads (M, N, etc.) to morphisms between monad stacks with base L (such as T_L^M, T_L^N, etc.).

[9] Is there a non-identity monad morphism of type $M^A \to M^A$ that is parametric in the monad M? See Problem 15.3.2.7.

So, let us consider the category of monad stacks with a fixed base monad L. The objects of that category are all monad stacks T_L^M, T_L^N, etc., with the same base monad. The morphisms of that category are all monad morphisms between those monad stacks. (Since monad stacks are themselves monads, the category of monad stacks with base L is a sub-category of the category of monads.)

Observe that `frun` plays the role of the `fmap` function for a (categorical) functor from the category of monads to the category of monad stacks with base L. That functor (which we may denote by $T_L^\bullet$) maps each monad M to the stack T_L^M, and each monad morphism $\phi : M \rightsquigarrow N$ to a monad morphism $\text{frun}_L(\phi) : T_L^M \rightsquigarrow T_L^N$. We may view the functor $T_L^\bullet$ as the "bare" monad transformer, not yet applied to a foreign monad M.

Note that the functor T_L maps the monad M to the stack T_L^M; it maps objects of one category to objects of another. It is not a function from *values* of type M^A to values of type $T_L^{M,A}$. That function is `flift`. What is the role of `flift` in the category of monads? The function `flift` works in the same way for any foreign monad M (because the code of `flift` is parametric in the type constructor M). So, let us look at the relationship between the functions `flift` applied to different foreign monads.

Assume M and N are some monads for which a monad morphism $\phi : M^A \rightsquigarrow N^A$ exists. We may use `flift` to transform M^A into $T_L^{M,A}$ and to transform N^A into $T_L^{N,A}$. But we can also obtain a monad morphism $T_L^M \rightsquigarrow T_L^N$ as $\text{frun}_L(\phi)$. These monad morphisms can be drawn as a square diagram.

$$\begin{tikzcd} M^A \arrow[r, "\text{flift}"] \arrow[d, "\phi"'] & T_L^{M,A} \arrow[d, "\text{frun}_L(\phi)"] \\ N^A \arrow[r, "\text{flift}"] & T_L^{N,A} \end{tikzcd}$$

The diagram will commute if the following **monadic naturality law** holds:

$$\text{flift}_L^{M,A} \fatsemi \text{frun}_L(\phi^{:M^A \to N^A}) = \phi \fatsemi \text{flift}_L^{N,A} \quad . \tag{14.5}$$

Is it reasonable to require this law? To get some intuition, consider three special cases where the law is simplified, namely $M = \text{Id}$, $N = \text{Id}$, or $M = N$.

When $M = \text{Id}$, we must choose ϕ as the monad morphism $\phi \stackrel{\text{def}}{=} \text{pu}_N : \text{Id} \rightsquigarrow N$. (Exercise 14.9.5.4 shows that there are no other monad morphisms between Id and N.) The monad stack T_L^M is simplified to $T_L^{\text{Id}} \cong L$. The function $\text{flift}_L^{M,A}$ has type signature $\text{Id}^A \to T_L^{M,A}$, which is the same as $A \to L^A$. For a general monad L, there is no other function of this type except pu_L. So, $\text{flift}_L^{\text{Id}}$ must be equal to pu_L. The monad morphism $\text{frun}_L(\phi)$ has type signature $T_L^M \rightsquigarrow T_L^N$, which is the same as $L \rightsquigarrow T_L^N$, the type signature of `blift`. Since the monad N is arbitrary, it is reasonable to assume that:

$$\text{frun}_L(\phi) = \text{frun}_L(\text{pu}_N) = \text{blift}_L^N \quad .$$

In this way, the monadic naturality law (14.5) gives the equation:

$$\text{pu}_L \fatsemi \text{blift}_L^N = \text{pu}_N \fatsemi \text{flift}_L^N \quad .$$

This equation holds due to the identity laws of lifts:

$$\text{pu}_L \fatsemi \text{blift}_L^N = \text{pu}_{T_L^N} \quad , \qquad \text{pu}_N \fatsemi \text{flift}_L^N = \text{pu}_{T_L^N} \quad .$$

When $N = \text{Id}$, we choose $\phi = \theta_M$, assuming that M has a runner. The monad stack T_L^N is simplified to $T_L^{\text{Id}} \cong L$. The function flift_L^N has type signature $A \to L^A$ and is equal to pu_L. So, we get:

$$\text{flift}_L^M \fatsemi \text{frun}_L(\theta_M) = \theta_M \fatsemi \text{pu}_L \quad .$$

This is the "purity" law of `flift` that we derived earlier.

When $M = N$, we may choose $\phi = \text{id}$. The monadic naturality law (14.5) becomes:

$$\text{flift} \fatsemi \text{frun}_L(\text{id}) = \text{flift} \quad .$$

This equation holds as long as `frun` satisfies the identity law of monad morphisms ($\text{frun}_L(\text{id}) = \text{id}$).

So, we found that the law (14.5) agrees with previously derived laws and, moreover, covers one of the laws of runners. The interpretation of the monadic naturality law becomes clear from viewing

`frun` as a lifting ("`fmap`") corresponding to the functor T_L. Denoting that lifting temporarily by $^{\uparrow T_L}$, we rewrite the law (14.5) as:

$$\text{flift} \fatsemi \phi^{\uparrow T_L} = \phi \fatsemi \text{flift} \quad .$$

This is a standard form of a naturality law if we view $\text{flift}^M : M \rightsquigarrow T_L^M$ as a natural transformation between the identity functor $\text{Id}^M \stackrel{\text{def}}{=} M$ (in the category of monads) to the functor $T_L^\bullet$, the monad transformer functor with the base monad L. The law expresses the property that the code of `flift` works in the same way for all foreign monads M. We call it a "**monadic**" naturality law because $\text{flift}_L^{M,A} : M^A \to T_L^{M,A}$ also satisfies the ordinary naturality law with respect to the type parameter A, for a fixed monad M.

What other runners have monadic naturality properties? Suppose that the monad L's transformer has a base runner, $\text{brun}_L : T_L^M \rightsquigarrow M$. Viewed in the category of monads, this function is a natural transformation between the functor $T_L^\bullet$ and the identity functor (going in the opposite way compared with `flift`). The corresponding **monadic naturality law** is:

$$\text{brun}_L^M \fatsemi \phi^{:M \rightsquigarrow N} = \text{frun}_L(\phi) \fatsemi \text{brun}_L^N \quad .$$

To see whether it is reasonable to assume this law, let us derive some consequences of this law by making specific choices of M and N.

With $M = \text{Id}$, the function ϕ must be $\phi = \text{pu}_N$ (Exercise 14.9.5.4), while $\text{frun}_L(\phi) : T_L^{\text{Id}} \rightsquigarrow T_L^N$ is equal to the base lift function (blift_L^N). We also have $\text{brun}_L^{\text{Id}} : T_L^{\text{Id}} \rightsquigarrow \text{Id}$, so this function is equal to L's runner ($\text{brun}_L^{\text{Id}} = \theta_L$). The resulting equation is the purity law of `blift` and `brun`:

$$\theta_L \fatsemi \text{pu}_N = \text{blift}_L^N \fatsemi \text{brun}_L^N \quad .$$

With $N = \text{Id}$, the function ϕ must be a runner (θ_M), while $\text{brun}_L^{\text{Id}} = \theta_L$ as before. We get:

$$\text{brun}_L^M \fatsemi \theta_M = \text{frun}_L(\theta_M) \fatsemi \theta_L \quad .$$

This is the commutativity law of runners we obtained earlier.

So, the monadic naturality law of `brun` covers two of the previously formulated laws.

The monadic naturality law of `frun` is equivalent to `frun`'s composition law:[10]

$$\text{frun}_L(\phi^{:M \rightsquigarrow N}) \fatsemi \chi^{\uparrow T_L} = \text{frun}_L(\phi) \fatsemi \text{frun}_L(\chi) = \text{frun}_L(\phi \fatsemi \chi) \quad .$$

Since the category of monad stacks is a sub-category of the category of monads, T_L is an *endofunctor* in the category of monads, such that a natural transformation $\text{flift}_L : \text{Id}^M \rightsquigarrow T_L^M$ exists (in the sense of the category of monads, i.e., parametric in the monad M and obeying the monad morphism laws).

Endofunctors F^A having a natural transformation $\text{pu}_F : \text{Id}^A \to F^A$ are "pointed"; those with a natural transformation $\text{ex}_F : F^A \to \text{Id}^A$ are "co-pointed". Naturality forces the nondegeneracy law $\text{pu}_F \fatsemi \text{ex}_F = \text{id}$ (see Section 8.3.7). We find that T_L is a pointed endofunctor in the category of monads. If the transformer T_L has a base runner ($\text{brun}_L : T_L^M \rightsquigarrow \text{Id}^M$) then T_L will be also co-pointed.

It is remarkable that all the laws of monad transformers can be derived from a single (but more abstract) definition: a monad transformer T is *just a pointed and co-pointed endofunctor in the category of monads*.[11] Such a endofunctor T maps monads M, N, etc., to monads T^M, T^N, etc. Suppose we were given such a T but were not told that it is a monad transformer for some base monad. Still, we can recover its base monad L as the image of the identity monad:

$$L \stackrel{\text{def}}{=} T^{\text{Id}} \quad .$$

The definition of T already specifies functions `flift`, `frun`, and `brun` as the natural transformations:

$$\text{flift}_T^M : \text{Id}^M \rightsquigarrow T^M \quad , \qquad \text{brun}_T^M : T^M \rightsquigarrow \text{Id}^M \quad , \qquad \text{frun}_T^{M,N} : (M \rightsquigarrow N) \to T^M \rightsquigarrow T^N \quad .$$

The functor laws and the monadic naturality laws for those transformations will then enforce the 18 laws of monad transformers. In this way, a single definition expresses all monad transformer laws.

[10] This is similar to the fact that the naturality law and the composition law are the same for `fmap`.

[11] "*What's the big deal?*" as the joke would go.

Open question Does the nondegeneracy law $\text{flift}_T \,⨾\, \text{brun}_T = \text{id}$ follow from the monadic naturality of `brun` or other properties? This book does not have a proof. The composition $(\text{flift}_T \,⨾\, \text{brun}_T)$ is a natural transformation of type $\text{Id}^\bullet \rightsquigarrow \text{Id}^\bullet$; but it is a natural transformation *in the category of monads*. One would need to prove that there are no non-identity natural transformations between identity functors in the category of monads.

A natural transformation of type $\text{Id}^\bullet \rightsquigarrow \text{Id}^\bullet$ means, by definition, a family of monad morphisms $\varepsilon^M : M \rightsquigarrow M$ that work the same way for every monad M. One such morphism is the identity function, $\varepsilon = \text{id}$. However, it is not clear (Problem 15.3.2.7) whether there exist non-identity monad morphisms $M \rightsquigarrow M$ that work the same way for every monad M. For instance, Exercise 14.9.5.6 shows a failed attempt to define such a monad morphism.

Since we have not shown that the nondegeneracy law for base runners automatically follows from monadic naturality, we will need to verify that law separately whenever the base runner exists.

14.2.4 Summary of the laws of monad transformers

We have found that a **monad transformer for a base monad** L is a type constructor $T_L^{M,A}$ with a type parameter A and a monad parameter M, such that the following laws hold:

1. **Monad laws**: $T_L^{M,\bullet}$ is a lawful monad for any monad M. (The functor $T_L^{M,\bullet}$ has methods pu_T and ftn_T that satisfy the monad laws.)

2. **Identity laws**: $T_L^{\text{Id},\bullet} \cong L^\bullet$ via a monad isomorphism, where Id is the identity monad ($\text{Id}^A \stackrel{\text{def}}{=} A$).

3. **Lifting laws**: There is a function $\text{flift}_L : M^A \to T_L^{M,A}$ (in a shorter notation, $\text{flift}_L : M \rightsquigarrow T_L^M$) that works parametrically in the monad M and obeys the laws of monad morphisms.

4. **Runner laws**: There is a **foreign runner**[12] $\text{frun}(\phi)$ such that for any monads M, N and any monad morphism $\phi : M \rightsquigarrow N$, the function $\text{frun}(\phi) : T_L^M \rightsquigarrow T_L^N$ is a monad morphism. The function `frun` lifts monad morphisms from $M \rightsquigarrow N$ to $T_L^M \rightsquigarrow T_L^N$ and must satisfy the corresponding **functor laws**:

$$\text{frun}(\text{id}) = \text{id} \quad , \qquad \text{frun}(\phi) \,⨾\, \text{frun}(\chi) = \text{frun}(\phi \,⨾\, \chi) \quad .$$

   It follows from the identity law $T_L^{\text{Id}} \cong L$ that the base monad L can be lifted into T_L^M. Namely, by computing $\text{frun}(\phi)$ with $\phi \stackrel{\text{def}}{=} \text{pu}_M : \text{Id} \rightsquigarrow M$, we obtain a **base lift** function (`blift`):

$$\text{blift} : T_L^{\text{Id}} \rightsquigarrow T_L^M \cong L \rightsquigarrow T_L^M \quad , \qquad \text{blift} \stackrel{\text{def}}{=} \text{frun}(\text{pu}_M) \quad .$$

   The base lift and the foreign runner automatically satisfy a **nondegeneracy law**:

$$\text{blift} \,⨾\, \text{frun}(\phi^{:M \rightsquigarrow \text{Id}}) = \text{id} \quad ,$$

   for any monad morphism $\phi : M \rightsquigarrow \text{Id}$, because the left-hand side equals $\text{frun}(\text{pu}_M \,⨾\, \phi)$, and the composition law for monad morphisms gives $\text{pu}_M \,⨾\, \phi = \text{pu}_{\text{Id}} = \text{id}$.

5. **Base runner laws**: There is a **base runner**, a monad morphism $\text{brun}_L^M : T_L^M \rightsquigarrow M$ defined parametrically in the monad M. The base runner must also satisfy a **nondegeneracy law**:

$$\text{flift}_L^M \,⨾\, \text{brun}_L^M = \text{id} \quad .$$

   Since monad transformers are specific to their base monad and do not support a transition to another base monad, there are no functor laws for `brun`, unlike `frun`. Some monads have a general base runner, $\text{brun}(\theta_L)$, parameterized by an arbitrary runner $\theta_L : L \rightsquigarrow \text{Id}$. If so, the function $\text{brun}(\theta_L)$ must obey the base runner laws for any fixed θ_L.

[12] This function is called `hoist` in Haskell standard libraries and in `scalaz`.

6. **Monadic naturality laws**: The functions `flift` and `brun` must satisfy the monadic naturality law with respect to the monad parameter M. For an arbitrary monad morphism $\phi : M \rightsquigarrow N$,

$$\text{monadic naturality law of flift}: \quad \text{flift}^M \fatsemi \text{frun}\,(\phi^{:M\rightsquigarrow N}) = \phi^{:M\rightsquigarrow N} \fatsemi \text{flift}^N \quad ,$$
$$\text{monadic naturality law of brun}: \quad \text{brun}^M \fatsemi \phi^{:M\rightsquigarrow N} = \text{frun}\,(\phi^{:M\rightsquigarrow N}) \fatsemi \text{brun}^N \quad .$$

In this chapter, we will prove a stronger monadic naturality law of `flift` that requires $\phi : M^A \to N^A$ to be merely an arbitrary natural transformation, not necessarily a monad morphism.

In total, we found 18 laws for monad transformers. Are all these laws necessary?

The main use of the laws is to verify correctness of the code. The next section shows some examples of incorrect implementations of monad transformers and indicates the laws that are violated.

14.2.5 Examples of invalid monad transformers

The laws of monad transformers guarantee that the transformed monad is able to represent, without loss of information, the operations of the base monad as well as the operations of the foreign monad. If some of these laws are omitted, we risk accepting a transformer that has all the methods with the required type signatures but does not work correctly.

The simplest example of an invalid monad transformer is obtained by defining the new monad to be the unit monad ($T_L^{M,A} \stackrel{\text{def}}{=} \mathbb{1}$) for any monads L and M. This "fake transformer" cannot possibly describe the effects of L and M, because the methods of the unit monad discard *all* information and return 1. However, the type constructor $T_L^{M,A}$ has the required methods pu_T, ftn_T, flift_L, and frun_L (they are constant functions returning 1). All these functions are automatically monad morphisms, since a function from any monad to the unit monad is always a monad morphism. So, the "fake transformer" actually satisfies many of the monad transformer laws! However, the identity law $T_L^{\text{Id}} \cong L$ is violated since $T_L^{\text{Id}} = \mathbb{1} \ncong L$. For this reason, $T_L^{M,A} \stackrel{\text{def}}{=} \mathbb{1}$ is not a lawful monad transformer.

Other examples of incorrect "transformers" violating some of the laws are $T_L^M \stackrel{\text{def}}{=} L$ (no lifting law) and $T_L^M \stackrel{\text{def}}{=} M$ (no identity law).

These examples demonstrate the importance of the monad transformer laws. A wrong implementation of a "transformer" could fail to represent properly the effects of the monads, even though the transformer's methods with the correct type signatures are provided. The only way for us to detect the problem is to establish a violation of some of the required laws.

In these cases, it is intuitively clear that the transformer definitions are incorrect because the information about either L or M is missing in the type T_L^M. A valid definition of T_L^M must be a type constructor that somehow combines the type constructors L and M. Many such definitions are possible, but few will satisfy the monad transformer laws, as the next subsection shows.

14.2.6 Examples of failure to define a general monad transformer

No type construction is known that works in the same way for all monad pairs L, M and correctly combines the effects of both monads (i.e., satisfies the laws of monad transformers). We will now consider a few general ways of combining the type constructors L and M. In all cases, we will find that some monad transformer laws are violated.

Simple ways of combining two type constructors $L^\bullet$ and $M^\bullet$ are functor composition ($L^{M^\bullet}$ or $M^{L^\bullet}$), product ($L^\bullet \times M^\bullet$), and co-product ($L^\bullet + M^\bullet$).

Functor composition As we have seen, the functor composition works only for certain monads and only in a certain order, so it cannot serve as a general monad transformer. A simple counterexample is $L^A \stackrel{\text{def}}{=} \mathbb{1} + A$ and $M^A \stackrel{\text{def}}{=} A \times A$ where $M^{L^A} = (\mathbb{1} + A) \times (\mathbb{1} + A)$ is a monad but $L^{M^A} = \mathbb{1} + A \times A$ is not (see Exercise 10.2.9.9). Another counterexample is the `State` monad, $\text{State}_S^A \stackrel{\text{def}}{=} S \to S \times A$. The `State` monad does not compose with arbitrary monads M in either order. For instance, $\mathbb{1} + \text{State}_S^A$ and $\text{State}_S^{Z\to A}$ are not monads (see Examples 14.1.1.3 and 14.1.1.4).

Functor product We know from Chapter 10 that the functor product $M_1^A \times M_2^A$ is a monad when M_1 and M_2 are themselves monads. However, the product $M_1^A \times M_2^A$ describes two separate computations with two separate effects. Instead, we need a single computation with a combined effect. Formally, we find that there is no naturally defined flift : $M^A \to L^A \times M^A$ because we cannot create values of type L^A out of values of type M^A when monads L and M are arbitrary.

Functor co-product The co-product $L^\bullet + M^\bullet$ is in general not a monad when L and M are arbitrary monads. A counterexample is found by using two `Reader` monads, $L^A \stackrel{\text{def}}{=} P \to A$ and $M^A \stackrel{\text{def}}{=} Q \to A$: the co-product $(P \to A) + (Q \to A)$ is not a monad (Exercise 10.2.9.8). But even when $L^\bullet + M^\bullet$ is a monad, the identity law is violated: with $M = \text{Id}$, the monad $L + \text{Id}$ is not equivalent to L.

Using the free monad The functor composition $L^{M^\bullet}$ and the co-product $L^\bullet + M^\bullet$ may not always be monads, but they are always functors. We can make monads out of those functors by using the free monad construction. So, we obtain $\text{Free}^{L^{M^\bullet}}$, the free monad over $L^{M^\bullet}$, and $\text{Free}^{L^\bullet + M^\bullet}$, the free monad over $L^\bullet + M^\bullet$. Many laws of the monad transformers are satisfied by these constructions. However, the identity laws fail because:

$$\text{Free}^{L^{\text{Id}^\bullet}} \cong \text{Free}^{L^\bullet} \ncong L \quad , \qquad \text{Free}^{L^\bullet + \text{Id}^\bullet} \ncong L \quad .$$

The lifting laws are also violated because flift : $M^A \to \text{Free}^{L^\bullet + M^\bullet, A}$ is not a monad morphism: it maps pu_M into a non-pure value of the free monad. Nevertheless, these constructions are not useless. Once we run the free monad into a concrete (non-free) monad, we could arrange to hide the violations of these laws, so that the monad laws hold for the resulting (non-free) monad.

"Day convolution" The construction called **Day convolution** defines a new functor $L \star M$ as:

$$(L \star M)^A \stackrel{\text{def}}{=} \exists P. \exists Q. \ (P \times Q \to A) \times L^P \times M^Q \quad . \tag{14.6}$$

This formula can be seen as a modified application of the co-Yoneda identities:

$$L^A \cong \exists P. L^P \times (P \to A) \quad , \qquad M^A \cong \exists Q. M^Q \times (Q \to A) \quad .$$

The functor product $L^A \times M^A$ is equivalent to $\exists P. \exists Q. L^P \times M^Q \times (P + Q \to A)$:

$$\begin{aligned} & L^A \times M^A \\ \text{co-Yoneda identities for } L^A \text{ and } M^A : \quad & \cong \exists P. L^P \times \underline{(P \to A)} \times \exists Q. M^Q \times \underline{(Q \to A)} \\ \text{equivalence in Eq. (14.8)} : \quad & \cong \exists P. \exists Q. L^P \times M^Q \times (P + Q \to A) \end{aligned} \tag{14.7}$$

where we used the type equivalence:

$$(P \to A) \times (Q \to A) \cong P + Q \to A \quad . \tag{14.8}$$

If we (arbitrarily) replace $P + Q$ by $P \times Q$ in Eq. (14.7), we will obtain Eq. (14.6).

The Day convolution is associative[13] and always produces a functor since Eq. (14.6) is covariant in A. A simple example where the Day convolution fails to produce a monad transformer is $L^A \stackrel{\text{def}}{=} 1 + A$ and $M^A \stackrel{\text{def}}{=} R \to A$. We can compute the functor $L \star M$ explicitly:

$$\begin{aligned} & (L \star M)^A \\ \text{definitions of } L, M, \star : \quad & \cong \exists P. \exists Q. \underline{(P \times Q \to A)} \times (\mathbb{1} + P) \times (R \to Q) \\ \text{curry the arguments, move a quantifier} : \quad & \cong \exists P. (\mathbb{1} + P) \times \underline{\exists Q. (Q \to P \to A) \times (R \to Q)} \\ \text{co-Yoneda identity with } \exists Q : \quad & \cong \exists P. (\mathbb{1} + P) \times \left(\underline{R \to P} \to A\right) \\ \text{swap curried arguments} : \quad & \cong \exists P. (\mathbb{1} + P) \times (P \to R \to A) \\ \text{co-Yoneda identity with } \exists P : \quad & \cong \mathbb{1} + (R \to A) \quad . \end{aligned}$$

[13] https://en.wikipedia.org/wiki/Day_convolution

This functor is not a monad (see Exercise 10.2.9.9), so the transformer's monad law fails.

The Day convolution works because the type-level operations $P+Q$ and $P\times Q$ are both monoidal, in a suitable sense. We could generalize the Day convolution to use another monoidal type-level operation, such as $P+Q+Z\times P\times Q$ (where Z is a fixed type), instead of $P\times Q$. The result is a different "monoidal convolution" ($\star_Z$):

$$(L\star_Z M)^A \stackrel{\text{def}}{=} \exists P.\,\exists Q.\,(P+Q+Z\times P\times Q\to A)\times L^P\times M^Q \quad .$$

But this construction also fails to produce a lawful monad transformer.

Using a codensity monad The type constructor:

$$\text{Cod}^{K,A} \stackrel{\text{def}}{=} \forall B.\,(A\to K^B)\to K^B$$

is called the **codensity monad** on a type constructor K. The codensity monad provides yet another way of making a monad out of any type constructor. Can we then combine arbitrary monads L, M into a new monad? It turns out that the codensity construction does not provide enough properties to create a usable monad transformer. Here are some attempts:

(1) The codensity monad on $L\circ M$ yields the following type constructor:

$$F^A \stackrel{\text{def}}{=} \forall B.\,\left(A\to L^{M^B}\right)\to L^{M^B} \quad .$$

It has no lift $M^A\to F^A$. Intuitively, the reason is that we can use values of types M^A and $A\to L^{M^B}$ only in one way: using `map` to transform M^A into $M^{L^{M^B}}$. We cannot exchange the layers of M and L when those monads are arbitrary, and so we cannot obtain a value of type L^{M^B}. A simple counterexample is found with $L^A\stackrel{\text{def}}{=}\mathbb{1}+A$ and $M^A\stackrel{\text{def}}{=}Z\to A$ for a fixed type Z. The type signature:

$$(Z\to A)\to(A\to\mathbb{1}+(Z\to B))\to\mathbb{1}+(Z\to B)$$

can be implemented only as a function that returns a constant unit value ($1+\mathbb{0}^{:Z\to B}$), which loses information. Formally, a constant function would violate the identity law of monad morphisms.

(2) The composition of $\text{Cod}^{L,\bullet}$ and $M^\bullet$ yields the following type constructor:

$$F^A \stackrel{\text{def}}{=} \forall B.\,(M^A\to L^B)\to L^B \quad .$$

However, this F^A fails to be a monad for all L and M. A counterexample is $L^A\stackrel{\text{def}}{=}S\to A$ and $M^A\stackrel{\text{def}}{=}R\to A$, for which the Yoneda identity gives:

$$\begin{aligned}F^A &= \forall B.\,(\underline{(R\to A)\to S}\to B)\to S\to B\cong\forall B.\,(\underline{(R\to A)\times S}\to B)\to L^B\\ &\cong L^{(R\to A)\times S}=S\to(R\to A)\times S \quad .\end{aligned}$$

The last type constructor is not a monad: it is a composition of the `Reader` and `State` monads in the wrong order (see Statement 14.1.1.4).

(3) The **composed codensity monad** is defined by:

$$\text{Cod}_L^{M,A} \stackrel{\text{def}}{=} \forall X.\,(A\to L^X)\to L^{M^X} \quad .$$

This type constructor can be viewed as the continuation monad transformer for the base monad $\forall X.\,(A\to X)\to M^X\cong M^A$ (this equivalence is a Yoneda identity). Although $\text{Cod}_L^{M,A}$ is a lawful monad (Exercise 14.9.5.22), it cannot serve as a monad transformer because the lift $M^A\to\text{Cod}_L^{M,A}$ does not exist for some L and M. A simple counterexample is found with $M^A\stackrel{\text{def}}{=}R\to A$ and $L^A\stackrel{\text{def}}{=}\mathbb{1}+A$ similarly to the counterexample shown in (1).

14.2.7 Functor composition with transformed monads

Suppose we are working with a base monad L and a foreign monad M, and we have constructed the transformed monad T_L^M. In this section, let us denote the transformed monad simply by T.

The main feature of a monad transformer is that the monad T adequately describes the effects of both monads L and M at the same time. Suppose we are working with a deeply nested type constructor involving many functor layers of monads L, M, and T such as:

$$T^{M^{T^{L^{M^{L^A}}}}} = (T \circ M \circ T \circ L \circ M \circ L)^A \quad .$$

The properties of the transformer allow us to convert this type to a single layer of the monad T. In this example, we will have a natural transformation:

$$(T \circ M \circ T \circ L \circ M \circ L)^A \to T^A \quad .$$

To achieve this, we first use the methods `blift` and `flift` to convert each layer of L or M to a layer of T, lifting into functors as necessary. The result will be a number of nested layers of T. Second, we use ftn_T as many times as necessary to flatten all nested layers of T into a single layer. The result will be a value of type T^A, as required. Due to the laws of `blift` and `flift`, the result does not depend on the order in which the different layers are converted and flattened.

14.2.8 Stacking two monads. Proofs

Suppose we know the transformers T_P and T_Q for some given monads P and Q. We can transform Q with P and obtain a monad $R^A \stackrel{\text{def}}{=} T_P^{Q,A}$. The monad transformer T_R for the monad R is defined to be the following monad stack (see Section 14.1.4):

$$T_R^{M,A} \stackrel{\text{def}}{=} T_P^{T_Q^{M,\bullet},A} \stackrel{\text{def}}{=} (P \triangleleft Q \triangleleft M)^A \quad .$$

Here, it is important that the monad $T_Q^{M,\bullet}$ is substituted into $T_P^{M,A}$ as the foreign monad M (not as the type parameter A). This way of composition is called **stacking** the monad transformers.

In Scala code, this "stacking" construction is written as

```
type RT[M, A] = PT[QT[M, *], A]
```

The resulting monad is a **stack** of three monads P, Q, and M. The order of monads in the stack is significant: in general, there will be no monad isomorphism between differently ordered stacks.

We will now prove that the transformer T_R is lawful (satisfies the laws stated in Section 14.2.4), as long as both T_P and T_Q satisfy the same laws. To shorten the notation, we talk about a "monad T_P^M" meaning the monad defined as $T_P^{M,\bullet}$ or, more verbosely, the monad $G^A \stackrel{\text{def}}{=} T_P^{M,A}$.

Monad laws We need to show that $T_P^{T_Q^M}$ is a monad for any monad M. The monad laws for T_Q say that T_Q^M is a monad. The monad laws for T_P say that T_P^S is a monad for any monad S; in particular, for $S = T_Q^M$. Therefore, $T_P^S = T_P^{T_Q^M}$ is a monad.

Identity law We need to show that $T_P^{T_Q^{\text{Id}}} \cong T_P^Q$ via a monad isomorphism. The identity law for T_Q says that $T_Q^{\text{Id}} \cong Q$ via a monad isomorphism. So, we already have a monad morphism $\phi : Q \rightsquigarrow T_Q^{\text{Id}}$ and its inverse, $\chi : T_Q^{\text{Id}} \rightsquigarrow Q$. The runner frun_P for T_P can be applied to both ϕ and χ since they are monad morphisms. So, we obtain two new monad morphisms:

$$\text{frun}_P(\phi) : T_P^Q \rightsquigarrow T_P^{T_Q^{\text{Id}}} \quad ; \qquad \text{frun}_P(\chi) : T_P^{T_Q^{\text{Id}}} \rightsquigarrow T_P^Q \quad .$$

Are these two monad morphisms inverses of each other? To show this, we need to verify:

$$\text{frun}_P(\phi) \,⨾\, \text{frun}_P(\chi) = \text{id} \quad , \qquad \text{frun}_P(\chi) \,⨾\, \text{frun}_P(\phi) = \text{id} \quad .$$

By the runner laws for T_P, we have $\text{frun}_P\,(\text{id}) = \text{id}$ and $\text{frun}_P\,(f) \fatsemi \text{frun}_P\,(g) = \text{frun}_P\,(f \fatsemi g)$ for any two monad morphisms f and g. So, we have:

$$\begin{aligned}\text{frun}_P\,(\phi) \fatsemi \text{frun}_P\,(\chi) &= \text{frun}_P\,(\phi \fatsemi \chi) = \text{frun}_P\,(\text{id}) = \text{id} \quad ,\\ \text{frun}_P\,(\chi) \fatsemi \text{frun}_P\,(\phi) &= \text{frun}_P\,(\chi \fatsemi \phi) = \text{frun}_P\,(\text{id}) = \text{id} \quad .\end{aligned}$$

We have indeed obtained a monad isomorphism between T_P^Q and $T_P^{T_Q^{\text{Id}}}$.

Lifting law We need to show that there exists a monad morphism $\text{flift}_R : M \rightsquigarrow T_P^{T_Q^M}$ for any monad M. The lifting law for T_Q gives a monad morphism $\text{flift}_Q : M \rightsquigarrow T_Q^M$. The lifting law for T_P can be applied to the monad T_Q^M, which gives a monad morphism $\text{flift}_P : T_Q^M \rightsquigarrow T_P^{T_Q^M}$. The composition $\text{flift}_R \stackrel{\text{def}}{=} \text{flift}_Q \fatsemi \text{flift}_P$ has the required type $M \rightsquigarrow T_P^{T_Q^M}$ and is a monad morphism by Statement 10.3.4.5.

Runner law We need to show that there exists a lifting:

$$\text{frun}_R : (M \rightsquigarrow N) \to T_P^{T_Q^M} \rightsquigarrow T_P^{T_Q^N} \quad .$$

First, we have to define $\text{frun}_R(\phi)$ for any given $\phi : M \rightsquigarrow N$. We use the lifting law for T_Q to get a monad morphism:

$$\text{frun}_Q(\phi) : T_Q^M \rightsquigarrow T_Q^N \quad .$$

Now we can apply the lifting law for T_P to this monad morphism and obtain:

$$\text{frun}_P\big(\text{frun}_Q(\phi)\big) : T_P^{T_Q^M} \rightsquigarrow T_P^{T_Q^N} \quad .$$

This function has the correct type signature. So, we can define:

$$\text{frun}_R \stackrel{\text{def}}{=} \text{frun}_Q \fatsemi \text{frun}_P \quad .$$

It remains to prove that frun_R is a lawful lifting. We use the fact that both flift_P and flift_Q are lawful liftings; we need to show that their composition is also a lawful lifting. To verify the identity law of lifting, apply flift_R to an identity function $\text{id} : M \rightsquigarrow M$,

$$\begin{aligned}&\text{frun}_R(\text{id}^{:M\rightsquigarrow M}) = \text{frun}_P\big(\underline{\text{frun}_Q(\text{id}^{:M\rightsquigarrow M})}\big)\\ \text{identity law of frun}_Q :\quad &= \text{frun}_P\big(\text{id}^{:T_Q^M\rightsquigarrow T_Q^M}\big)\\ \text{identity law of frun}_P :\quad &= \text{id} \quad .\end{aligned}$$

To verify the composition law of lifting, apply frun_R to a composition of two monad morphisms $\phi : L \rightsquigarrow M$ and $\chi : M \rightsquigarrow N$,

$$\begin{aligned}\text{frun}_R\,(\phi \fatsemi \chi) &= \text{frun}_P\underline{(\text{frun}_Q(\phi \fatsemi \chi))}\\ \text{composition law of frun}_Q :\quad &= \underline{\text{frun}_P}(\text{frun}_Q(\phi) \fatsemi \text{frun}_Q(\chi))\\ \text{composition law of frun}_P :\quad &= \underline{\text{frun}_P(\text{frun}_Q(\phi))} \fatsemi \underline{\text{frun}_P(\text{frun}_Q(\phi))}\\ \text{definition of frun}_R :\quad &= \text{frun}_R(\phi) \fatsemi \text{frun}_R(\chi) \quad .\end{aligned}$$

Base runner law We need to show that for any monad morphism $\theta : T_P^Q \rightsquigarrow \text{Id}$ and for any monad M, there exists a monad morphism $\text{brun}_R(\theta) : T_P^{T_Q^M} \rightsquigarrow M$. To define this morphism for a given θ, we will need to use the base runners for T_P and T_Q. The base runner for T_Q has the type signature:

$$\text{brun}_Q : (Q \rightsquigarrow \text{Id}) \to (T_Q^M \rightsquigarrow M) \quad .$$

We can apply the base runner for T_P with the foreign monad T_Q^M,

$$\text{brun}_P : (P \rightsquigarrow \text{Id}) \to (T_P^{T_Q^M} \rightsquigarrow T_Q^M) \quad .$$

It is now clear that we could obtain a monad morphism $T_P^{T_Q^M} \rightsquigarrow M$ if we had some monad morphisms $\phi : P \rightsquigarrow \text{Id}$ and $\chi : Q \rightsquigarrow \text{Id}$,

$$\text{brun}_P(\phi) ⨾ \text{brun}_Q(\chi) : T_P^{T_Q^M} \rightsquigarrow M \quad .$$

However, we are only given a single monad morphism $\theta : T_P^Q \rightsquigarrow \text{Id}$. How can we compute ϕ and χ out of θ? We can use the liftings $\text{blift}_P : P \rightsquigarrow T_P^Q$ and $\text{flift}_P : Q \rightsquigarrow T_P^Q$, which are both monad morphisms, and compose them with θ:

$$(\text{blift}_P ⨾ \theta) : P \rightsquigarrow \text{Id} \quad ; \qquad (\text{flift}_P ⨾ \theta) : Q \rightsquigarrow \text{Id} \quad .$$

So we can define the monad morphism $\text{brun}_R(\theta)$ as:

$$\begin{aligned} &\text{brun}_R(\theta) : T_P^{T_Q^M} \rightsquigarrow M \quad , \\ &\text{brun}_R(\theta) \stackrel{\text{def}}{=} \text{brun}_P\,(\text{blift}_P ⨾ \theta) ⨾ \text{brun}_Q\,(\text{flift}_P ⨾ \theta) \quad . \end{aligned}$$

Since we have defined $\text{brun}_R(\theta)$ as a composition of monad morphisms, $\text{brun}_R(\theta)$ is itself a monad morphism by Statement 10.3.4.5.

To verify the nondegeneracy law of the base runner, $\text{flift}_R ⨾ \text{brun}_R(\theta) = \text{id}$, we need to use the nondegeneracy laws for the base runners of T_P and T_Q, which are:

$$\text{flift}_P ⨾ \text{brun}_P(\chi^{:P\rightsquigarrow\text{Id}}) = \text{id} \quad , \qquad \text{flift}_Q ⨾ \text{brun}_Q(\psi^{:Q\rightsquigarrow\text{Id}}) = \text{id} \quad .$$

Then we can write:

$$\begin{aligned} & \underline{\text{flift}_R} ⨾ \underline{\text{brun}_R(\theta)} \\ \text{expand definitions}: \quad & = \text{flift}_Q ⨾ \underline{\text{flift}_P ⨾ \text{brun}_P\,(\text{blift}_P ⨾ \theta)} ⨾ \text{brun}_Q\,(\text{flift}_P ⨾ \theta) \\ \text{non-degeneracy law of brun}_P: \quad & = \underline{\text{flift}_Q ⨾ \text{brun}_Q\,(\text{flift}_P ⨾ \theta)} \\ \text{non-degeneracy law of brun}_Q: \quad & = \text{id} \quad . \end{aligned}$$

Monadic naturality laws The monadic naturality law of flift_R is:

$$\text{flift}_R ⨾ \text{frun}_R(\phi) = \phi ⨾ \text{flift}_R \quad .$$

We have defined $\text{flift}_R \stackrel{\text{def}}{=} \text{flift}_Q ⨾ \text{flift}_P$, and we may assume that the monadic naturality laws hold for flift_P and flift_Q. A composition of natural transformations is again a natural transformation; this holds for any category, including the category of monads. We can also verify this law directly:

$$\begin{aligned} \text{expect to equal } \phi ⨾ \text{flift}_R: \quad & \text{flift}_R ⨾ \text{frun}_R(\phi) = \text{flift}_Q ⨾ \underline{\text{flift}_P ⨾ \text{frun}_P}(\text{frun}_Q(\phi)) \\ \text{monadic naturality of flift}_P: \quad & = \underline{\text{flift}_Q ⨾ \text{frun}_Q(\phi)} ⨾ \text{flift}_P \\ \text{monadic naturality of flift}_Q: \quad & = \phi ⨾ \text{flift}_Q ⨾ \text{flift}_P = \phi ⨾ \text{flift}_R \quad . \end{aligned}$$

The monadic naturality of brun_R is verified similarly, assuming the same law for brun_P and brun_Q:

$$\begin{aligned} \text{expect brun}_R(\theta) ⨾ \phi: \quad & \text{frun}_R(\phi) ⨾ \text{brun}_R(\theta) = \underline{\text{frun}_P(\text{frun}_Q(\phi)) ⨾ \text{brun}_P}\,(\text{blift}_P ⨾ \theta) ⨾ \text{brun}_Q\,(\text{flift}_P ⨾ \theta) \\ \text{same law for brun}_P: \quad & = \text{brun}_P\,(\text{blift}_P ⨾ \theta) ⨾ \underline{\text{frun}_Q(\phi) ⨾ \text{brun}_Q}\,(\text{flift}_P ⨾ \theta) \\ \text{same law for brun}_Q: \quad & = \text{brun}_P\,(\text{blift}_P ⨾ \theta) ⨾ \text{brun}_Q\,(\text{flift}_P ⨾ \theta) ⨾ \phi = \text{brun}_R(\theta) ⨾ \phi \quad . \end{aligned}$$

14.2.9 Stacking any number of monads. Proofs

If the monad transformer for T_P^Q is applied to another monad K, we obtain a new monad S:

$$S^A \stackrel{\text{def}}{=} T_P^{T_Q^K, A} = (P \triangleleft Q \triangleleft K)^A.$$

What is the monad transformer T_S for the monad S? Assuming that the monad transformer T_K is known, we may stack the monads one level higher:

$$T_S^{M,A} \stackrel{\text{def}}{=} T_P^{T_Q^{T_K^M}, A} = (P \triangleleft Q \triangleleft K \triangleleft M)^A \quad .$$

The result is a stack of four monads P, Q, K, and M. Note that the type parameter A is used as $T_P^{(\ldots),A}$, that is, it belongs to the *outer* transformer (T_P).

We can now define a transformer stack for any number of monads P, Q, ..., Z in a similar way:

$$T_S^{M,A} \stackrel{\text{def}}{=} T_P^{T_Q^{\cdot^{\cdot^{T_Z^M}}}, A} = (P \triangleleft Q \triangleleft \ldots \triangleleft Z \triangleleft M)^A \quad . \tag{14.9}$$

The type parameter A will always remain at the outer transformer level, while the foreign monad M will be in the innermost nested position.

It turns out that T_S is a lawful monad transformer for *any* number of stacked monads. We can prove this by induction on the number of monads. In the previous section, we have proved the transformer laws for any *three* stacked monads. Now we need to prove the same laws for a general transformer stack, such as that in Eq. (14.9). Let us temporarily denote by J the monad:

$$J \stackrel{\text{def}}{=} T_Q^{\cdot^{\cdot^{T_Z^{\text{Id}}}}} = Q \triangleleft \ldots \triangleleft Z \triangleleft \text{Id} \cong Q \triangleleft \ldots \triangleleft Z \quad ,$$

where we used the identity monad Id in the place normally taken by a foreign monad M. The monad J is a shorter transformer stack than S, so the inductive assumption tells us that the transformer laws already hold for the transformer T_J defined as:

$$T_J^M \stackrel{\text{def}}{=} T_Q^{\cdot^{\cdot^{T_Z^M}}} = (Q \triangleleft \ldots \triangleleft Z) \triangleleft M \quad .$$

Since both T_P and T_J are lawful transformers, their stacking composition $T_P^{T_J^\bullet}$ is also a lawful transformer (this was shown in the Section 14.2.8). In our notation, $T_S^{M,A} = T_P^{T_J^M, A}$, and so we have shown that T_S is a lawful transformer.

14.3 Common properties of compositional transformers

We have seen examples of monad transformers built via functor composition, either as composed-inside or as composed-outside. The simplest examples are the `OptionT` transformer:

$$L^A \stackrel{\text{def}}{=} \mathbb{1} + A, \qquad T_L^{M,A} \stackrel{\text{def}}{=} M^{L^A} = M^{\mathbb{1}+A} \quad ,$$

which puts the base monad L *inside* the monad M, and the `ReaderT` transformer:

$$L^A \stackrel{\text{def}}{=} R \to A, \qquad T_L^{M,A} \stackrel{\text{def}}{=} L^{M^A} = R \to M^A \quad ,$$

which puts the base monad L *outside* the foreign monad M.

We can prove many properties of both kinds of monad transformers via a single derivation if we temporarily drop the distinction between the base monad and the foreign monad. We simply assume

that two different monads, L and M, have a functor composition $T^\bullet \stackrel{\text{def}}{=} L^{M^\bullet}$ that also happens to be a monad. Since the assumptions on the monads L and M are the same, the resulting properties of the composed monad T will apply equally to both kinds of monad transformers.

To interpret the results, we will assume that L is the base monad for the composed-outside transformers, and that M is the base monad for the composed-inside transformers. For instance, we will be able to prove the laws of liftings $L \rightsquigarrow T$ and $M \rightsquigarrow T$ regardless of the choice of the base monad.

What properties of monad transformers will *not* be derivable in this way? Monad transformers depend on the structure on the base monad, but not on the structure of the foreign monad; the transformer's methods `pure` and `flatten` are parametric in the foreign monad. This is expressed via the monad transformer laws for the runners `frun` and `brun`, which we will need to derive separately for each of the two kinds of transformers.

14.3.1 Motivation for the `swap` function

The first task is to verify the monad laws; that is, to show that $T^\bullet \stackrel{\text{def}}{=} L^{M^\bullet}$ is a lawful monad. For that, we need to define the methods for the monad T, namely `pure` (short notation "pu_T") and `flatten` (short notation "ftn_T"), with the type signatures:

$$\text{pu}_T : A \to L^{M^A} \quad , \qquad \text{ftn}_T : L^{M^{L^{M^A}}} \to L^{M^A} \quad .$$

How can we implement these methods? *All we know* about L and M is that they are monads with their own methods pu_L, ftn_L, pu_M, and ftn_M. We can easily implement pu_T:

$$\begin{tikzcd} A \arrow[r, "\text{pu}_M"] \arrow[dr, "\text{pu}_T \stackrel{\text{def}}{=}"'] & M^A \arrow[d, "\text{pu}_L"] \\ & L^{M^A} \end{tikzcd}$$

$$\text{pu}_T \stackrel{\text{def}}{=} \text{pu}_M \,\text{⨾}\, \text{pu}_L \quad . \tag{14.10}$$

It remains to implement ftn_T. In the type $L^{M^{L^{M^A}}}$, we have two layers of the functor L and two layers of the functor M. We could use the available method ftn_L to flatten the two layers of L if we could *somehow* bring these nested layers together. However, these layers are separated by a layer of the functor M. To show this layered structure in an easier way, let us use another notation for the functor composition:

$$(L \circ M)^A \stackrel{\text{def}}{=} L^{M^A} \quad .$$

In this notation, the type signature for `flatten` is written as:

$$\text{ftn}_T : L \circ M \circ L \circ M \rightsquigarrow L \circ M \quad .$$

If we had $L \circ L \circ M \circ M$ here, we would have applied ftn_L and flattened the two layers of the functor L. Then we would have flattened the remaining two layers of the functor M. How can we proceed with $L \circ M \circ L \circ M$? The trick is to assume that we have a function called `swap` (short notation "sw"), which will interchange the order of the layers. The type signature of `swap` is:

$$\text{sw} : M \circ L \rightsquigarrow L \circ M \quad , \quad \text{or more verbosely:} \qquad \text{sw}^A_{L,M} : M^{L^A} \to L^{M^A} \quad .$$

If the `swap` operation were *somehow* defined for the two monads L and M, we could implement ftn_T by first swapping the order of the inner layers M and L as:

$$\text{sw}^{\uparrow L}_{L,M} : L \circ M \circ L \circ M \rightsquigarrow L \circ L \circ M \circ M \quad ,$$

and then applying the `flatten` methods of the monads L and M. The resulting code for the function ftn_T and the corresponding type diagram are:

$$\begin{tikzdiagram}
L^{M^{L^{M^A}}} \arrow[r, "\text{sw}^{\uparrow L}"] \arrow[drr, "\text{ftn}_T \stackrel{\text{def}}{=}"'] & L^{M^{M^A}} \arrow[r, "\text{ftn}_L"] & L^{M^{M^A}} \arrow[d, "\text{ftn}_M^{\uparrow L}"] \\
& & L^{M^A}
\end{tikzdiagram}$$

$$\text{ftn}_T \stackrel{\text{def}}{=} \text{sw}^{\uparrow L} ⨾ \text{ftn}_L ⨾ \text{ftn}_M^{\uparrow L} \quad . \tag{14.11}$$

In this way, the new monad's `flatten` method can be defined through the `swap` operation in *both* cases (the composed-inside and the composed-outside transformers). For the two kinds of transformers, the required type signatures of those functions are:

$$\text{composed-inside}: \quad \text{ftn}_T : M^{L^{M^{L^A}}} \to M^{L^A} \quad , \quad \text{sw}^A_{M,L} : L^{M^A} \to M^{L^A} \quad ,$$

$$\text{composed-outside}: \quad \text{ftn}_T : L^{M^{L^{M^A}}} \to L^{M^A} \quad , \quad \text{sw}^A_{L,M} : M^{L^A} \to L^{M^A} \quad .$$

The difference between `swap` and `sequence` There is a certain similarity between the `swap` operation introduced here and the `sequence` operation introduced in Chapter 12 for traversable functors. Indeed, the type signature of `sequence` is:

$$\text{seq} : L^{F^A} \to F^{L^A} \quad ,$$

where L is a traversable functor and F is an arbitrary applicative functor (which could be M, since all monads are also applicative functors). However, the similarity ends here. If we wish $M \circ L$ to be a monad, the `swap` operation must obey special laws (to be derived next) that are quite different from the laws of traversable functors. In particular, it is not true that $M \circ L$ is a monad whenever the monad L is a traversable functor. A counterexample is found with $L^A \stackrel{\text{def}}{=} A \times A$ and $M^A \stackrel{\text{def}}{=} \mathbb{1} + A$. Both monads L and M are traversable (since they are polynomial functors), but their functor composition $Q^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$ is not a monad (see Exercise 10.2.9.9).

Another difference between `swap` and `sequence` is that the `swap` operation needs to be parametric in the foreign monad (which may be either L or M according to the type of the monad transformer), whereas `sequence` is always parametric in the applicative functor F.

To avoid confusion, we call the function $\text{sw}_{L,M} : M^{L^\bullet} \rightsquigarrow L^{M^\bullet}$ "swap" rather than "sequence". Let us now find out what laws are required for the `swap` operation.[14]

14.3.2 Deriving the necessary laws for `swap`

The first law is that `swap` must be a natural transformation. Since `swap` has only one type parameter, there is one naturality law: for any function $f : A \to B$,

$$\begin{tikzdiagram}
M^{L^A} \arrow[r, "f^{\uparrow L \uparrow M}"] \arrow[d, "\text{sw}"'] & M^{L^B} \arrow[d, "\text{sw}"] \\
L^{M^A} \arrow[r, "f^{\uparrow M \uparrow L}"] & L^{M^B}
\end{tikzdiagram}$$

$$f^{\uparrow L \uparrow M} ⨾ \text{sw}_{L,M} = \text{sw}_{L,M} ⨾ f^{\uparrow M \uparrow L} \quad . \tag{14.12}$$

To derive further laws for `swap`, consider the requirement that the transformed monad T should satisfy the monad laws:

$$\text{pu}_T ⨾ \text{ftn}_T = \text{id} \quad , \quad \text{pu}_T^{\uparrow T} ⨾ \text{ftn}_T = \text{id} \quad , \quad \text{ftn}_T^{\uparrow T} ⨾ \text{ftn}_T = \text{ftn}_T ⨾ \text{ftn}_T \quad .$$

We will now discover the laws for `swap` that make the laws for ftn_T hold automatically, as long as ftn_T is derived from `swap` using Eq. (14.11).

We substitute Eq. (14.11) into the left identity law for ftn_T and simplify:

$$\begin{aligned}
\text{expect to equal id}: \quad & \text{pu}_T ⨾ \underline{\text{ftn}_T} \\
\text{replace ftn}_T \text{ using Eq. (14.11)}: \quad & = \text{pu}_T ⨾ \text{sw}^{\uparrow L} ⨾ \text{ftn}_L ⨾ \text{ftn}_M^{\uparrow L} \\
\text{replace pu}_T \text{ using Eq. (14.10)}: \quad & = \text{pu}_M ⨾ \underline{\text{pu}_L ⨾ \text{sw}^{\uparrow L}} ⨾ \text{ftn}_L ⨾ \text{ftn}_M^{\uparrow L} \\
\text{naturality law of pu}_L: \quad & = \text{pu}_M ⨾ \text{sw} ⨾ \underline{\text{pu}_L ⨾ \text{ftn}_L} ⨾ \text{ftn}_M^{\uparrow L} \\
\text{left identity law for } L: \quad & = \text{pu}_M ⨾ \text{sw} ⨾ \text{ftn}_M^{\uparrow L} \quad .
\end{aligned} \tag{14.13}$$

[14] The `swap` operation was studied in the paper "*Composing monads*": `http://web.cecs.pdx.edu/~mpj/pubs/RR-1004.pdf`

How could the last expression in Eq. (14.13) be equal to `identity`? We know nothing about the `pure` and `flatten` methods of the monads L and M, except that these methods obey the monad laws. We could satisfy Eq. (14.13) if we somehow reduce that expression to:

$$\text{pu}_M^{\uparrow L} \fatsemi \text{ftn}_M^{\uparrow L} = (\text{pu}_M \fatsemi \text{ftn}_M)^{\uparrow L} = \text{id} \quad .$$

This will be possible only if we are able to interchange the order of function compositions with `swap` and eliminate `swap` from the expression. So, we must require the "outer identity law" for `swap`:

$$\begin{tikzcd} L^A \arrow[r, "\text{pu}_M"] \arrow[dr, "\text{pu}_M^{\uparrow L}"'] & M^{L^A} \arrow[d, "\text{sw}"] \\ & L^{M^A} \end{tikzcd} \qquad \text{pu}_M \fatsemi \text{sw} = \text{pu}_M^{\uparrow L} \quad . \tag{14.14}$$

Intuitively, this law says that a pure layer of the monad M remains pure after interchanging the order of layers with `swap`.

With this law, we can finish the derivation in Eq. (14.13):

$$\begin{aligned} &\text{expect to equal id}: && \text{pu}_M \fatsemi \text{sw} \fatsemi \text{ftn}_M^{\uparrow L} \\ &\text{outer identity law of sw}: && = \text{pu}_M^{\uparrow L} \fatsemi \text{ftn}_M^{\uparrow L} = (\underline{\text{pu}_M \fatsemi \text{ftn}_M})^{\uparrow L} = \text{id}^{\uparrow L} = \text{id} \quad . \end{aligned}$$

So, the M-identity law for `swap` entails the left identity law for T.

In the same way, we motivate the "inner identity" law for `swap`:

$$\begin{tikzcd} M^A \arrow[r, "\text{pu}_L^{\uparrow M}"] \arrow[dr, "\text{pu}_L"'] & M^{L^A} \arrow[d, "\text{sw}"] \\ & L^{M^A} \end{tikzcd} \qquad \text{pu}_L^{\uparrow M} \fatsemi \text{sw} = \text{pu}_L \quad . \tag{14.15}$$

This law requires that a "pure" layer of the functor L should remain "pure" after swapping the order of layers.

Assuming this law, we can derive the right identity law for T:

$$\begin{aligned} &\text{expect to equal id}: && \text{pu}_T^{\uparrow T} \fatsemi \text{ftn}_T \\ &\text{by definition, } f^{\uparrow T} \stackrel{\text{def}}{=} f^{\uparrow M \uparrow L}: && = (\text{pu}_T)^{\uparrow M \uparrow L} \fatsemi \text{ftn}_T \\ &\text{definitions of } \text{pu}_T \text{ and } \text{ftn}_T: && = \text{pu}_M^{\uparrow M \uparrow L} \fatsemi \underline{\text{pu}_L^{\uparrow M \uparrow L} \fatsemi \text{sw}^{\uparrow L}} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\ &\text{inner identity law of sw, under } {}^{\uparrow L}: && = \text{pu}_M^{\uparrow M \uparrow L} \fatsemi \underline{\text{pu}_L^{\uparrow L} \fatsemi \text{ftn}_L} \fatsemi \text{ftn}_M^{\uparrow L} \\ &\text{right identity law of } L: && = \text{pu}_M^{\uparrow M \uparrow L} \fatsemi \text{ftn}_M^{\uparrow L} = \underline{(\text{pu}_M^{\uparrow M} \fatsemi \text{ftn}_M)}^{\uparrow L} = \text{id}^{\uparrow L} = \text{id} \quad . \end{aligned}$$

Deriving the monad associativity law for T,

$$\text{ftn}_T^{\uparrow T} \fatsemi \text{ftn}_T = \text{ftn}_T \fatsemi \text{ftn}_T \quad ,$$

turns out to require *two* further laws of `swap`. Let us see why.

Substituting the definition of ftn_T into the associativity law, we get:

$$\begin{aligned} &(\text{sw}^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L})^{\uparrow M \uparrow L} \fatsemi \text{sw}^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\ &\stackrel{?}{=} \text{sw}^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \fatsemi \text{sw}^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \quad . \end{aligned} \tag{14.16}$$

The only hope of proving this law is being able to interchange ftn_L as well as ftn_M with sw. In other words, the `swap` function should be in some way adapted to the `flatten` methods of both L and M.

Let us look for such interchange laws. One possibility is to have a law involving $\text{ftn}_M \fatsemi \text{sw}$, which is a function of type $M^{M^{L^A}} \to L^{M^A}$ or, in equivalent but shorter notation, $M \circ M \circ L \rightsquigarrow L \circ M$. This function first flattens the two adjacent layers of M, obtaining $M \circ L$, and then swaps the two remaining layers, moving the L layer outside. Let us think about what law could exist for this kind of transformation. It is plausible that we may obtain the same result if we first swap the layers twice, so that the L layer moves to the outside, obtaining $L \circ M \circ M$, and then flatten the two inner M layers. Writing this assumption in code, we obtain the "outer interchange" law:

$$\text{ftn}_M \,⨾\, \text{sw} = \text{sw}^{\uparrow M} \,⨾\, \text{sw} \,⨾\, \text{ftn}_M^{\uparrow L} \quad . \tag{14.17}$$

$$\begin{tikzcd}
 & M^{M^{L^A}} \arrow[r, "\text{ftn}_M"] \arrow[dl, "\text{sw}^{\uparrow M}"'] & M^{L^A} \arrow[d, "\text{sw}"] \\
M^{L^{M^A}} \arrow[r, "\text{sw}"] & L^{M^{M^A}} \arrow[r, "\text{ftn}_M^{\uparrow L}"] & L^{M^A}
\end{tikzcd}$$

The analogous "inner interchange" law involving two layers of L and a transformation $M \circ L \circ L \rightsquigarrow L \circ M$ is written as:

$$\text{ftn}_L^{\uparrow M} \,⨾\, \text{sw} = \text{sw} \,⨾\, \text{sw}^{\uparrow L} \,⨾\, \text{ftn}_L \quad . \tag{14.18}$$

$$\begin{tikzcd}
 & M^{L^{L^A}} \arrow[r, "\text{ftn}_L^{\uparrow M}"] \arrow[dl, "\text{sw}"'] & M^{L^A} \arrow[d, "\text{sw}"] \\
L^{M^{L^A}} \arrow[r, "\text{sw}^{\uparrow L}"] & L^{L^{M^A}} \arrow[r, "\text{ftn}_L"] & L^{M^A}
\end{tikzcd}$$

At this point, we have simply written down two plausible laws, hoping that they will help us derive the associativity law for T. We will now verify that this is indeed so.

Both sides of the law in Eq. (14.16) involve compositions of several `flattens` and `swaps`. The heuristic idea of the proof is to use various laws to move all `flattens` to right of the composition, while moving all `swaps` to the left. In this way we will rewrite both sides of Eq. (14.16) to a similar form, hoping to prove that they are equal.

We begin with the right-hand side of Eq. (14.16) since it is simpler than the left-hand side, and look for ways of using the interchange laws. At every step of the calculation, there happens to be only one place where some law can be applied:

$$\begin{aligned}
& \text{sw}^{\uparrow L} \,⨾\, \text{ftn}_L \,⨾\, \underline{\text{ftn}_M^{\uparrow L} \,⨾\, \text{sw}^{\uparrow L}} \,⨾\, \text{ftn}_L \,⨾\, \text{ftn}_M^{\uparrow L} \\
\text{composition for } L: \quad & = \text{sw}^{\uparrow L} \,⨾\, \text{ftn}_L \,⨾\, \underline{(\text{ftn}_M \,⨾\, \text{sw})^{\uparrow L}} \,⨾\, \text{ftn}_L \,⨾\, \text{ftn}_M^{\uparrow L} \\
\text{outer interchange law of sw}: \quad & = \text{sw}^{\uparrow L} \,⨾\, \text{ftn}_L \,⨾\, \underline{(\text{sw}^{\uparrow M} \,⨾\, \text{sw} \,⨾\, \text{ftn}_M^{\uparrow L})^{\uparrow L}} \,⨾\, \text{ftn}_L \,⨾\, \text{ftn}_M^{\uparrow L} \\
\text{composition under } {}^{\uparrow L}: \quad & = \text{sw}^{\uparrow L} \,⨾\, \text{ftn}_L \,⨾\, \underline{\text{sw}^{\uparrow M \uparrow L} \,⨾\, \text{sw}^{\uparrow L}} \,⨾\, \underline{\text{ftn}_M^{\uparrow L \uparrow L} \,⨾\, \text{ftn}_L} \,⨾\, \text{ftn}_M^{\uparrow L} \\
\text{naturality law of ftn}_L: \quad & = \text{sw}^{\uparrow L} \,⨾\, \underline{\text{ftn}_L \,⨾\, (\text{sw}^{\uparrow M} \,⨾\, \text{sw})^{\uparrow L}} \,⨾\, \text{ftn}_L \,⨾\, \text{ftn}_M^{\uparrow L} \,⨾\, \text{ftn}_M^{\uparrow L} \\
\text{naturality law of ftn}_L: \quad & = \text{sw}^{\uparrow L} \,⨾\, \underline{(\text{sw}^{\uparrow M} \,⨾\, \text{sw})^{\uparrow L \uparrow L} \,⨾\, \text{ftn}_L} \,⨾\, \text{ftn}_L \,⨾\, \text{ftn}_M^{\uparrow L} \,⨾\, \text{ftn}_M^{\uparrow L} \quad .
\end{aligned}$$

Now all `swaps` are on the left and all `flattens` on the right of the expression.

Transform the right-hand side of Eq. (14.16) in the same way as:

$$\begin{aligned}
& (\text{sw}^{\uparrow L} \,⨾\, \text{ftn}_L \,⨾\, \underline{\text{ftn}_M^{\uparrow L}})^{\uparrow M \uparrow L} \,⨾\, \underline{\text{sw}^{\uparrow L}} \,⨾\, \text{ftn}_L \,⨾\, \text{ftn}_M^{\uparrow L} \\
\text{functor composition}: \quad & = (\text{sw}^{\uparrow L} \,⨾\, \text{ftn}_L)^{\uparrow M \uparrow L} \,⨾\, \underline{(\text{ftn}_M^{\uparrow L \uparrow M} \,⨾\, \text{sw})^{\uparrow L}} \,⨾\, \text{ftn}_L \,⨾\, \text{ftn}_M^{\uparrow L} \\
\text{naturality law of sw}: \quad & = (\text{sw}^{\uparrow L} \,⨾\, \text{ftn}_L)^{\uparrow M \uparrow L} \,⨾\, (\text{sw} \,⨾\, \underline{\text{ftn}_M^{\uparrow M \uparrow L}})^{\uparrow L} \,⨾\, \underline{\text{ftn}_L} \,⨾\, \text{ftn}_M^{\uparrow L} \\
\text{naturality law of ftn}_L: \quad & = \text{sw}^{\uparrow L \uparrow M \uparrow L} \,⨾\, \text{ftn}_L^{\uparrow M \uparrow L} \,⨾\, \text{sw}^{\uparrow L} \,⨾\, \text{ftn}_L \,⨾\, \underline{\text{ftn}_M^{\uparrow M \uparrow L} \,⨾\, \text{ftn}_M^{\uparrow L}} \\
\text{associativity of ftn}_M: \quad & = \text{sw}^{\uparrow L \uparrow M \uparrow L} \,⨾\, \underline{(\text{ftn}_L^{\uparrow M} \,⨾\, \text{sw})^{\uparrow L}} \,⨾\, \text{ftn}_L \,⨾\, \text{ftn}_M^{\uparrow L} \,⨾\, \text{ftn}_M^{\uparrow L} \\
\text{inner interchange law of sw}: \quad & = \text{sw}^{\uparrow L \uparrow M \uparrow L} \,⨾\, (\text{sw} \,⨾\, \text{sw}^{\uparrow L} \,⨾\, \underline{\text{ftn}_L})^{\uparrow L} \,⨾\, \underline{\text{ftn}_L} \,⨾\, \text{ftn}_M^{\uparrow L} \,⨾\, \text{ftn}_M^{\uparrow L} \\
\text{associativity law of ftn}_L: \quad & = (\text{sw}^{\uparrow L \uparrow M} \,⨾\, \text{sw} \,⨾\, \text{sw}^{\uparrow L})^{\uparrow L} \,⨾\, \underline{\text{ftn}_L \,⨾\, \text{ftn}_L} \,⨾\, \text{ftn}_M^{\uparrow L} \,⨾\, \text{ftn}_M^{\uparrow L} \quad .
\end{aligned}$$

We have again managed to move all `swaps` to the left and all `flattens` to the right of the expression.

Comparing now the two sides of the associativity law, we see that all the `flattens` occur in the same combination: $\text{ftn}_L \,⨾\, \text{ftn}_L \,⨾\, \text{ftn}_M^{\uparrow L} \,⨾\, \text{ftn}_M^{\uparrow L}$. It remains to show that:

$$\text{sw}^{\uparrow L} \,⨾\, (\text{sw}^{\uparrow M} \,⨾\, \text{sw})^{\uparrow L \uparrow L} \stackrel{?}{=} (\text{sw}^{\uparrow L \uparrow M} \,⨾\, \text{sw} \,⨾\, \text{sw}^{\uparrow L})^{\uparrow L} \quad ,$$

or equivalently:

$$(\text{sw} \,⨾\, \text{sw}^{\uparrow M \uparrow L} \,⨾\, \text{sw}^{\uparrow L})^{\uparrow L} \stackrel{?}{=} (\text{sw}^{\uparrow L \uparrow M} \,⨾\, \text{sw} \,⨾\, \text{sw}^{\uparrow L})^{\uparrow L} \quad .$$

The two sides are equal due to the naturality law of `swap`:

$$\text{sw} \,⨾\, \text{sw}^{\uparrow M \uparrow L} = \text{sw}^{\uparrow L \uparrow M} \,⨾\, \text{sw}.$$

We have proved the following statement:

Statement 14.3.2.1 For two monads L and M, assume that there exists a function `swap`:

$$\text{sw}_{L,M} : M^{L^A} \to L^{M^A} \quad ,$$

which is a natural transformation satisfying four additional laws:

$$\begin{aligned}
\text{outer identity}: &\quad \text{pu}_L^{\uparrow M} \fatsemi \text{sw}_{L,M} = \text{pu}_L \quad , \\
\text{inner identity}: &\quad \text{pu}_M \fatsemi \text{sw}_{L,M} = \text{pu}_M^{\uparrow L} \quad , \\
\text{outer interchange}: &\quad \text{ftn}_L^{\uparrow M} \fatsemi \text{sw}_{L,M} = \text{sw}_{L,M} \fatsemi \text{sw}_{L,M}^{\uparrow L} \fatsemi \text{ftn}_L \quad , \\
\text{inner interchange}: &\quad \text{ftn}_M \fatsemi \text{sw}_{L,M} = \text{sw}_{L,M}^{\uparrow M} \fatsemi \text{sw}_{L,M} \fatsemi \text{ftn}_M^{\uparrow L} \quad .
\end{aligned}$$

Then the functor composition of L and M, denoted by T:

$$T^A \stackrel{\text{def}}{=} L^{M^A} = (L \circ M)^A$$

is a monad with the methods `pure` and `flatten` defined by:

$$\text{pu}_T \stackrel{\text{def}}{=} \text{pu}_M \fatsemi \text{pu}_L \quad , \tag{14.19}$$

$$\text{ftn}_T \stackrel{\text{def}}{=} \text{sw}_{L,M}^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \quad . \tag{14.20}$$

Intuition behind `swap`'s interchange laws Those laws guarantee that any functor composition built up from L and M, e.g.:

$$M \circ M \circ L \circ M \circ L \circ L \circ M \circ M \circ L \quad ,$$

can be simplified to a value of type $T^A = L^{M^A}$ by flattening the layers using ftn_L, ftn_M, or ftn_T, or by interchanging the layers with `swap`. We may apply `flatten` or `swap` in any order, and we will always get the same final value of type T^A.

In other words, the monadic effects of the monads L and M can be arbitrarily interleaved, swapped, and flattened in any order, with no change to the final results. The programmer is free to refactor a monadic program, say, by first computing some L-effects in a separate functor block of L-`flatMap`s and only then combining the result with the rest of the computation in the monad T. Regardless of the refactoring, the monad T computes all the effects correctly. This is what programmers would expect of the monad T, if it is to be regarded as a useful monad transformer.

We will now derive the properties of the monad T that follow from the interchange laws. We will find that it is easier to formulate these laws in terms of `swap` than in terms of ftn_T. In practice, all known examples of compositional monad transformers (the `Writer` monad, the pass/fail monads, and the "rigid" monads) are defined via a `swap` function.

14.3.3 Deriving `swap` from `flatten`

We have shown that the `flatten` method of the monad $T^\bullet = L^{M^\bullet}$ can be defined via the `swap` method. However, we have seen examples of some composable monads (such as `Reader` and `Option`) where we already know the definitions of the `flatten` method for the composed monad T. Does a suitable `swap` function exist for these examples? In other words, if a `flatten` function for the monad $T = L \circ M$ is already known, can we find out whether a `swap` function exists such that the *given* `flatten` function is expressed via Eq. (14.11)?

To answer this question, let us look at the type signature of `flatten` for T:

$$\text{ftn}_T : L \circ M \circ L \circ M \rightsquigarrow L \circ M \quad .$$

This type signature is different from $\text{sw} : M \circ L \rightsquigarrow L \circ M$ in that the argument of ftn_T has extra layers of the functors L and M outside the $M \circ L$ composition. We can use the `pure` methods of M and L to add such extra layers to a value of type $M \circ L$, without modifying any monadic effects present in $M \circ L$. This will allow us to apply ftn_T and to obtain a value of type $L \circ M$. The resulting code for the function ftn_T and the corresponding type diagram are:

$$\text{sw} = \text{pu}_M^{\uparrow L\uparrow M} \fatsemi \text{pu}_L \fatsemi \text{ftn}_T \quad . \tag{14.21}$$

$$\begin{tikzcd} M^{L^A} \arrow[r, "\text{pu}_M^{\uparrow L\uparrow M}"] \arrow[drr, "\text{sw}\overset{\text{def}}{=}"'] & M^{L^{M^A}} \arrow[r, "\text{pu}_L"] & L^{M^{L^{M^A}}} \arrow[d, "\text{ftn}_T"] \\ & & L^{M^A} \end{tikzcd}$$

We have related ftn_T and $\text{sw}_{L,M}$ to each other. Are these functions always equivalent?

Statement 14.3.3.1 The equivalence of `flatten` and `swap` holds under the following conditions:

(a) Given an implementation of `swap` that obeys the naturality law, we define ftn_T via Eq. (14.11) and then substitute that ftn_T into Eq. (14.21) to define a new `swap` function. The new `swap` function will be equal to the `swap` function given initially.

(b) Given an implementation of ftn_T satisfying the "compatibility laws" (14.23)–(14.24), we define `swap` via Eq. (14.21) and then substitute that `swap` into Eq. (14.11) to define a new `flatten` function. The new `flatten` function will be equal to the ftn_T function given initially.

Proof **(a)** Substitute ftn_T from Eq. (14.11) into Eq. (14.21):

$$\begin{aligned} \text{expect to equal sw}: \quad & \text{pu}_M^{\uparrow L\uparrow M} \fatsemi \text{pu}_L \fatsemi \underline{\text{ftn}_T} = \text{pu}_M^{\uparrow L\uparrow M} \fatsemi \underline{\text{pu}_L \fatsemi \text{sw}^{\uparrow L}} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{naturality law of } \text{pu}_L: \quad & = \text{pu}_M^{\uparrow L\uparrow M} \fatsemi \text{sw} \fatsemi \underline{\text{pu}_L \fatsemi \text{ftn}_L} \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{left identity law of } L: \quad & = \underline{\text{pu}_M^{\uparrow L\uparrow M} \fatsemi \text{sw}} \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{naturality law of sw}: \quad & = \text{sw} \fatsemi \underline{\text{pu}_M^{\uparrow M\uparrow L} \fatsemi \text{ftn}_M^{\uparrow L}} = \text{sw} \fatsemi \underline{(\text{pu}_M^{\uparrow M} \fatsemi \text{ftn}_M)^{\uparrow L}} = \text{sw} \quad . \end{aligned}$$

We recovered the initial `swap` function.

(b) Substitute `swap` from Eq. (14.21) into Eq. (14.11):

$$\begin{aligned} & \underline{\text{sw}}^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} = \underline{(\text{pu}_M^{\uparrow L\uparrow M} \fatsemi \text{pu}_L \fatsemi \text{ftn}_T)^{\uparrow L}} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{functor composition}: \quad & = \text{pu}_M^{\uparrow L\uparrow M\uparrow L} \fatsemi \text{pu}_L^{\uparrow L} \fatsemi \text{ftn}_T^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \quad . \end{aligned} \tag{14.22}$$

At this point, we are stuck: we cannot transform the last expression into ftn_T. Without assuming additional laws, it *does not follow* that the right-hand side of Eq. (14.22) is equal to ftn_T. Let us now derive those additional laws.

The only sub-expression in Eq. (14.22) that we could possibly transform is the composition $\text{ftn}_T^{\uparrow L} \fatsemi \text{ftn}_L$. So, we need to assume a law involving the expression:

$$(\text{ftn}_T^{\uparrow L} \fatsemi \text{ftn}_L) : L \circ L \circ M \circ L \circ M \rightsquigarrow L \circ M \quad .$$

This function flattens the two layers of $(L \circ M)$ and then flattens the remaining two layers of L. Another function with the same type signature could first flatten the two *outside* layers of L and then flatten the two remaining layers of $(L \circ M)$:

$$(\text{ftn}_L \fatsemi \text{ftn}_T) : L \circ L \circ M \circ L \circ M \rightsquigarrow L \circ M \quad .$$

We see that a possibly useful additional law for ftn_T is:

$$\begin{tikzcd} L^{M^{L^{M^A}}} \arrow[r, "\text{ftn}_L"] \arrow[d, "\text{ftn}_T^{\uparrow L}"'] & L^{M^{L^{M^A}}} \arrow[d, "\text{ftn}_T"'] \\ L^{M^A} \arrow[r, "\text{ftn}_L"'] & L^{M^A} \end{tikzcd}$$

$$\text{ftn}_L \fatsemi \text{ftn}_T = \text{ftn}_T^{\uparrow L} \fatsemi \text{ftn}_L \quad . \tag{14.23}$$

This law expresses a certain kind of "compatibility" between the monads L and T.

With this law, the right-hand side of Eq. (14.22) becomes:

$$\begin{aligned} & \text{pu}_M^{\uparrow L\uparrow M\uparrow L} \fatsemi \underline{\text{pu}_L^{\uparrow L} \fatsemi \text{ftn}_L} \fatsemi \text{ftn}_T \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{right identity law of } L: \quad & = \text{pu}_M^{\uparrow L\uparrow M\uparrow L} \fatsemi \text{ftn}_T \fatsemi \text{ftn}_M^{\uparrow L} \quad . \end{aligned}$$

Again, we cannot proceed unless we assume a new law involving the expression:

$$(\text{ftn}_T \fatsemi \text{ftn}_M^{\uparrow L}) : L \circ M \circ L \circ M \circ M \rightsquigarrow L \circ M \quad .$$

This function first flattens the two layers of $(L \circ M)$ and then flattens the remaining two layers of M. An alternative order of flattenings is to first flatten the *innermost* two layers of M:

$$(\text{ftn}_M^{\uparrow L \uparrow M \uparrow L} \fatsemi \text{ftn}_T) : L \circ M \circ L \circ M \circ M \rightsquigarrow L \circ M \quad .$$

So, the second law to be assumed is:

$$\begin{tikzcd} L^{M^{L^{M^{M^A}}}} \arrow[r, "\text{ftn}_T"] \arrow[d, "\text{ftn}_M^{\uparrow L \uparrow M \uparrow L}"'] & L^{M^{M^A}} \arrow[d, "\text{ftn}_M^{\uparrow L}"'] \\ L^{M^{L^{M^A}}} \arrow[r, "\text{ftn}_T"'] & L^{M^A} \end{tikzcd}$$

$$\text{ftn}_T \fatsemi \text{ftn}_M^{\uparrow L} = \text{ftn}_M^{\uparrow L \uparrow M \uparrow L} \fatsemi \text{ftn}_T \quad . \tag{14.24}$$

This law expresses a certain kind of "compatibility" between the monads M and T.

Assuming this law, we can finally complete the derivation:

$$\begin{aligned} \text{expect to equal } \text{ftn}_T : \quad & \text{pu}_M^{\uparrow L \uparrow M \uparrow L} \fatsemi \text{ftn}_T \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{substitute the second conjecture} : \quad & = \underline{\text{pu}_M^{\uparrow L \uparrow M \uparrow L} \fatsemi \text{ftn}_M^{\uparrow L \uparrow M \uparrow L}} \fatsemi \text{ftn}_T \\ \text{functor composition} : \quad & = \underline{(\text{pu}_M \fatsemi \text{ftn}_M)^{\uparrow L \uparrow M \uparrow L}} \fatsemi \text{ftn}_T \\ \text{left identity law of } M : \quad & = \text{ftn}_T \quad . \end{aligned}$$

We recovered the initial ftn_T by assuming two additional laws. □

It turns out that those laws will always hold when ftn_T is defined via `swap` (see Exercise 14.9.5.10).

It may be hard to verify directly the monad laws for $L \circ M$ because of deeply nested type constructors, e.g., $L \circ M \circ L \circ M \circ L \circ M$. If the monad $L \circ M$ has a `swap` method, it is simpler first to verify the laws of `swap` and then to obtain the monad laws of $L \circ M$ via Statement 14.3.2.1.

14.3.4 Monad transformer identity law. Proofs

Statement 14.3.4.1 Assume that pu_T and ftn_T are defined by Eqs. (14.19)–(14.20), and that the two identity laws of `swap` hold (see Statement 14.3.2.1):

$$\begin{aligned} \text{outer identity law} : \quad & \text{pu}_L^{\uparrow M} \fatsemi \text{sw}_{L,M} = \text{pu}_L \quad , \\ \text{inner identity law} : \quad & \text{pu}_M \fatsemi \text{sw}_{L,M} = \text{pu}_M^{\uparrow L} \quad . \end{aligned}$$

Then the identity law holds, $T_L^M \cong L$ for $M = \text{Id}$, via monad isomorphisms.

Proof First consider what happens to $\text{sw}_{L,M}$ when L or M are set to the identity monad (Id). For composed-outside transformers ($T_L^M = L \circ M$) the foreign monad is M. Setting $M = \text{Id}$ in the inner identity law, we obtain:

$$\text{pu}_{\text{Id}} \fatsemi \text{sw}_{L,\text{Id}} = \text{pu}_{\text{Id}}^{\uparrow L} \quad .$$

Since $\text{pu}_{\text{Id}} = \text{id}$, it follows that $\text{sw}_{L,\text{Id}} = \text{id}$. In a similar way, for composed-inside transformers ($T_L^M = M \circ L$) we need to switch the roles of M and L in the same computation and substitute $L = \text{id}$ into the outer identity law:

$$\text{pu}_{\text{Id}}^{\uparrow M} \fatsemi \text{sw}_{\text{Id},M} = \text{pu}_{\text{Id}} \quad .$$

We obtain $\text{sw}_{\text{Id},M} = \text{id}$.

To demonstrate a monadic isomorphism between the monads T_L^{Id} and L, we will consider separately the cases of composed-inside and composed-outside transformers.

For composed-inside transformers $T_L^M = M \circ L$, we set $M = \text{Id}$ and find that the monad $T_L^{\text{Id}} = \text{Id} \circ L = L$ is the same type constructor as L. So, the isomorphism maps between T_L^{Id} and L are simply the identity maps in both directions, $\text{id} : T_L^{\text{Id},A} \to L^A$ and $\text{id} : L^A \to T_L^{\text{Id},A}$.

For composed-outside transformers $T_L^M = L \circ M$, the monad $T_L^{\text{Id}} = L \circ \text{Id} = L$ is again the same type constructor as L. The isomorphisms between T_L^{Id} and L are again the identity maps in both directions, $\text{id} : T_L^{\text{Id},A} \to L^A$ and $\text{id} : L^A \to T_L^{\text{Id},A}$.

We have found the isomorphism maps between T_L^{Id} and L. However, we still need to verify that the monad structure of T_L^{Id} is the same as that of L; otherwise the isomorphism would not be a *monad* morphism (i.e., an isomorphism that preserves the structure of the monads). To verify this, it is sufficient to show that the methods pu_T and ftn_T defined by Eqs. (14.19)–(14.20) for the monad T_L^{Id} are *the same functions* as the given methods pu_L and ftn_L of the monad L. If the monad's methods are the same functions, i.e. $\text{pu}_L = \text{pu}_T$ and $\text{ftn}_L = \text{ftn}_T$, then the identity map $\text{id} : T^A \to L^A$ will satisfy the laws of the monad morphisms:

$$\text{pu}_T \fatsemi \text{id} = \text{pu}_L \quad , \qquad \text{ftn}_T \fatsemi \text{id} = \text{id}^{\uparrow T} \fatsemi \text{id} \fatsemi \text{ftn}_L \quad .$$

In the same way, the laws of the monad morphisms will hold for "id" in the direction $L \rightsquigarrow T$.

For composed-inside transformers: We need to show that $\text{pu}_M = \text{pu}_T$ and $\text{ftn}_M = \text{ftn}_T$. Designate L as the foreign monad and M as the base monad in Eq. (14.19), as appropriate for the composed-inside transformer $T_L^M = M \circ L$. Setting the foreign monad to identity, $L = \text{Id}$, in Eq. (14.19) gives:

$$\text{pu}_T = \text{pu}_M \fatsemi \text{pu}_{\text{Id}} = \text{pu}_M .$$

To show that $\text{ftn}_M = \text{ftn}_T$, we use Eq. (14.20) with $L = \text{Id}$:

$$\begin{aligned} \text{use Eq. (14.20)}: \quad & \text{ftn}_T = \text{sw}_{\text{Id},M}^{\uparrow \text{Id}} \fatsemi \text{ftn}_{\text{Id}} \fatsemi \text{ftn}_M^{\uparrow \text{Id}} \\ \text{use } \text{ftn}_{\text{Id}} = \text{id} \text{ and } \text{sw}_{\text{Id},M} = \text{id}: \quad & = \text{ftn}_M \quad . \end{aligned}$$

For composed-outside transformers: We need to show that $\text{pu}_L = \text{pu}_T$ and $\text{ftn}_L = \text{ftn}_T$. Designate M as the foreign monad and L as the base monad in Eq. (14.19), as appropriate for the composed-outside transformer $T_L^M = L \circ M$. Setting the foreign monad to identity, $M = \text{Id}$, in Eq. (14.19) gives:

$$\text{pu}_T = \text{pu}_{\text{Id}} \fatsemi \text{pu}_L = \text{pu}_L .$$

To show that $\text{ftn}_L = \text{ftn}_T$, use Eq. (14.20) with $M = \text{Id}$:

$$\begin{aligned} \text{use Eq. (14.20)}: \quad & \text{ftn}_T = \text{sw}_{L,\text{Id}}^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_{\text{Id}}^{\uparrow L} \\ \text{use } \text{ftn}_{\text{Id}} = \text{id} \text{ and } \text{sw}_{L,\text{Id}} = \text{id}: \quad & = \text{ftn}_L \quad . \end{aligned}$$

14.3.5 Monad transformer lifting laws. Proofs

Statement 14.3.5.1 The monad transformer lifting laws hold if the monad methods of T are defined using Eqs. (14.19)–(14.20), and if the `swap` function obeys the laws defined in Statement 14.3.2.1.

Proof To be specific, let us assume that L is the base monad of the transformer. For the other choice of the base monad, we only need to interchange the definitions of `flift` and `blift`.

The lift functions of a compositional monad transformer are:

$$\text{flift} = \text{pu}_L : M^A \to L^{M^A} \quad , \qquad \text{blift} = \text{pu}_M^{\uparrow L} : L^A \to L^{M^A} \quad .$$

Their laws of monad morphisms (the identity and the composition laws) are:

$$\begin{aligned} \text{pu}_M \fatsemi \text{flift} &= \text{pu}_T \quad , \qquad \text{pu}_L \fatsemi \text{blift} = \text{pu}_T \quad , \\ \text{ftn}_M \fatsemi \text{flift} &= \text{flift}^{\uparrow M} \fatsemi \text{flift} \fatsemi \text{ftn}_T \quad , \qquad \text{ftn}_L \fatsemi \text{blift} = \text{blift}^{\uparrow L} \fatsemi \text{blift} \fatsemi \text{ftn}_T \quad . \end{aligned}$$

The identity laws are verified quickly:

$$\begin{aligned} \text{expect to equal } \text{pu}_T: \quad & \text{pu}_M \fatsemi \text{flift} = \text{pu}_M \fatsemi \text{pu}_L = \text{pu}_T \quad , \\ \text{expect to equal } \text{pu}_T: \quad & \text{pu}_L \fatsemi \text{blift} = \text{pu}_L \fatsemi \text{pu}_M^{\uparrow L} = \text{pu}_M \fatsemi \text{pu}_L = \text{pu}_T \quad . \end{aligned}$$

To verify the composition laws, we need to start from their right-hand sides because the left-hand sides cannot be simplified. We then substitute the definition of ftn_T in terms of `swap`.

The composition law for `flift`:

$$\begin{aligned}
&\text{expect to equal } \text{ftn}_M \fatsemi \text{pu}_L : && \text{flift}^{\uparrow M} \fatsemi \text{flift} \fatsemi \text{ftn}_T \\
&\text{definitions of flift and ftn}_T : && = \text{pu}_L^{\uparrow M} \fatsemi \underline{\text{pu}_L \fatsemi \text{sw}^{\uparrow L}} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\
&\text{naturality law of pu}_L : && = \text{pu}_L^{\uparrow M} \fatsemi \text{sw} \fatsemi \underline{\text{pu}_L \fatsemi \text{ftn}_L} \fatsemi \text{ftn}_M^{\uparrow L} \\
&\text{left identity law of } L : && = \underline{\text{pu}_L^{\uparrow M} \fatsemi \text{sw}} \fatsemi \text{ftn}_M^{\uparrow L} \\
&\text{inner identity law of sw} : && = \text{pu}_L \fatsemi \text{ftn}_M^{\uparrow L} = \text{ftn}_M \fatsemi \text{pu}_L \quad .
\end{aligned}$$

The composition law for `blift`:

$$\begin{aligned}
&\text{expect to equal } \text{ftn}_L \fatsemi \text{pu}_M^{\uparrow L} : && \text{blift}^{\uparrow L} \fatsemi \text{blift} \fatsemi \text{ftn}_T \\
&\text{definitions of blift and ftn}_T : && = \text{pu}_M^{\uparrow L \uparrow L} \fatsemi \underline{\text{pu}_M^{\uparrow L} \fatsemi \text{sw}^{\uparrow L}} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\
&\text{functor composition in } L : && = \text{pu}_M^{\uparrow L \uparrow L} \fatsemi \underline{(\text{pu}_M \fatsemi \text{sw})^{\uparrow L}} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\
&\text{outer identity law of sw} : && = \underline{(\text{pu}_M^{\uparrow L \uparrow L} \fatsemi \text{pu}_M^{\uparrow L \uparrow L})} \fatsemi \underline{\text{ftn}_L} \fatsemi \text{ftn}_M^{\uparrow L} \\
&\text{naturality law of ftn}_L : && = \text{ftn}_L \fatsemi (\text{pu}_M^{\uparrow L} \fatsemi \underline{\text{pu}_M^{\uparrow L}}) \fatsemi \underline{\text{ftn}_M^{\uparrow L}} \\
&\text{right identity law of } M : && = \text{ftn}_L \fatsemi \text{pu}_M^{\uparrow L} \quad .
\end{aligned}$$

So, the lifting laws for T follow from the laws of `swap`.

14.3.6 Monad transformer runner laws. Proofs

Statement 14.3.6.1 The runners (`frun` and `brun`) satisfy the runner laws and the base runner laws, as long as the `swap` function is monadically natural as given by Eqs. (14.25) or (14.26) below. Compositional monad transformers have general base runners, $\text{brun}(\theta)$, that satisfy the laws for any runner $\theta : L \rightsquigarrow \text{Id}$. The functions `frun` and `brun` also satisfy the monadic naturality laws.

Proof The laws of runners are not symmetric with respect to the base monad and the foreign monad: the runners are parametric in the foreign monad but not in the base monad. In each case, the `swap` function must be monadically natural with respect to the *foreign* monad. So, the laws need to be written differently, depending on the choice of the base monad. Let us consider separately the cases where either L or M is the base monad.

(a) If the base monad is L, the runners are defined by:

$$\begin{aligned}
&\text{frun}(\phi^{:M \rightsquigarrow N}) : L \circ M \rightsquigarrow L \circ N \quad , \qquad \text{frun}(\phi) = \phi^{\uparrow L} \quad ; \\
&\text{brun}(\theta^{:L \rightsquigarrow \text{Id}}) : L \circ M \rightsquigarrow M \quad , \qquad \text{brun}(\theta) = \theta \quad .
\end{aligned}$$

The laws of runners require that $\text{frun}(\phi)$ and $\text{brun}(\theta)$ must be monad morphisms, i.e., the identity and composition laws must hold:

$$\begin{aligned}
&\text{pu}_{L \circ M} \fatsemi \text{frun}(\phi) = \text{pu}_{L \circ N} \quad , \qquad \text{ftn}_{L \circ M} \fatsemi \text{frun}(\phi) = (\text{frun}(\phi))^{\uparrow M \uparrow L} \fatsemi \text{frun}(\phi) \fatsemi \text{ftn}_{L \circ N} \quad , \\
&\text{pu}_{L \circ M} \fatsemi \text{brun}(\theta) = \text{pu}_M \quad , \qquad \text{ftn}_{L \circ M} \fatsemi \text{brun}(\theta) = (\text{brun}(\theta))^{\uparrow M \uparrow L} \fatsemi \text{brun}(\theta) \fatsemi \text{ftn}_M \quad .
\end{aligned}$$

To derive these laws, we may use the identity and composition laws of monad morphisms for ϕ and θ. We also use Eqs. (14.19)–(14.20) as definitions of the monad T. Additionally, we assume the **monadic naturality** of `swap` with respect to ϕ and θ:

$$\text{sw}_{L,M} \fatsemi \phi^{\uparrow L} = \phi \fatsemi \text{sw}_{L,N} \quad , \qquad \text{sw}_{L,M} \fatsemi \theta = \theta^{\uparrow M} \quad . \tag{14.25}$$

$$\begin{tikzcd} M^{L^A} \arrow[r, "\mathrm{sw}_{L,M}"] \arrow[d, "\phi"'] & L^{M^A} \arrow[d, "\phi^{\uparrow L}"] \\ N^{L^A} \arrow[r, "\mathrm{sw}_{L,N}"] & L^{N^A} \end{tikzcd} \qquad \begin{tikzcd} M^{L^A} \arrow[r, "\mathrm{sw}_{L,M}"] \arrow[dr, "\theta^{\uparrow M}"'] & L^{M^A} \arrow[d, "\theta"] \\ & M^A \end{tikzcd}$$

The first law to be shown is the identity law of frun (ϕ):

$$\begin{aligned} \text{expect to equal } \mathrm{pu}_{L\circ N} : &\quad \mathrm{pu}_{L\circ M} \fatsemi \mathrm{frun}\,(\phi) \\ \text{definitions of frun and } \mathrm{pu}_{L\circ M} : &\quad = \mathrm{pu}_M \fatsemi \underline{\mathrm{pu}_L \fatsemi \phi^{\uparrow L}} \\ \text{naturality law of } \mathrm{pu}_L : &\quad = \underline{\mathrm{pu}_M \fatsemi \phi} \fatsemi \mathrm{pu}_L \\ \text{identity law of } \phi : &\quad = \mathrm{pu}_N \fatsemi \mathrm{pu}_L \\ \text{definition of } \mathrm{pu}_{L\circ N} : &\quad = \mathrm{pu}_{L\circ N} \quad . \end{aligned}$$

The next law to be verified is the composition law of frun (ϕ):

$$\begin{aligned} \text{expect to equal } \mathrm{ftn}_T \fatsemi \phi^{\uparrow L} : &\quad (\mathrm{frun}\,(\phi))^{\uparrow M \uparrow L} \fatsemi \mathrm{frun}\,(\phi) \fatsemi \mathrm{ftn}_{L\circ N} \\ \text{definitions of frun and } \mathrm{ftn}_{L\circ N} : &\quad = \phi^{\uparrow L \uparrow M \uparrow L} \fatsemi \underline{\phi^{\uparrow L} \fatsemi \mathrm{sw}^{\uparrow L}_{L,N}} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}^{\uparrow L}_N \\ \text{monadic naturality law of } \mathrm{sw}_{L,M} : &\quad = \underline{\phi^{\uparrow L \uparrow M \uparrow L} \fatsemi \mathrm{sw}^{\uparrow L}_{L,M}} \fatsemi \phi^{\uparrow L \uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}^{\uparrow L}_N \\ \text{naturality law of } \mathrm{sw}_{L,M} : &\quad = \mathrm{sw}^{\uparrow L}_{L,M} \fatsemi \underline{\phi^{\uparrow M \uparrow L \uparrow L} \fatsemi \phi^{\uparrow L \uparrow L} \fatsemi \mathrm{ftn}_L} \fatsemi \mathrm{ftn}^{\uparrow L}_N \\ \text{naturality law of } \mathrm{ftn}_L : &\quad = \mathrm{sw}^{\uparrow L}_{L,M} \fatsemi \mathrm{ftn}_L \fatsemi \underline{(\phi^{\uparrow M} \fatsemi \phi \fatsemi \mathrm{ftn}_N)^{\uparrow L}} \\ \text{composition law of } \phi : &\quad = \underline{\mathrm{sw}^{\uparrow L}_{L,M} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}^{\uparrow L}_M} \fatsemi \phi^{\uparrow L} \\ \text{definition of } \mathrm{ftn}_T : &\quad = \underline{\mathrm{ftn}_T} \fatsemi \phi^{\uparrow L} \quad . \end{aligned}$$

The next law is the identity law for `brun`:

$$\begin{aligned} \text{expect to equal } \mathrm{pu}_M : &\quad \mathrm{pu}_{L\circ M} \fatsemi \mathrm{brun}\,\theta \\ \text{definitions of brun and } \mathrm{pu}_{L\circ M} : &\quad = \mathrm{pu}_M \fatsemi \underline{\mathrm{pu}_L \fatsemi \theta} \\ \text{identity law of } \theta : &\quad = \mathrm{pu}_M \quad . \end{aligned}$$

The last law is the composition law for `brun`. Begin with its right-hand side since it is longer:

$$\begin{aligned} & (\mathrm{brun}\,(\theta))^{\uparrow M \uparrow L} \fatsemi \mathrm{brun}\,(\theta) \fatsemi \mathrm{ftn}_M \\ \text{definition of brun} : &\quad = \theta^{\uparrow M \uparrow L} \fatsemi \theta \fatsemi \mathrm{ftn}_M \quad . \end{aligned}$$

We cannot simplify this expression any more, and yet it is still different from the left-hand side. So let us transform the left-hand side, hoping to obtain the same expression. In particular, we need to move ftn_M to the right and θ to the left:

$$\begin{aligned} & \mathrm{ftn}_{L\circ M} \fatsemi \mathrm{brun}\,(\theta) \\ \text{definitions of } \mathrm{ftn}_{L\circ M} \text{ and brun} : &\quad = \mathrm{sw}^{\uparrow L}_{L,M} \fatsemi \mathrm{ftn}_L \fatsemi \underline{\mathrm{ftn}^{\uparrow L}_M \fatsemi \theta} \\ \text{naturality law of } \theta : &\quad = \mathrm{sw}^{\uparrow L}_{L,M} \fatsemi \underline{\mathrm{ftn}_L \fatsemi \theta} \fatsemi \mathrm{ftn}_M \\ \text{composition law of } \theta : &\quad = \underline{\mathrm{sw}^{\uparrow L}_{L,M} \fatsemi \theta^{\uparrow L}} \fatsemi \theta \fatsemi \mathrm{ftn}_M \\ \text{functor composition} : &\quad = \underline{(\mathrm{sw}_{L,M} \fatsemi \theta)^{\uparrow L}} \fatsemi \theta \fatsemi \mathrm{ftn}_M \\ \text{monadic naturality law of } \mathrm{sw}_{L,M} : &\quad = \underline{\theta^{\uparrow M \uparrow L}} \fatsemi \theta \fatsemi \mathrm{ftn}_M \quad . \end{aligned}$$

We have transformed both sides of the law into the same expression.

The functor laws for `frun` are:

$$\mathrm{frun}\,(\mathrm{id}) = \mathrm{id} \quad , \qquad \mathrm{frun}\,(\phi) \fatsemi \mathrm{frun}\,(\chi) = \mathrm{frun}\,(\phi \fatsemi \chi) \quad .$$

Since $\mathrm{frun}\,(\phi) = \phi^{\uparrow L}$ in our case, these laws hold because they are the same as the functor laws of L.

To verify the nondegeneracy law for `brun`:

$$\begin{aligned} \text{expect to equal id}: \quad & \mathrm{flift} \fatsemi \mathrm{brun}\,(\theta) \\ \text{definitions of flift and brun}: \quad & = \mathrm{pu}_L \fatsemi \theta \\ \text{identity law of } \theta: \quad & = \mathrm{id} \quad . \end{aligned}$$

Finally, we need to check the monadic naturality laws of `flift` and `brun`:

$$\phi^{:M \rightsquigarrow N} \fatsemi \mathrm{flift}_L^N = \mathrm{flift}_L^M \fatsemi (\phi^{:M \rightsquigarrow N})^{\uparrow L} \quad , \qquad (\phi^{:M \rightsquigarrow N})^{\uparrow L} \fatsemi \mathrm{brun}_L^N\,(\theta_L) = \mathrm{brun}_L^M\,(\theta_L) \fatsemi \phi^{:M \rightsquigarrow N} \quad .$$

To verify these laws, use the naturality laws of pu_L and θ_L:

$$\begin{aligned} \phi \fatsemi \mathrm{flift} &= \underline{\phi \fatsemi \mathrm{pu}_L} = \mathrm{pu}_L \fatsemi \phi^{\uparrow L} = \mathrm{flift} \fatsemi \phi^{\uparrow L} \quad , \\ \phi^{\uparrow L} \fatsemi \mathrm{brun}\,(\theta_L) &= \underline{\phi^{\uparrow L} \fatsemi \theta_L} = \theta_L \fatsemi \phi = \mathrm{brun}\,(\theta_L) \fatsemi \phi \quad . \end{aligned}$$

(b) If the base monad is M, the runners are defined by:

$$\begin{aligned} \mathrm{frun}\,(\phi^{:L \rightsquigarrow N}) &: L \circ M \rightsquigarrow N \circ M \quad , \qquad \mathrm{frun}\,(\phi) = \phi \quad ; \\ \mathrm{brun}\,(\theta^{:M \rightsquigarrow \mathrm{Id}}) &: L \circ M \rightsquigarrow L \quad , \qquad \mathrm{brun}\,(\theta) = \theta^{\uparrow L} \quad . \end{aligned}$$

The laws of runners require that $\mathrm{frun}\,(\phi)$ and $\mathrm{brun}\,(\theta)$ must be monad morphisms, i.e., the identity and composition laws must hold:

$$\begin{aligned} &\mathrm{pu}_{L \circ M} \fatsemi \mathrm{frun}\,(\phi) = \mathrm{pu}_{N \circ M} \quad , \qquad \mathrm{ftn}_{L \circ M} \fatsemi \mathrm{frun}\,(\phi) = (\mathrm{frun}\,(\phi))^{\uparrow M \uparrow L} \fatsemi \mathrm{frun}\,(\phi) \fatsemi \mathrm{ftn}_{N \circ M} \quad , \\ &\mathrm{pu}_{L \circ M} \fatsemi \mathrm{brun}\,(\theta) = \mathrm{pu}_L \quad , \qquad \mathrm{ftn}_{L \circ M} \fatsemi \mathrm{brun}\,(\theta) = (\mathrm{brun}\,(\theta))^{\uparrow M \uparrow L} \fatsemi \mathrm{brun}\,(\theta) \fatsemi \mathrm{ftn}_L \quad . \end{aligned}$$

The monadic naturality laws of `swap` with respect to ϕ and χ are:

$$\mathrm{sw}_{L,M} \fatsemi \phi = \phi^{\uparrow M} \fatsemi \mathrm{sw}_{N,M} \quad , \qquad \mathrm{sw}_{L,M} \fatsemi \theta^{\uparrow L} = \theta \quad . \tag{14.26}$$

$$\begin{tikzcd} M^{L^A} \arrow[r, "\mathrm{sw}_{L,M}"] \arrow[d, "\phi^{\uparrow M}"'] & L^{M^A} \arrow[d, "\phi"] \\ M^{N^A} \arrow[r, "\mathrm{sw}_{N,M}"] & N^{M^A} \end{tikzcd} \qquad \begin{tikzcd} M^{L^A} \arrow[r, "\mathrm{sw}_{L,M}"] \arrow[dr, "\theta"'] & L^{M^A} \arrow[d, "\theta^{\uparrow L}"] \\ & L^A \end{tikzcd}$$

The first law to be proved is the identity law of $\mathrm{frun}\,(\phi)$:

$$\begin{aligned} \text{expect to equal } \mathrm{pu}_{N \circ M}: \quad & \mathrm{pu}_{L \circ M} \fatsemi \mathrm{frun}\,(\phi) \\ \text{definitions of frun and } \mathrm{pu}_{L \circ M}: \quad & = \mathrm{pu}_M \fatsemi \underline{\mathrm{pu}_L \fatsemi \phi} \\ \text{identity law of } \phi: \quad & = \mathrm{pu}_M \fatsemi \mathrm{pu}_N \\ \text{definition of } \mathrm{pu}_{N \circ M}: \quad & = \mathrm{pu}_{N \circ M} \quad . \end{aligned}$$

The next law is the composition law of frun (ϕ):

$$\begin{aligned}
\text{expect to equal } \mathrm{ftn}_T \fatsemi \phi : &\quad (\mathrm{frun}\,(\phi))^{\uparrow M \uparrow L} \fatsemi \mathrm{frun}\,(\phi) \fatsemi \mathrm{ftn}_{N \circ M} \\
\text{definitions of frun and } \mathrm{ftn}_{N \circ M} : &\quad = \phi^{\uparrow M \uparrow L} \fatsemi \underline{\phi \fatsemi \mathrm{sw}^{\uparrow N}_{N,M}} \fatsemi \mathrm{ftn}_N \fatsemi \mathrm{ftn}^{\uparrow N}_M \\
\text{naturality law of } \phi : &\quad = \underline{\phi^{\uparrow M \uparrow L} \fatsemi \mathrm{sw}^{\uparrow L}_{N,M}} \fatsemi \phi \fatsemi \mathrm{ftn}_N \fatsemi \mathrm{ftn}^{\uparrow N}_M \\
\text{monadic naturality of } \mathrm{sw}_{N,M} \text{ raised to } L : &\quad = \mathrm{sw}^{\uparrow L}_{L,M} \fatsemi \underline{\phi^{\uparrow L} \fatsemi \phi \fatsemi \mathrm{ftn}_N} \fatsemi \mathrm{ftn}^{\uparrow N}_M \\
\text{composition law of } \phi : &\quad = \mathrm{sw}^{\uparrow L}_{L,M} \fatsemi \mathrm{ftn}_L \fatsemi \underline{\phi \fatsemi \mathrm{ftn}^{\uparrow N}_M} \\
\text{naturality law of } \phi : &\quad = \underline{\mathrm{sw}^{\uparrow L}_{L,M} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}^{\uparrow L}_M} \fatsemi \phi \\
\text{definition of } \mathrm{ftn}_T : &\quad = \underline{\mathrm{ftn}_T} \fatsemi \phi \quad .
\end{aligned}$$

The next law is the identity law of brun (θ):

$$\begin{aligned}
\text{expect to equal } \mathrm{pu}_L : &\quad \mathrm{pu}_{L \circ M} \fatsemi \mathrm{brun}\,(\theta) \\
\text{definitions of brun and } \mathrm{pu}_{L \circ M} : &\quad = \mathrm{pu}_M \fatsemi \underline{\mathrm{pu}_L \fatsemi \theta^{\uparrow L}} = \underline{\mathrm{pu}_M \fatsemi \theta} \fatsemi \mathrm{pu}_L \\
\text{identity law of } \theta : &\quad = \mathrm{pu}_L \quad .
\end{aligned}$$

The last law is the composition law of brun (θ). Begin with its right-hand side:

$$\begin{aligned}
& (\mathrm{brun}\,(\theta))^{\uparrow M \uparrow L} \fatsemi \mathrm{brun}\,(\theta) \fatsemi \mathrm{ftn}_L \\
\text{definition of brun} : &\quad = \theta^{\uparrow L \uparrow M \uparrow L} \fatsemi \theta^{\uparrow L} \fatsemi \mathrm{ftn}_L \\
\text{functor composition} : &\quad = (\theta^{\uparrow L \uparrow M} \fatsemi \theta)^{\uparrow L} \fatsemi \mathrm{ftn}_L \\
\text{naturality law of } \theta : &\quad = (\theta \fatsemi \theta^{\uparrow L})^{\uparrow L} \fatsemi \mathrm{ftn}_L \quad .
\end{aligned}$$

We now transform the left-hand side, aiming to obtain the same expression. We need to move ftn_L to the right and θ to the left:

$$\begin{aligned}
\text{expect to equal } \theta^{\uparrow L} \fatsemi \theta^{\uparrow L \uparrow L} \fatsemi \mathrm{ftn}_L : &\quad \mathrm{ftn}_{L \circ M} \fatsemi \mathrm{brun}\,(\theta) \\
\text{definitions of } \mathrm{ftn}_{L \circ M} \text{ and brun} : &\quad = \mathrm{sw}^{\uparrow L}_{L,M} \fatsemi \mathrm{ftn}_L \fatsemi \underline{\mathrm{ftn}^{\uparrow L}_M \fatsemi \theta^{\uparrow L}} \\
\text{composition law of } \theta : &\quad = \mathrm{sw}^{\uparrow L}_{L,M} \fatsemi \underline{\mathrm{ftn}_L \fatsemi (\theta \fatsemi \theta)^{\uparrow L}} = \underline{\mathrm{sw}^{\uparrow L}_{L,M} \fatsemi (\theta \fatsemi \theta)^{\uparrow L \uparrow L}} \fatsemi \mathrm{ftn}_L \\
\text{functor composition} : &\quad = \underline{(\mathrm{sw}_{L,M} \fatsemi \theta^{\uparrow L})^{\uparrow L}} \fatsemi \theta^{\uparrow L \uparrow L} \fatsemi \mathrm{ftn}_L \\
\text{monadic naturality law of } \mathrm{sw}_{L,M} : &\quad = \underline{\theta^{\uparrow L}} \fatsemi \theta^{\uparrow L \uparrow L} \fatsemi \mathrm{ftn}_L \quad .
\end{aligned}$$

The functor laws of `frun` are:

$$\mathrm{frun}\,(\mathrm{id}) = \mathrm{id} \quad , \qquad \mathrm{frun}\,(\phi) \fatsemi \mathrm{frun}\,(\chi) = \mathrm{frun}\,(\phi \fatsemi \chi) \quad .$$

Since $\mathrm{frun}\,(\phi) = \phi$ in our case, these laws are satisfied.

To verify the nondegeneracy law of `brun`:

$$\begin{aligned}
\text{expect to equal id} : &\quad \mathrm{flift} \fatsemi \mathrm{brun}\,(\theta) \\
\text{definitions of flift and brun} : &\quad = \mathrm{pu}^{\uparrow L}_M \fatsemi \theta^{\uparrow L} = \mathrm{id}^{\uparrow L} = \mathrm{id} \quad .
\end{aligned}$$

Finally, we need to check the monadic naturality laws of `flift` and `brun`:

$$\phi^{:L \rightsquigarrow K} \fatsemi \mathrm{flift}^K_M = \mathrm{flift}^L_M \fatsemi \phi^{:L \rightsquigarrow K} \quad , \qquad \phi^{:L \rightsquigarrow K} \fatsemi \mathrm{brun}^K_M\,(\theta_M) = \mathrm{brun}^L_M\,(\theta_M) \fatsemi \phi^{:L \rightsquigarrow K} \quad .$$

To verify these laws, use the naturality law of $\phi^{:L \rightsquigarrow K}$:

$$\begin{aligned}
& \phi \fatsemi \mathrm{flift}^K_M = \underline{\phi \fatsemi \mathrm{pu}^{\uparrow K}_M} = \mathrm{pu}^{\uparrow L}_M \fatsemi \phi = \mathrm{flift}^L_N \fatsemi \phi \quad , \\
& \phi \fatsemi \mathrm{brun}^K_M\,(\theta_M) = \underline{\phi \fatsemi \theta^{\uparrow K}_M} = \theta^{\uparrow L}_M \fatsemi \phi = \mathrm{brun}^L_M\,(\theta_M) \fatsemi \phi \quad .
\end{aligned}$$

14.3.7 Summary of results

The following two statements summarize the derivations in Section 14.3.

Statement 14.3.7.1 (composed-outside) For a base monad L and a foreign monad M, the functor composition $L \circ M$ is a lawful monad transformer if a `swap` function $\text{sw}_{L,M} : M \circ L \rightsquigarrow L \circ M$ exists, satisfying the conditions of Statement 14.3.2.1 and the monadic naturality laws:

$$\text{sw}_{L,M} \fatsemi \phi^{\uparrow L} = \phi \fatsemi \text{sw}_{L,N} \quad , \qquad \text{sw}_{L,M} \fatsemi \theta = \theta^{\uparrow M} \quad ,$$

with arbitrary monad morphisms $\phi : M \rightsquigarrow N$ and $\theta : L \rightsquigarrow \text{Id}$. An additional law, $\text{sw}_{L,\text{Id}} = \text{id}$, follows from the conditions of Statement 14.3.2.1.

Statement 14.3.7.2 (composed-inside) For a base monad M and a foreign monad L, the functor composition $L \circ M$ is a lawful monad transformer if a `swap` function $\text{sw}_{L,M} : M \circ L \rightsquigarrow L \circ M$ exists, satisfying the conditions of Statement 14.3.2.1 and the monadic naturality laws:

$$\text{sw}_{L,M} \fatsemi \phi = \phi^{\uparrow M} \fatsemi \text{sw}_{N,M} \quad , \qquad \text{sw}_{L,M} \fatsemi \theta^{\uparrow L} = \theta \quad ,$$

with respect to arbitrary monad morphisms $\phi : L \rightsquigarrow N$ and $\theta : M \rightsquigarrow \text{Id}$. An additional law, $\text{sw}_{\text{Id},M} = \text{id}$, follows from the conditions of Statement 14.3.2.1.

In many cases, Statements 14.3.7.1–14.3.7.2 make monad transformer proofs quicker because the 6 laws of `swap` use simpler types and replace the 18 monad transformer laws.

14.4 Composed-inside transformers. Linear monads

In the previous section, we derived a number of properties common to all compositional transformers. We will now use those properties to prove the laws of composed-inside transformers. All those transformers have a "linear" monad as their base monad.

A monad M is **linear** if its type is of the form $M^A \stackrel{\text{def}}{=} P + Q \times A$, where P and Q are fixed types and Q is a monoid. (The name "linear" is motivated by considering the polynomial $P + Q \times A$, which is linear in the parameter A.) Well-known examples of linear monads are `Option`, `Either`, and `Writer`. The effect of the monad $M^A \stackrel{\text{def}}{=} P + Q \times A$ combines the effects of `Either` and `Writer` monads: a computation that can either succeed and append a log message (of type Q), or fail (with an error of type P).

A different (but also linear) monad is obtained from the composition of `Writer` and `Either`. The type constructor of this monad is $Q \times (P + A)$. This monad's effect is to append to a monoid Q and, at the same time, to compute a value of type A or to fail with an error of type P.

A composition of two linear monads $M_1^A \stackrel{\text{def}}{=} P_1 + Q_1 \times A$ and $M_2^A \stackrel{\text{def}}{=} P_2 + Q_2 \times A$ is again linear:

$$\begin{aligned} & P_1 + Q_1 \times (P_2 + Q_2 \times A) \\ \text{expand brackets}: \quad & \cong \underline{P_1 + Q_1 \times P_2} + \underline{Q_1 \times Q_2} \times A \\ \text{define new } P \text{ and } Q: \quad & \cong P + Q \times A \quad . \end{aligned}$$

By assumption, Q_1 and Q_2 are monoids, so $Q \stackrel{\text{def}}{=} Q_1 \times Q_2$ is also a monoid.

For a linear monad M and any foreign monad L, the functor composition $L \circ M$ is a monad. For example, the type constructor for the `OptionT` monad transformer can be written as

```
type OptionT[L[_], A] = L[Option[A]]
```

The `Option` type constructor must be composed *inside* the foreign monad `L`. This is the case for all linear monads. Also, linear monads are the only known examples of monads whose transformers are defined by composing inside the foreign monad.

14.4.1 Definitions of `swap` and `flatten`

To show that $T_M^{L,A} = L^{M^A}$ is a lawful monad transformer for the base monad $M^A \stackrel{\text{def}}{=} P + Q \times A$, we will implement a suitable `swap` function having the type signature:

$$\text{sw}_{L,M} : M^{L^A} \to L^{M^A} \quad ,$$

for the base monad M and an arbitrary foreign monad L. We will then prove that `swap` satisfies all the laws stated in Statement 14.3.7.2. This will guarantee that $T_M^{L,A}$ is a lawful monad transformer.

Expanding the definition of the type constructor $M^\bullet$, we can write the type signature of `swap` as:

$$\text{sw}_{L,M} : P + Q \times L^A \to L^{P+Q\times A} \quad .$$

We can map P to L^P by applying pu_L. We can also map $Q \times L^A \to L^{Q\times A}$ since L is a functor:

$$q \times l \to l \triangleright (a \to q \times a)^{\uparrow L} \quad .$$

It remains to combine these two functions. In the matrix notation, we write:

$$\text{sw}_{L,M} = \left| \begin{array}{c|c} & L^{P+Q\times A} \\ \hline P & (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \,⨾\, \text{pu}_L \\ Q \times L^A & q \times l \to l \triangleright (a^{:A} \to \mathbb{0}^{:P} + q \times a)^{\uparrow L} \end{array} \right| \quad . \tag{14.27}$$

```
type M[A, P, Q] = Either[P, (Q, A)]
def swap[L[_]: Monad, A, P, Q]: M[L[A]] => L[M[A]] = {
  case Left(p) => Monad[L].pure(Left(p))
  case Right((q, la)) => la.map(a => Right((q, a)))
}
```

Given this `swap` function, we define the `pure` and `flatten` methods for the transformed monad T (short notation pu_T and ftn_T) by the standard formulas:

$$\text{pu}_T = \text{pu}_M \,⨾\, \text{pu}_L \quad , \qquad \text{ftn}_T = \text{sw}^{\uparrow L} \,⨾\, \text{ftn}_L \,⨾\, \text{ftn}_M^{\uparrow L} \quad .$$

```
def pure[L[_]: Monad, A, P, Q: Monoid](x: A): L[M[A]] =
  Monad[L].pure(Right((Monoid[Q].empty, x)))
def flatten[L[_]: Monad, A, P, Q: Monoid](tt: L[M[L[M[A]]]]): L[M[A]] =
  tt.map(swap).flatten.map(_.flatten) // Assuming suitable implicits in scope.
```

14.4.2 The laws of `swap`. Proofs

To verify that the laws of `swap` hold, we will need to use the code for the methods fmap_M, ftn_M, and pu_M of the monad M:

$$f^{\uparrow M} = \left| \begin{array}{c|cc} & P & Q \times B \\ \hline P & \text{id} & \mathbb{0} \\ Q \times A & \mathbb{0} & q \times a \to q \times f(a) \end{array} \right| \quad ,$$

$$\text{pu}_M(a^{:A}) = 0^{:P} + q_0 \times a \quad , \quad \text{or equivalently:} \quad \text{pu}_M = \left| \begin{array}{c|cc} & P & Q \times A \\ \hline A & \mathbb{0} & a \to q_0 \times a \end{array} \right| \quad ,$$

$$\text{ftn}_M^{:M^{M^A} \to M^A} = \left| \begin{array}{c|cc} & P & Q \times A \\ \hline P & \text{id} & \mathbb{0} \\ Q \times P & q \times p \to p & \mathbb{0} \\ Q \times Q \times A & \mathbb{0} & q_1 \times q_2 \times a \to (q_1 \oplus q_2) \times a \end{array} \right| \quad .$$

We do not have to verify naturality since `swap` is defined as a fully parametric function.

The inner identity law We need to show that $\mathrm{pu}_L^{\uparrow M} \fatsemi \mathrm{sw} = \mathrm{pu}_L$:

$$\mathrm{pu}_L^{\uparrow M} \fatsemi \mathrm{sw} = \left\| \begin{array}{cc} \mathrm{id} & \mathbb{0} \\ \mathbb{0} & q \times a \to q \times \mathrm{pu}_L(a) \end{array} \right| \fatsemi \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \mathrm{pu}_L \\ q \times l \to l \triangleright (a^{:A} \to \mathbb{0}^{:P} + q \times a)^{\uparrow L} \end{array} \right|$$

$$\text{composition}: \quad = \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \mathrm{pu}_L \\ q \times a \to a \triangleright \underline{\mathrm{pu}_L \fatsemi (a^{:A} \to \mathbb{0}^{:P} + q \times a)^{\uparrow L}} \end{array} \right|$$

$$\mathrm{pu}_L\text{'s naturality}: \quad = \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \mathrm{pu}_L \\ (q \times a \to \mathbb{0}^{:P} + q \times a) \fatsemi \mathrm{pu}_L \end{array} \right| = \left| \begin{array}{c} P \\ Q \times A \end{array} \right\| \begin{array}{c} x^{:P} \to x + \mathbb{0}^{:Q\times A} \\ q \times a \to \mathbb{0}^{:P} + q \times a \end{array} \left| \fatsemi \mathrm{pu}_L \right.$$

$$\text{matrix notation}: \quad = \mathrm{id} \fatsemi \mathrm{pu}_L = \mathrm{pu}_L \quad .$$

The outer identity law We need to show that $\mathrm{pu}_M \fatsemi \mathrm{sw} = \mathrm{pu}_M^{\uparrow L}$:

$$\mathrm{pu}_M \fatsemi \mathrm{sw} = \left\| \begin{array}{cc} \mathbb{0} & l^{:L^A} \to q_0 \times l \end{array} \right| \fatsemi \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \mathrm{pu}_L \\ q \times l \to l \triangleright (x^{:A} \to \mathbb{0}^{:P} + q \times x)^{\uparrow L} \end{array} \right|$$

$$\text{composition}: \quad = l^{:L^A} \to l \triangleright \underline{(x^{:A} \to \mathbb{0}^{:P} + q_0 \times x)}^{\uparrow L}$$

$$\text{definition of } \mathrm{pu}_M: \quad = l \to l \triangleright \mathrm{pu}_M^{\uparrow L} = \mathrm{pu}_M^{\uparrow L} \quad .$$

The inner interchange law We need to show that $\mathrm{ftn}_L^{\uparrow M} \fatsemi \mathrm{sw} = \mathrm{sw} \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L$:

$$\mathrm{ftn}_L^{\uparrow M} \fatsemi \mathrm{sw} = \left\| \begin{array}{cc} \mathrm{id} & \mathbb{0} \\ \mathbb{0} & q \times l \to q \times (l \triangleright \mathrm{ftn}_L) \end{array} \right| \fatsemi \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \mathrm{pu}_L \\ q \times l \to l \triangleright (a \to \mathbb{0}^{:P} + q \times a)^{\uparrow L} \end{array} \right|$$

$$= \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \mathrm{pu}_L \\ q \times l \to l \triangleright \mathrm{ftn}_L \fatsemi (a \to \mathbb{0}^{:P} + q \times a)^{\uparrow L} \end{array} \right| \quad , \tag{14.28}$$

$$\mathrm{sw} \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L = \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \mathrm{pu}_L \\ q \times l \to l \triangleright (a \to \mathbb{0}^{:P} + q \times a)^{\uparrow L} \end{array} \right| \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L$$

$$= \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \mathrm{pu}_L \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \\ (q \times l \to l \triangleright (a \to \mathbb{0}^{:P} + q \times a)^{\uparrow L}) \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \end{array} \right| \quad .$$

Let us simplify each expression in the last column separately and then compare with the column in Eq. (14.28). Simplify the upper expression:

$$(x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \underline{\mathrm{pu}_L \fatsemi \mathrm{sw}^{\uparrow L}} \fatsemi \mathrm{ftn}_L$$

$$\mathrm{pu}_L\text{'s naturality}: \quad = (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \mathrm{sw} \fatsemi \underline{\mathrm{pu}_L \fatsemi \mathrm{ftn}_L}$$

$$\text{identity law of } L: \quad = (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \mathrm{sw} = x \to \underline{(x^{:P} + \mathbb{0}^{:Q\times A}) \triangleright \mathrm{sw}}$$

$$\text{definition of sw}: \quad = x \to (x + \mathbb{0}) \triangleright \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \mathrm{pu}_L \\ q \times l \to l \triangleright (a \to \mathbb{0}^{:P} + q \times a)^{\uparrow L} \end{array} \right| = (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \mathrm{pu}_L \quad .$$

This equals the upper expression in Eq. (14.28). Simplify the lower expression:

$$(q \times l \to l \triangleright (a \to \mathbb{0}^{:P} + q \times a)^{\uparrow L}) \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L$$

$$\triangleright\text{-notation}: \quad = q \times l \to l \triangleright (a \to \mathbb{0}^{:P} + q \times a)^{\uparrow L} \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \quad . \tag{14.29}$$

Simplify the expression $(a \to \mathbb{0}^{:P} + q \times a)^{\uparrow L} ⨾ \text{sw}^{\uparrow L}$ separately:

$$\begin{aligned}
& (a \to \mathbb{0}^{:P} + q \times a) ⨾ \text{sw} \\
\text{compute composition}: \quad & = a \to \underline{(\mathbb{0}^{:P} + q \times a) \triangleright \text{sw}} \\
\text{definition of sw}: \quad & = \underline{a \to a \triangleright} (x \to \mathbb{0}^{:P} + q \times x)^{\uparrow L} \\
\text{unexpand function}: \quad & = (x \to \mathbb{0}^{:P} + q \times x)^{\uparrow L} \quad .
\end{aligned} \tag{14.30}$$

Then we continue simplifying Eq. (14.29):

$$\begin{aligned}
& q \times l \to l \triangleright \underline{(a \to \mathbb{0}^{:P} + q \times a)^{\uparrow L} ⨾ \text{sw}^{\uparrow L}} ⨾ \text{ftn}_L \\
\text{use Eq. (14.30)}: \quad & = q \times l \to l \triangleright \underline{(x \to \mathbb{0}^{:P} + q \times x)^{\uparrow L \uparrow L} ⨾ \text{ftn}_L} \\
\text{naturality law of ftn}_L: \quad & = q \times l \to l \triangleright \text{ftn}_L ⨾ (x \to \mathbb{0}^{:P} + q \times x)^{\uparrow L} \quad .
\end{aligned}$$

This equals the lower expression in Eq. (14.28) after renaming the bound variable x to a.

The outer interchange law We need to show that $\text{ftn}_M ⨾ \text{sw} = \text{sw}^{\uparrow M} ⨾ \text{sw} ⨾ \text{ftn}_M^{\uparrow L}$. The left-hand side is written using the matrices for ftn_M and sw:

$$\text{ftn}_M ⨾ \text{sw} = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ q \times p \to p & \mathbb{0} \\ \mathbb{0} & q_1 \times q_2 \times a \to (q_1 \oplus q_2) \times a \end{array} \right| ⨾ \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}) ⨾ \text{pu}_L \\ q \times l \to l \triangleright (x \to \mathbb{0} + q \times x)^{\uparrow L} \end{array} \right|$$

$$= \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}) ⨾ \text{pu}_L \\ (q \times p \to p + \mathbb{0}) ⨾ \text{pu}_L \\ q_1 \times q_2 \times a \to a \triangleright (x \to \mathbb{0} + (q_1 \oplus q_2) \times x)^{\uparrow L} \end{array} \right| \quad . \tag{14.31}$$

We cannot simplify this matrix, so we hope to transform the right-hand side $(\text{sw}^{\uparrow M} ⨾ \text{sw} ⨾ \text{ftn}_M^{\uparrow L})$ to the same matrix. Begin by writing the matrix for $\text{sw}^{\uparrow M}$, expanding the rows for the input type $M^{M^{L^A}}$:

$$\text{sw}^{\uparrow M} = \left| \begin{array}{c|c|c} & P & Q \times L^{M^A} \\ \hline P & \text{id} & \mathbb{0} \\ Q \times P & \mathbb{0} & q \times p \to q \times \text{sw}\,(p + \mathbb{0}) \\ Q \times Q \times L^A & \mathbb{0} & q_1 \times q_2 \times l \to q_1 \times \text{sw}\,(\mathbb{0} + q_2 \times l) \end{array} \right|$$

$$= \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & q \times p \to q \times \text{pu}_L\,(p + \mathbb{0}) \\ \mathbb{0} & q_1 \times q_2 \times l \to q_1 \times (l \triangleright (x \to \mathbb{0} + q_2 \times x)^{\uparrow L}) \end{array} \right| \quad .$$

Then compute the composition $\text{sw}^{\uparrow M} ⨾ \text{sw}$:

$$\text{sw}^{\uparrow M} ⨾ \text{sw} = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & q \times p \to q \times \text{pu}_L\,(p + \mathbb{0}) \\ \mathbb{0} & q_1 \times q_2 \times l \to q_1 \times (l \triangleright (x \to \mathbb{0} + q_2 \times x)^{\uparrow L}) \end{array} \right| ⨾ \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}) ⨾ \text{pu}_L \\ q \times l \to (l \triangleright (x \to \mathbb{0} + q \times x)^{\uparrow L}) \end{array} \right|$$

$$= \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}) ⨾ \text{pu}_L \\ q \times p \to (p + \mathbb{0}) \triangleright \text{pu}_L ⨾ (x^{:M^A} \to \mathbb{0}^{:P} + q \times x)^{\uparrow L} \\ q_1 \times q_2 \times l \to l \triangleright (x^{:M^A} \to \mathbb{0}^{:P} + q_1 \times x)^{\uparrow L} ⨾ (x \to \mathbb{0} + q_2 \times x)^{\uparrow L} \end{array} \right|$$

$$= \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}) \,\fatsemi\, \mathrm{pu}_L \\ q \times p \to (\mathbb{0}^{:P} + q \times (p + \mathbb{0})) \triangleright \mathrm{pu}_L \\ q_1 \times q_2 \times l \to l \triangleright (x^{:M^A} \to \mathbb{0} + q_1 \times (\mathbb{0} + q_2 \times x))^{\uparrow L} \end{array} \right| \quad .$$

Now we need to post-compose $\mathrm{ftn}_M^{\uparrow L}$ with this column:

$$\mathrm{sw}^{\uparrow M} \fatsemi \mathrm{sw} \fatsemi \mathrm{ftn}_M^{\uparrow L} = \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}) \,\fatsemi\, \underline{\mathrm{pu}_L \fatsemi \mathrm{ftn}_M^{\uparrow L}} \\ (q \times p \to \mathbb{0}^{:P} + q \times (p + \mathbb{0})) \,\fatsemi\, \underline{\mathrm{pu}_L \fatsemi \mathrm{ftn}_M^{\uparrow L}} \\ q_1 \times q_2 \times l \to l \triangleright (x^{:M^A} \to \mathbb{0} + q_1 \times (\mathbb{0} + q_2 \times x))^{\uparrow L} \fatsemi \mathrm{ftn}_M^{\uparrow L} \end{array} \right|$$

$$\text{naturality law of } \mathrm{pu}_L : \quad = \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}) \,\fatsemi\, \mathrm{ftn}_M \fatsemi \mathrm{pu}_L \\ (q \times p \to \mathbb{0}^{:P} + q \times (p + \mathbb{0})) \,\fatsemi\, \mathrm{ftn}_M \fatsemi \mathrm{pu}_L \\ q_1 \times q_2 \times l \to l \triangleright (x^{:M^A} \to \underline{\mathrm{ftn}_M\ (\mathbb{0} + q_1 \times (\mathbb{0} + q_2 \times x))})^{\uparrow L} \end{array} \right|$$

$$\text{compute } \mathrm{ftn}_M(\ldots) : \quad = \left\| \begin{array}{c} (x^{:P} \to \mathrm{ftn}_M(x + \mathbb{0})) \,\fatsemi\, \mathrm{pu}_L \\ (q \times p \to \underline{\mathrm{ftn}_M(\mathbb{0}^{:P} + q \times (p + \mathbb{0}))}) \,\fatsemi\, \mathrm{pu}_L \\ q_1 \times q_2 \times l \to l \triangleright (x^{:M^A} \to \mathbb{0} + (q_1 \oplus q_2) \times x)^{\uparrow L} \end{array} \right|$$

$$\text{compute } \mathrm{ftn}_M(\ldots) : \quad = \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}) \,\fatsemi\, \mathrm{pu}_L \\ (q \times p \to p + \mathbb{0}) \,\fatsemi\, \mathrm{pu}_L \\ q_1 \times q_2 \times l \to l \triangleright (x^{:M^A} \to \mathbb{0} + (q_1 \oplus q_2) \times x)^{\uparrow L} \end{array} \right| \quad .$$

After renaming l to a, this is the same as the column in Eq. (14.31), which proves the law.

Monadic naturality laws We need to verify the laws of Statement 14.3.7.2:

$$\mathrm{sw}_{\mathrm{Id},M} = \mathrm{id} \quad , \qquad \mathrm{sw}_{L,M} \fatsemi \phi = \phi^{\uparrow M} \fatsemi \mathrm{sw}_{N,M} \quad , \qquad \mathrm{sw}_{L,M} \fatsemi \theta^{\uparrow L} = \theta \quad ,$$

for arbitrary monad morphisms $\phi : L \rightsquigarrow N$ and $\theta : M \rightsquigarrow \mathrm{Id}$.

The first law is the swap identity law, $\mathrm{sw}_{\mathrm{Id},M} = \mathrm{id}$:

$$\mathrm{sw}_{\mathrm{Id},M}$$

$$\text{Eq. (14.27) with } L = \mathrm{Id} : \quad = \left| \begin{array}{c} P \\ Q \times \mathrm{Id}^A \end{array} \right\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q \times A}) \,\fatsemi\, \mathrm{pu}_{\mathrm{Id}} \\ q \times l \to l \triangleright (a^{:A} \to \mathbb{0}^{:P} + q \times a)^{\uparrow \mathrm{Id}} \end{array} \Big|$$

$$\text{matrix notation} : \quad = \left| \begin{array}{c} P \\ Q \times A \end{array} \right\| \begin{array}{c} x^{:P} \to x + \mathbb{0}^{:Q \times A} \\ q \times a \to \mathbb{0}^{:P} + q \times a \end{array} \Big| = \mathrm{id} \quad .$$

Begin with the left-hand side of the second law:

$$\mathrm{sw}_{L,M} \fatsemi \phi$$

$$\text{definition of } \mathrm{sw}_{L,M} : \quad = \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q \times A}) \,\fatsemi\, \mathrm{pu}_L \\ q \times l \to l \triangleright (a \to \mathbb{0}^{:P} + q \times a)^{\uparrow L} \end{array} \right| \underline{\fatsemi \phi}$$

$$\text{compose with } \phi : \quad = \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q \times A}) \,\fatsemi\, \mathrm{pu}_L \fatsemi \phi \\ q \times l \to l \triangleright \underline{(a \to \mathbb{0}^{:P} + q \times a)^{\uparrow L} \fatsemi \phi} \end{array} \right|$$

$$\text{naturality law of } \phi : \quad = \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q \times A}) \,\fatsemi\, \mathrm{pu}_N \\ q \times l \to l \triangleright \phi \fatsemi (a \to \mathbb{0}^{:P} + q \times a)^{\uparrow N} \end{array} \right| \quad .$$

The right-hand side is:

$$\phi^{\uparrow M} \fatsemi \text{sw}_{N,M} = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & q \times l \to q \times (l \triangleright \phi) \end{array} \right| \fatsemi \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \text{pu}_N \\ q \times n \to n \triangleright (a \to \mathbb{0}^{:P} + q \times a)^{\uparrow N} \end{array} \right|$$

$$\text{composition}: \quad = \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \text{pu}_N \\ q \times l \to l \triangleright \phi \fatsemi (a \to \mathbb{0}^{:P} + q \times a)^{\uparrow N} \end{array} \right| \quad .$$

Both sides of the second law are now shown to be equal.

The left-hand side of the third law is:

$$\text{sw}_{L,M} \fatsemi \theta^{\uparrow L}$$

$$\text{compose with } \theta^{\uparrow L}: \quad = \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \underline{\text{pu}_L \fatsemi \theta^{\uparrow L}} \\ q \times l \to l \triangleright (a \to \mathbb{0}^{:P} + q \times a)^{\uparrow L} \fatsemi \theta^{\uparrow L} \end{array} \right|$$

$$\text{naturality law of } \text{pu}_L: \quad = \left\| \begin{array}{c} (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \theta \fatsemi \text{pu}_L \\ q \times l \to l \triangleright \left(a \to \theta(\mathbb{0}^{:P} + q \times a)\right)^{\uparrow L} \end{array} \right| \quad . \tag{14.32}$$

We expect this to equal the right-hand side, which we write as:

$$m^{:M^{L^A}} \to \theta(m)$$

$$\text{matrix notation}: \quad = \left\| \begin{array}{c} x^{:P} \to \theta(x + \mathbb{0}^{:Q\times L^A}) \\ q \times l \to \theta(\mathbb{0}^{:P} + q \times l) \end{array} \right| \quad . \tag{14.33}$$

Now consider each line in Eq. (14.32) separately. The upper line can be transformed as:

$$(x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \underline{\theta \fatsemi \text{pu}_L}$$

$$\text{naturality law of } \theta: \quad = (x^{:P} \to x + \mathbb{0}^{:Q\times A}) \fatsemi \underline{\text{pu}_L^{\uparrow M}} \fatsemi \theta$$

$$\text{definition of } {}^{\uparrow M}: \quad = x^{:P} \to \left| \begin{array}{cc} x & \mathbb{0} \end{array} \right| \triangleright \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & q \times l \to q \times \text{pu}_L(l) \end{array} \right| \fatsemi \theta$$

$$\text{matrix composition}: \quad = x^{:P} \to (x + \mathbb{0}^{:Q\times L^A}) \triangleright \theta \quad .$$

This is now equal to the upper line of Eq. (14.33).

To proceed with the proof for the lower line of Eq. (14.32), we need to evaluate the monad morphism $\theta : M^A \to A$ on a specific value of type M^A of the form $\mathbb{0} + q \times a$. We note that the value $\theta(\mathbb{0} + q \times a)$ must be of type A and must be computed in the same way for all types A, because θ is a natural transformation. It seems clear that the result cannot depend on the value $q^{:Q}$ since Q is a type unrelated to A. In other words, we expect that $\theta(\mathbb{0} + q \times a) = a$ as a consequence of naturality of θ. To derive this formally, we use the trick of starting with a unit type ($\mathbb{1}$) and then mapping it to A within the naturality law. For any values $q^{:Q}$ and $a^{:A}$, we define:

$$m^{:P+Q\times A} \stackrel{\text{def}}{=} \mathbb{0}^{:P} + q \times a \quad , \qquad m_1^{:P+Q\times\mathbb{1}} \stackrel{\text{def}}{=} \mathbb{0}^{:P} + q \times 1 \quad .$$

We can compute m from m_1 if we replace 1 by a under the functor M. To write this as a formula, define the function $f^{:\mathbb{1}\to A}$ as $f \stackrel{\text{def}}{=} (_ \to a)$ using the fixed value a. Then we have $m = m_1 \triangleright f^{\uparrow M}$. Now we apply both sides of the naturality law $f^{\uparrow M} \fatsemi \theta = \theta \fatsemi f$ to the value m_1:

$$m_1 \triangleright f^{\uparrow M} \fatsemi \theta = m_1 \triangleright \theta \fatsemi f \quad .$$

Simplify the left-hand side to:

$$\underline{m_1 \triangleright f^{\uparrow M}} \fatsemi \theta = m \triangleright \theta = \theta(\mathbb{0}^{:P} + q \times a) \quad .$$

Simplify the right-hand side to:

$$m_1 \triangleright \theta \fatsemi f = f\,(\theta\,(m_1)) = a \quad ,$$

since the function f always returns a. So, we have:

$$\theta(\mathbb{0}^{:P} + q \times a) = a \quad . \tag{14.34}$$

We can now compute the second line in Eq. (14.32) as:

$$q \times l \to l \triangleright \big(a \to \theta(\mathbb{0}^{:P} + q \times a)\big)^{\uparrow L}$$

$$\text{use Eq. (14.34)}: \quad = q \times l \to l \triangleright \underline{\big(a \to a\big)^{\uparrow L}} = q \times l \to l \triangleright \mathrm{id}^{\uparrow L} = q \times l \to l \quad .$$

The second line in Eq. (14.33) is the same function, namely $q \times l \to l$.

This concludes the proof of the laws of `swap` for linear monads. It follows that all linear monads have lawful monad transformers that compose inside the foreign monad.

14.4.3 Composition of transformers for linear monads

We have just shown that any linear monad $M^A \stackrel{\text{def}}{=} P + Q \times A$ has a `swap` function obeying the laws necessary for a composed-inside transformer. The other type of linear monad is $W^A \stackrel{\text{def}}{=} Q \times (P + A)$. Do we need to show separately that the monad W has a lawful `swap` function? No; instead we may use the stacking property of monad transformers (see Sections 14.2.8–14.2.9). The monad W is a functor composition of the `Writer` monad $Q \times A$ with the `Either` monad $P + A$, which is the same as applying the `Either` monad's transformer to the `Writer` monad. Because of the transformer stacking property, the monad transformer of W works as a composed-inside transformer.

We can show that the functor composition of any two linear monads has a composed-inside transformer. Suppose M_1 and M_2 are linear monads, so their transformers are also composed-inside:

$$M_1 \triangleleft N = T^N_{M_1} = N \circ M_1 \quad , \qquad M_2 \triangleleft N = T^N_{M_2} = N \circ M_2 \quad .$$

The functor composition of M_1 and M_2 can be seen as a monad stack:

$$M_1 \circ M_2 = T^{M_1}_{M_2} = M_2 \triangleleft M_1 \quad .$$

What is the transformer for the monad $M_1 \circ M_2$? For any foreign monad N, we have:

$$(M_1 \circ M_2) \triangleleft N = (M_2 \triangleleft M_1) \triangleleft N = M_2 \triangleleft (M_1 \triangleleft N) = M_2 \triangleleft (N \circ M_1) = N \circ M_1 \circ M_2 \quad .$$

This is a lawful monad transformer, as we have seen in Section 14.2.8. So, this is the transformer for $M_1 \circ M_2$, and it is of the composed-inside kind.

14.5 Composed-outside transformers. Rigid monads

Section 14.4 shows that the composed-inside transformers are available for monads of the form $M^A = P + Q \times A$ and $M^A = Q \times (P + A)$, called "linear". No other examples of composed-inside transformers are known. It turns out that the composed-*outside* transformers are available for a significantly wider range of monads. Those monads are called "rigid" in this book[15] because one of their general properties is having a single, fixed "data shape" (Statement 14.9.3.7 below).

[15] Monads of this kind do not seem to have an already accepted name.

Definition A monad R is **rigid** if it has a lawful composed-outside monad transformer, $T_R^M = R \circ M$, where M is a foreign monad.

This definition states the required properties but does not explain which monads are rigid (or how to recognize a non-rigid monad). We will answer these questions below.

Two examples of rigid monads are the `Reader` monad and the `Sel` (selector) monad:[16]

$$\begin{aligned} \text{(the } \texttt{Reader} \text{ monad)}: &\quad \text{Reader}^{Z,A} \stackrel{\text{def}}{=} Z \to A \quad , \\ \text{(the } \texttt{Sel} \text{ monad)}: &\quad \text{Sel}^{Z,A} \stackrel{\text{def}}{=} (A \to Z) \to A \quad , \end{aligned}$$

where Z is a fixed type. These monads have composed-outside transformers:

$$\begin{aligned} \text{(the } \texttt{ReaderT} \text{ transformer)}: &\quad T_{\text{Reader}}^{M,A} \stackrel{\text{def}}{=} Z \to M^A \quad , \\ \text{(the } \texttt{SelT} \text{ transformer)}: &\quad T_{\text{Sel}}^{M,A} \stackrel{\text{def}}{=} (M^A \to Z) \to M^A \quad . \end{aligned}$$

To build intuition for rigid monads, we will perform structural analysis looking for general constructions that create new rigid monads or combine existing rigid monads into new ones. In this section, we will prove that the following three constructions produce rigid monads:

1. "Product": $P^A \times R^A$ is a rigid monad if P and R are rigid monads.
2. "Composition": $P \circ R$ is a rigid monad if P and R are rigid monads.
3. "Choice": $C^A \stackrel{\text{def}}{=} H^A \to A$ is a rigid monad if H is any contrafunctor.

It is not known whether these constructions are the only possible ways of creating new rigid monads. Other open questions about rigid monads are listed in Problem 15.3.2.8.

14.5.1 Rigid monad construction 1 (product)

Statement 14.5.1.1 The product of rigid monads, $R_1^A \times R_2^A$, is a rigid monad.

Proof If we show that $R_1^A \times R_2^A$ has a composed-outside transformer, it will follow (by definition) that $R_1^A \times R_2^A$ is a rigid monad. Due to Statement 14.6.1.1 (whose proof does not depend on any of the results of this section), a lawful monad transformer $T_{R_1 \times R_2}^{M,A}$ for the product monad $R_1^A \times R_2^A$ is the product of transformers: for any foreign monad M,

$$T_{R_1}^{M,A} \times T_{R_2}^{M,A} = R_1^{M^A} \times R_2^{M^A} \quad .$$

This is the required composed-outside transformer $(T_{R_1 \times R_2}^{M,A})$ for the monad $R_1^A \times R_2^A$.

14.5.2 Rigid monad construction 2 (composition)

Functor composition is the second construction that produces rigid monads. This is a consequence of the properties of monad transformer stacks.

Statement 14.5.2.1 The composition $R_1^{R_2^\bullet}$ of two rigid monads R_1 and R_2 is also a rigid monad.

Proof Since R_1 is rigid, its outside-composition $R_1 \circ M$ with any other monad M is a monad. So $R_1 \circ R_2$ is a monad. To show that $R_1 \circ R_2$ is a rigid monad, we need to show that its monad transformer is of the composed-outside kind. By Statement 14.2.8, the stacking of monad transformers T_{R_1} and T_{R_2} is a lawful monad transformer. Since the transformers for R_1 and R_2 are of the composed-outside kind, $T_{R_1}^M = R_1 \circ M$ and $T_{R_2}^M = R_2 \circ M$, the stack of transformers is expressed as

$$T_{R_1}^{T_{R_2}^M} = R_1 \circ T_{R_2}^M = R_1 \circ (R_2 \circ M) = R_1 \circ R_2 \circ M \quad .$$

Therefore $T^M \stackrel{\text{def}}{=} R_1 \circ R_2 \circ M$ is a lawful monad transformer applied to the foreign monad M. This shows, by definition of a rigid monad, that $R_1 \circ R_2$ is a rigid monad.

[16] See http://math.andrej.com/2008/11/21/

Example 14.5.2.2 Consider the functor composition of the `Sel` monad $R_1^A \stackrel{\text{def}}{=} (A \to Q) \to A$ and the `Reader` monad $R_2^A \stackrel{\text{def}}{=} Z \to A$:

$$P^A \stackrel{\text{def}}{=} ((Z \to A) \to Q) \to Z \to A \quad .$$

It follows from Statement 14.5.2.1 that the functor P is a rigid monad; so P's transformer is of the composed-outside kind. The the transformed monad for any foreign monad M is:

$$T^A \stackrel{\text{def}}{=} ((Z \to M^A) \to Q) \to Z \to M^A \quad .$$

To define the monad methods for T, we need to have the definitions of the transformers $T_{R_1}^M$ and $T_{R_2}^M$. Since both the `Sel` and the `Reader` monads are special cases of the `Choice` monad construction (Section 14.5.3) where the contrafunctor H is chosen to be $H^A \stackrel{\text{def}}{=} A \to Q$ and $H^A \stackrel{\text{def}}{=} Z$ respectively, we can use Eq. (14.37) to define the `flatMap` methods for the transformers $T_{R_1}^M$ and $T_{R_2}^M$:

```
type R1[A] = (A => Q) => A
def map_R1[A, B](r1: R1[A])(f: A => B): R1[B] = { (b2q: B => Q) => f(r1(f andThen b2q)) }
def flatMap_R1[A, B, M[_]: Monad](r1: R1[M[A]])(f: A => R1[M[B]]): R1[M[B]] = {
  (q: M[B] => Q) => map_R1(r1){ (m: M[A]) => m.flatMap(x => f(x)(q)) }(q)
}

type R2[A] = Z => A
def map_R2[A, B](r2: R2[A])(f: A => B): R2[B] = { r2 andThen f }
def flatMap_R2[A, B, M[_]: Monad](r2: R2[M[A]])(f: A => R2[M[B]]): R2[M[B]] = {
  z => map_R2(r2){ (m: M[A]) => m.flatMap(x => f(x)(z)) }(z)
}
```

Now we can define the `flatMap` method for T by using the monad $T_{R_2}^M$ instead of M in the `flatMap` method for $T_{R_1}^M$:

```
type T[A] = R1[R2[A]]
def flatMap_T[A, B, M[_]: Monad](t: T[M[A]])(f: A => T[M[B]]): T[M[B]] = {
  (q: R2[M[B]] => Q) => map_R1(t){ (m: R2[M[A]]) => flatMap_R2(m)(x => f(x)(q)) }(q)
}
```

14.5.3 Rigid monad construction 3 (choice)

The construction called the **choice** monad, $R^A \stackrel{\text{def}}{=} H^A \to A$, defines a rigid monad R for any given contrafunctor H. This monad somehow "chooses" a value of type A. The contrafunctor H may *consume* values of type A (and could check some conditions on those values). For example, H could be a constant contrafunctor ($H^A \stackrel{\text{def}}{=} Q$), a function ($H^A \stackrel{\text{def}}{=} A \to Q$), or a more complicated contrafunctor.

Different contrafunctors H give various rigid monads such as $R^A \stackrel{\text{def}}{=} \mathbb{1}$ (the unit monad), $R^A \stackrel{\text{def}}{=} A$ (the identity monad), $R^A \stackrel{\text{def}}{=} Z \to A$ (the `Reader`), and $\text{Sel}^{Q,A} \stackrel{\text{def}}{=} (A \to Q) \to A$ (the selector monad).

The selector monad represents the effect of somehow "selecting" a value of type A using a function of type $A \to Q$. A simple example of a selector monad is found by setting $Q \stackrel{\text{def}}{=} \mathbb{2}$ (where $\mathbb{2}$ denotes the `Boolean` type). One may implement a function of type $(A \to \mathbb{2}) \to A$ that *somehow* finds a value of type A that might satisfy the given predicate of type $A \to \mathbb{2}$. The intention is to return, if possible, a value that satisfies the predicate. If no such value can be found, some value of type A is still returned.

A closely related monad is the **search monad**, $\text{Search}^{Q,A} \stackrel{\text{def}}{=} (A \to \mathbb{1} + P) \to \mathbb{1} + A$. This (non-rigid) monad returns an empty value $1 + \mathbb{0}^{:A}$ if no value satisfying the "predicate" (a function of type $A \to \mathbb{1} + P$) is found. There is a mapping from the selector monad to the search monad, implemented by checking whether the value returned by the selector monad does actually satisfy the predicate. However, that mapping is not a monad morphism (Exercise 14.9.5.12).

The selector monad is superficially similar to the continuation monad, and there is a monad morphism in one direction between `Sel` and `Cont` (Exercise 14.9.5.13).

Assume that H is a contrafunctor and M is a monad, and denote for brevity:

$$R^A \stackrel{\text{def}}{=} H^A \to A \quad , \qquad T^A \stackrel{\text{def}}{=} R^{M^A} = H^{M^A} \to M^A \quad .$$

Let us now implement a suitable `swap` function and verify its laws. Then the results of Section 14.3 will ensure that T is a lawful monad transformer.

Statement 14.5.3.1 The `swap` function and the monad methods for T can be written as:

$$\text{sw}_{R,M} : M^{R^A} \to R^{M^A} \quad , \qquad \text{sw}_{R,M} \stackrel{\text{def}}{=} m^{:M^{R^A}} \to q^{:H^{M^A}} \to m \triangleright (r \to q \triangleright \text{pu}_M^{\downarrow H} \fatsemi r)^{\uparrow M} \quad , \tag{14.35}$$

$$\text{pu}_T : A \to H^{M^A} \to M^A \quad , \qquad \text{pu}_T \stackrel{\text{def}}{=} \text{pu}_M \fatsemi \text{pu}_R = a^{:A} \to _^{:H^{M^A}} \to \text{pu}_M(a) \quad ,$$

$$\text{ftn}_T \stackrel{\text{def}}{=} t^{:T^{T^A}} \to k^{:H^{M^A}} \to k \triangleright \left(t \triangleright (\text{flm}_M(r^{:R^{M^A}} \to r(k)))^{\uparrow R}\right) = \text{flm}_R\,(t \to q \to t \triangleright \text{flm}_M(r \to r(q))) \quad ,$$

$$\text{flm}_T(f) = \text{flm}_R\big(y \to q \to y \triangleright \text{flm}_M(x \to q \triangleright (x \triangleright f))\big) \tag{14.36}$$

$$= t^{:R^{M^A}} \to q^{:H^{M^B}} \to q \triangleright \left(t \triangleright \left(\text{flm}_M(x^{:A} \to q \triangleright (x \triangleright f))\right)^{\uparrow R}\right) \quad . \tag{14.37}$$

The monad R's methods may be defined by:

$$\text{pu}_R : A \to H^A \to A \quad , \qquad \text{pu}_R \stackrel{\text{def}}{=} a \to _^{:H^A} \to a \quad ,$$

$$\text{ftn}_R : (H^{H^A \to A} \to A) \to H^A \to A \quad , \qquad \text{ftn}_R \stackrel{\text{def}}{=} r^{:R^{R^A}} \to k^{:H^A} \to r \triangleright (q^{:R^A} \to q(k))^{\uparrow R} \quad ,$$

$$\text{flm}_R(g^{:A \to R^B}) \stackrel{\text{def}}{=} r^{:R^A} \to k^{:H^B} \to k \triangleright \left(r \triangleright (a^{:A} \to g(a)(k))^{\uparrow R}\right) \quad . \tag{14.38}$$

Proof We begin by deriving the code for the `swap` function. The monad T's methods depend on the monad R, whose code we will need to write next. Finally, we will obtain alternative formulas for flm_T and flm_R that will be useful in derivations.

The code for `swap` Throughout the proof, we assume M to be a fixed, lawful monad. Begin implementing `swap` by using typed holes:

$$\text{sw}_{R,M} : M^{R^A} \to H^{M^A} \to M^A \quad , \qquad \text{sw}_{R,M} \stackrel{\text{def}}{=} m^{:M^{H^A \to A}} \to q^{:H^{M^A}} \to ???^{:M^A} \quad .$$

Since the monad M is arbitrary and no values of type A are given, the only way of computing a value of type M^A is by using the given value m. We could transform $m : M^{H^A \to A}$ to a value of type M^A if we could transform a value of type $H^A \to A$ into a value of type A:

$$\text{sw}_{R,M} \stackrel{\text{def}}{=} m^{:M^{H^A \to A}} \to q^{:H^{M^A}} \to m \triangleright \left(r^{:H^A \to A} \to ???^{:A}\right)^{\uparrow M} \quad .$$

Lacking any given values of type A, we have to apply r to a value of type H^A:

$$\text{sw}_{R,M} \stackrel{\text{def}}{=} m^{:M^{H^A \to A}} \to q^{:H^{M^A}} \to m \triangleright \left(r^{:H^A \to A} \to r(???^{:H^A})\right)^{\uparrow M} \quad .$$

We have a value $q : H^{M^A}$ and a method $\text{pu}_M : A \to M^A$. This gives us the solution:

$$\text{sw}_{R,M} \stackrel{\text{def}}{=} m^{:M^{H^A \to A}} \to q^{:H^{M^A}} \to m \triangleright \left(r^{:H^A \to A} \to r(q \triangleright \text{pu}_M^{\downarrow H})\right)^{\uparrow M} \quad .$$

Rewrite the formula in the $\triangleright$-notation to obtain Eq. (14.35):

$$\text{sw}_{R,M} : M^{R^A} \to R^{M^A} \quad , \qquad \text{sw}_{R,M} \stackrel{\text{def}}{=} m^{:M^{R^A}} \to q^{:H^{M^A}} \to m \triangleright (r^{:H^A \to A} \to q \triangleright \text{pu}_M^{\downarrow H} \fatsemi r)^{\uparrow M} \quad .$$

Equivalently, we may write the code of $\text{sw}_{R,M}$ as:

$$q \triangleright \left(m \triangleright \text{sw}_{R,M}\right) = m \triangleright (r \to q \triangleright \text{pu}_M^{\downarrow H} \fatsemi r)^{\uparrow M} \quad . \tag{14.39}$$

We will use these formulas for `swap` in proofs of its various laws.

The code for the monad R We now derive the code for the monad R's methods. That monad's laws were proved in Statement 10.2.8.6 using flipped types of Kleisli morphisms and the flipped Kleisli composition ($\tilde{\diamond}_R$). To restore the standard type signatures for `pure` and `flatMap`, we begin by flipping the arguments of $\tilde{\text{pu}}_R$:

$$\text{use Eq. (10.20)}: \quad \text{pu}_R \stackrel{\text{def}}{=} \text{flip}\,(\tilde{\text{pu}}_R) = \text{flip}\,(_^{:H^A} \to a^{:A} \to a) = a^{:A} \to _^{:H^A} \to a \quad .$$

To recover the Kleisli composition ($\diamond_R$), we need to flip both arguments of $\tilde{\diamond}_R$:

$$\begin{aligned}
& f^{:A\to H^B\to B} \diamond_R g^{:B\to H^C\to C} \stackrel{\text{def}}{=} \text{flip}\big(\text{flip}\,(f)\,\tilde{\diamond}_R \text{flip}\,(g)\big) \\
\text{use Eq. (10.19)}: \quad & = \text{flip}\big(k^{:H^C} \to \tilde{f}\big(k \triangleright (\tilde{g}(k))^{\downarrow H}\big) \fatsemi \tilde{g}(k)\big) \\
\text{expand function}: \quad & = \text{flip}\big(k^{:H^C} \to a^{:A} \to a \triangleright \tilde{f}\big(k \triangleright (\tilde{g}(k))^{\downarrow H}\big) \fatsemi \tilde{g}(k)\big) \\
\text{definition of flip}: \quad & = a^{:A} \to k^{:H^C} \to a \triangleright \tilde{f}\big(k \triangleright (\tilde{g}(k))^{\downarrow H}\big) \fatsemi \tilde{g}(k) \quad ,
\end{aligned}$$

where the flipped versions of the functions f and g are defined by:

$$\begin{aligned}
& \tilde{f} : H^B \to A \to B \quad , \qquad \tilde{f} \stackrel{\text{def}}{=} h^{:H^B} \to a^{:A} \to f(a)(h) \quad , \qquad a \triangleright \tilde{f}(h) = f(a)(h) \quad , \\
& \tilde{g} : H^C \to B \to C \quad , \qquad \tilde{g} \stackrel{\text{def}}{=} h^{:H^C} \to b^{:B} \to g(b)(h) \quad , \qquad b \triangleright \tilde{g}(h) = g(b)(h) \quad .
\end{aligned}$$

Substituting these definitions, we compute:

$$\begin{aligned}
& f \diamond_R g = a^{:A} \to k^{:H^C} \to a \triangleright \tilde{f}\big(k \triangleright (\tilde{g}(k))^{\downarrow H}\big) \underline{\fatsemi} \tilde{g}(k) \\
\triangleright\text{-notation}: \quad & = a \to k \to a \triangleright \underline{\tilde{f}}\big(k \triangleright (\tilde{g}(k))^{\downarrow H}\big) \triangleright \tilde{g}(k) \\
& = a \to k \to f(a)(k \underline{\triangleright (\tilde{g}(k))}^{\downarrow H}) \underline{\triangleright \tilde{g}(k)} = a \to k \to g\big(f(a)(k \triangleright (b \to g(b)(k))^{\downarrow H})\big)(k) \quad .
\end{aligned}$$

This code can be shortened by defining a helper function $\gamma(g,k)$, also denoted $\gamma_{g,k}$ for brevity:

$$\begin{aligned}
& \gamma_{g,k} : B \to C \quad , \qquad \gamma_{g,k} \stackrel{\text{def}}{=} b \to g(b)(k) = \tilde{g}(k) \quad , \\
\text{then write}: \quad & f \diamond_R g = a^{:A} \to k^{:H^C} \to f(a)(k \triangleright \gamma_{g,k}^{\downarrow H}) \triangleright \gamma_{g,k} \quad .
\end{aligned}$$

The next step is to derive the code for flm_R:

$$\begin{aligned}
& \text{flm}_R(g^{:A\to R^B}) \stackrel{\text{def}}{=} \text{id}^{:R^A\to R^A} \diamond_R g = r^{:R^A} \to k^{:H^B} \to \text{id}\,(r)\,(k \triangleright \gamma_{g,k}^{\downarrow H}) \triangleright \gamma_{g,k} \\
& = r \to k \to r(k \triangleright \gamma_{g,k}^{\downarrow H}) \triangleright \gamma_{g,k} = r \to k \to g\big(r(k \triangleright (b \to g(b)(k))^{\downarrow H})\big)(k) \quad .
\end{aligned}$$

Finally, we find ftn_R as:

$$\text{ftn}_R \stackrel{\text{def}}{=} \text{flm}_R(\text{id}^{:R^A\to R^A}) = r^{:R^{R^A}} \to k^{:H^A} \to r(k \triangleright (q^{:R^A} \to q(k))^{\downarrow H})\big)(k) \quad .$$

The last two formulas may be shortened if we notice that the lifting to R is expressed as:

$$f^{\uparrow R} = r^{:H^A\to A} \to k^{:H^B} \to f(r(k \triangleright f^{\downarrow H})) = r^{:R^A} \to f^{\downarrow H} \fatsemi r \fatsemi f \quad .$$

So, we may equivalently write the code for flm_R and ftn_R as:

$$\text{flm}_R(g^{:A\to R^B}) \stackrel{\text{def}}{=} r^{:R^A} \to k^{:H^B} \to k \triangleright \big(r \triangleright \gamma_{g,k}^{\uparrow R}\big) \quad , \quad \text{ftn}_R \stackrel{\text{def}}{=} r^{:R^{R^A}} \to k^{:H^B} \to k \triangleright \big(r \triangleright (q^{:R^A} \to q(k))^{\uparrow R}\big) \quad .$$

This corresponds to Eq. (14.38).

Deriving ftn_T from $\text{sw}_{R,M}$ Given the code of `swap` defined by Eq. (14.35), we can recover the code of ftn_T by using Eq. (14.11), where we need to use the monad R instead of L:

$$\begin{aligned}
\text{ftn}_T &= \underline{\text{sw}^{\uparrow R} \fatsemi \text{ftn}_R} \fatsemi \text{ftn}_M^{\uparrow R} = \text{flm}_R(\text{sw}) \fatsemi \text{ftn}_M^{\uparrow R} \\
\text{naturality law of } \text{flm}_R: \quad &= \text{flm}_R(\text{sw} \fatsemi \text{ftn}_M^{\uparrow R}) \quad . \qquad (14.40)
\end{aligned}$$

To proceed, we need to simplify $\text{sw} \fatsemi \text{ftn}_M^{\uparrow R}$:

$$\begin{aligned}
& \text{sw} \fatsemi \text{ftn}_M^{\uparrow R} \\
\text{definitions}: \quad &= \left(m \to q \to m \triangleright \left((r \to \underline{q} \triangleright \text{pu}_M^{\downarrow H} \fatsemi r)\right)^{\uparrow M}\right) \fatsemi \left(\underline{r \to \text{ftn}_M^{\downarrow H} \fatsemi r \fatsemi \text{ftn}_M}\right) \\
\text{composition}: \quad &= m \to \underline{\text{ftn}_M^{\downarrow H}} \fatsemi \left(q \to m \triangleright \left((r \to q \triangleright \text{pu}_M^{\downarrow H} \fatsemi r)\right)^{\uparrow M}\right) \fatsemi \text{ftn}_M \\
\text{expansion}: \quad &= m \to \left(q \to \underline{q \triangleright \text{ftn}_M^{\downarrow H}}\right) \fatsemi \left(q \to m \triangleright \left((r \to \underline{q} \triangleright \text{pu}_M^{\downarrow H} \fatsemi r)\right)^{\uparrow M}\right) \fatsemi \text{ftn}_M \\
\text{composition}: \quad &= m \to \left(q \to m \triangleright \left((r \to \underline{q \triangleright \text{ftn}_M^{\downarrow H} \triangleright \text{pu}_M^{\downarrow H}} \fatsemi r)\right)^{\uparrow M}\right) \fatsemi \text{ftn}_M \quad . \qquad (14.41)
\end{aligned}$$

We can transform the sub-expression $(r \to q \triangleright \text{ftn}_M^{\downarrow H} \triangleright \text{pu}_M^{\downarrow H} \fatsemi r)$ to:

$$\begin{aligned}
\triangleright\text{-notation}: \quad & r \to q \triangleright \underline{\text{ftn}_M^{\downarrow H} \fatsemi \text{pu}_M^{\downarrow H}} \fatsemi r \\
\text{composition law of } H: \quad & = r \to q \triangleright \left(\underline{\text{pu}_M \fatsemi \text{ftn}_M}\right)^{\downarrow H} \fatsemi r \\
\text{left identity law of } M: \quad & = r \to \underline{q \triangleright r} = r \to r(q) \quad . \qquad (14.42)
\end{aligned}$$

Using this simplification, we continue transforming Eq. (14.41):

$$\begin{aligned}
& \text{sw} \fatsemi \text{ftn}_M^{\uparrow R} = m \to \left(q \to m \triangleright \left((r \to \underline{q \triangleright \text{ftn}_M^{\downarrow H} \triangleright \text{pu}_M^{\downarrow H}} \fatsemi r)\right)^{\uparrow M}\right) \fatsemi \text{ftn}_M \\
\text{use Eq. (14.42)}: \quad & = m \to \left(\underline{q} \to m \triangleright (r \to r(q))^{\uparrow M}\right) \underline{\fatsemi} \text{ftn}_M \\
\text{composition}: \quad & = m \to q \to m \triangleright (r \to r(q))^{\uparrow M} \underline{\fatsemi \text{ftn}_M} = m \to q \to m \triangleright \text{flm}_M\,(r \to r(q)) \quad .
\end{aligned}$$

It follows that:

$$(\text{sw} \fatsemi \text{ftn}_M^{\uparrow R})(x)(y) = x \triangleright \text{flm}_M\,(r \to r(y)) \quad . \qquad (14.43)$$

Now we rewrite Eq. (14.40) and get:

$$\begin{aligned}
& \text{ftn}_T = \text{flm}_R(\text{sw} \fatsemi \text{ftn}_M^{\uparrow R}) \\
\text{use Eq. (14.38)}: \quad & = t \to k \to k \triangleright \left(t \triangleright (a \to (\underline{\text{sw} \fatsemi \text{ftn}_M^{\uparrow R}})(a)(k))^{\uparrow R}\right) \\
\text{use Eq. (14.43)}: \quad & = t \to k \to k \triangleright \left(t \triangleright (\underline{a \to a \triangleright} \text{flm}_M\,(r \to r(k)))^{\uparrow R}\right) \\
\text{unexpand function}: \quad & = t \to k \to k \triangleright \left(t \triangleright (\text{flm}_M\,(r \to r(k)))^{\uparrow R}\right) \quad .
\end{aligned}$$

It is sometimes convenient to express the code of ftn_T in terms of flm_R:

$$\text{ftn}_T = \text{flm}_R(\text{sw} \fatsemi \text{ftn}_M^{\uparrow R}) = \text{flm}_R\left(m \to q \to m \triangleright \text{flm}_M\,(r \to r(q))\right) \quad .$$

Deriving flm_T from ftn_T We use the formula for ftn_T just derived above:

$$\begin{aligned}
& \text{flm}_T\,(g) = g^{\uparrow T} \fatsemi \underline{\text{ftn}_T} = \underline{g^{\uparrow M \uparrow R} \fatsemi \text{flm}_R}\left(m \to q \to m \triangleright \text{flm}_M\,(r \to r(q))\right) \\
\text{left naturality law of } \text{flm}_R: \quad & = \text{flm}_R\left(\underline{g^{\uparrow M} \fatsemi (m} \to q \to m \triangleright \text{flm}_M\,(r \to r(q)))\right) \\
\text{compute composition}: \quad & = \text{flm}_R\left(m \to q \to m \triangleright \underline{g^{\uparrow M}} \triangleright \text{flm}_M\,(r \to r(q))\right) \\
\text{naturality law of } \text{flm}_M: \quad & = \text{flm}_R\left(m \to q \to m \triangleright \text{flm}_M\left(\underline{g \fatsemi (r \to r(q))}\right)\right) \\
\text{compute composition}: \quad & = \text{flm}_R\left(m \to q \to m \triangleright \text{flm}_M\left(r \to g(r)(q)\right)\right) \\
\text{obtain Eq. (14.36)}: \quad & = \text{flm}_R\left(m \to q \to m \triangleright \text{flm}_M\,(\gamma_{g,q})\right) \quad .
\end{aligned}$$

This concludes the derivation of the formulas listed in Statement 14.5.3.1.

Statement 14.5.3.2 The function `swap` (short notation $\mathrm{sw}_{R,M}$) defined in Eq. (14.35) satisfies the four laws of Statement 14.3.2.1 and the monadic naturality laws of Statement 14.3.7.1.

Proof We will verify each of the laws in turn.

The inner identity law is $\mathrm{pu}_R^{\uparrow M} \fatsemi \mathrm{sw} = \mathrm{pu}_R$. To verify this law, compute:

$$\begin{aligned}
\text{expect to equal } \mathrm{pu}_R: \quad & \mathrm{pu}_R^{\uparrow M} \fatsemi \underline{\mathrm{sw}} \\
\text{use Eq. (14.35)}: \quad & = (m \to \underline{m \triangleright \mathrm{pu}_R^{\uparrow M}}) \fatsemi (m \to q \to \underline{m} \triangleright (r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r)^{\uparrow M}) \\
\text{function composition}: \quad & = m \to q \to m \triangleright \underline{\mathrm{pu}_R^{\uparrow M} \fatsemi (r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r)^{\uparrow M}} \\
\text{composition under } {}^{\uparrow M}: \quad & = m \to q \to m \triangleright (\mathrm{pu}_R \fatsemi (r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r))^{\uparrow M} \quad .
\end{aligned} \tag{14.44}$$

To proceed, we simplify the expression $\mathrm{pu}_R \fatsemi (r \to ...)$:

$$\begin{aligned}
& \mathrm{pu}_R \fatsemi (r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r) = (m \to m \triangleright \mathrm{pu}_R) \fatsemi (r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \triangleright r) \\
\text{compute composition}: \quad & = m \to q \triangleright \mathrm{pu}_M^{\downarrow H} \triangleright (m \triangleright \mathrm{pu}_R) \quad .
\end{aligned} \tag{14.45}$$

We now have to use the definition of pu_R, which is $\mathrm{pu}_R = x \to y \to x$, or in the pipe notation:

$$y \triangleright (x \triangleright \mathrm{pu}_R) = x \quad . \tag{14.46}$$

With this simplification at hand, we continue from Eq. (14.45) to:

$$\begin{aligned}
& m \to \underline{q \triangleright \mathrm{pu}_M^{\downarrow H} \triangleright (m \triangleright \mathrm{pu}_R)} \\
\text{use Eq. (14.46)}: \quad & = m \to m = \mathrm{id} \quad .
\end{aligned}$$

Therefore, Eq. (14.44) becomes:

$$m \to q \to m \triangleright \underline{(\mathrm{pu}_R \fatsemi (r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r))^{\uparrow M}} = (m \to q \to \underline{m \triangleright \mathrm{id}}) = (m \to q \to m) = \mathrm{pu}_R \quad .$$

This proves the inner identity law.

The outer identity law is $\mathrm{pu}_M \fatsemi \mathrm{sw} = \mathrm{pu}_M^{\uparrow R}$. To verify this law, write:

$$\begin{aligned}
\text{expect to equal } \mathrm{pu}_M^{\uparrow R}: \quad & \mathrm{pu}_M \fatsemi \underline{\mathrm{sw}} \\
\text{use Eq. (14.35)}: \quad & = (m \to \underline{m \triangleright \mathrm{pu}_M}) \fatsemi (m \to q \to \underline{m} \triangleright (r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r)^{\uparrow M}) \\
\text{compute composition}: \quad & = m \to q \to m \triangleright \underline{\mathrm{pu}_M \fatsemi (r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r)^{\uparrow M}} \\
\text{naturality law of } \mathrm{pu}_M: \quad & = m \to q \to m \triangleright (r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r) \fatsemi \mathrm{pu}_M \\
\triangleright\text{-notation}: \quad & = m \to q \to \underline{m \triangleright (r \to} q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi \underline{r} \fatsemi \mathrm{pu}_M) \\
\text{apply function to } m: \quad & = m \to \underline{q \to q} \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi m \fatsemi \mathrm{pu}_M \\
\text{unexpand function}: \quad & = m \to \mathrm{pu}_M^{\downarrow H} \fatsemi m \fatsemi \mathrm{pu}_M \\
\text{definition of } {}^{\uparrow R}: \quad & = \mathrm{pu}_M^{\uparrow R} \quad .
\end{aligned}$$

The inner interchange law is written as:

$$\mathrm{ftn}_R^{\uparrow M} \fatsemi \mathrm{sw} = \mathrm{sw} \fatsemi \mathrm{sw}^{\uparrow R} \fatsemi \mathrm{ftn}_R \quad . \tag{14.47}$$

We will apply both sides of the law to arbitrary $m : M^{R^{R^A}}$ and $q : H^{M^A}$, then rewrite both sides to the same expression.

Below, we will need a simplified formula for ftn_R derived from Eq. (14.38):

$$\begin{aligned}
&\text{ftn}_R = \text{flm}_R(\text{id}) \\
\text{use Eq. (14.38)}: \quad &= t \to q \to q \triangleright (t \triangleright (x \to x(q))^{\uparrow R}) \\
\text{definition of } {}^{\uparrow R}: \quad &= t \to q \to q \triangleright \left(t \triangleright \left(r \to (x \to x(q))^{\downarrow H} ⨾ r ⨾ (x \to x(q))\right)\right) \\
\text{apply to argument}: \quad &= t \to q \to q \triangleright \left((x \to x(q))^{\downarrow H} ⨾ t ⨾ (x \to x(q))\right) \\
\text{use } \triangleright\text{-notation}: \quad &= t \to q \to q \triangleright \left(q \triangleright (x \to q \triangleright x)^{\downarrow H} ⨾ t\right) \quad . \qquad (14.48)
\end{aligned}$$

We first apply the left-hand side of the law (14.47) to m and q:

$$\begin{aligned}
&(\text{ftn}_R^{\uparrow M} ⨾ \text{sw})(m)(q) = q \triangleright \left(m \triangleright \text{ftn}_R^{\uparrow M} ⨾ \text{sw}\right) = q \triangleright \left(m \triangleright \text{ftn}_R^{\uparrow M} \triangleright \text{sw}\right) \\
\text{use Eq. (14.39)}: \quad &= m \triangleright \text{ftn}_R^{\uparrow M} ⨾ (r \to q \triangleright \text{pu}_M^{\downarrow H} ⨾ r)^{\uparrow M} \\
\text{composition law of } M: \quad &= m \triangleright \left(\text{ftn}_R ⨾ (r \to q \triangleright \text{pu}_M^{\downarrow H} ⨾ r)\right)^{\uparrow M} \quad .
\end{aligned}$$

We now need to simplify the sub-expression under $(...)^{\uparrow M}$:

$$\begin{aligned}
&\text{ftn}_R ⨾ (r \to q \triangleright \text{pu}_M^{\downarrow H} \triangleright r) \\
\text{compute composition}: \quad &= r \to q \triangleright \text{pu}_M^{\downarrow H} \triangleright \text{ftn}_R(r) \\
\text{use Eq. (14.48)}: \quad &= r \to q \triangleright \text{pu}_M^{\downarrow H} \triangleright \left(q \triangleright \text{pu}_M^{\downarrow H} \triangleright (x \to q \triangleright \text{pu}_M^{\downarrow H} \triangleright x)^{\downarrow H} ⨾ r\right) \\
\text{composition law of } H: \quad &= r \to q \triangleright \text{pu}_M^{\downarrow H} ⨾ \left(q \triangleright (x \to q \triangleright \text{pu}_M^{\downarrow H} ⨾ x ⨾ \text{pu}_M)^{\downarrow H} ⨾ r\right) \\
\text{definition of } {}^{\uparrow R}: \quad &= r \to q \triangleright \text{pu}_M^{\downarrow H} ⨾ \left(q \triangleright (x \to q \triangleright \text{pu}_M^{\uparrow R}(x))^{\downarrow H} ⨾ r\right) \quad .
\end{aligned}$$

The left-hand side of the law (14.47) then becomes:

$$m \triangleright \left(r \to q \triangleright \text{pu}_M^{\downarrow H} ⨾ \left(q \triangleright (x \to q \triangleright \text{pu}_M^{\uparrow R}(x))^{\downarrow H} ⨾ r\right)\right)^{\uparrow M} \quad .$$

Now apply the right-hand side of the law (14.47) to m and q:

$$\begin{aligned}
&q \triangleright \left(m \triangleright \text{sw} ⨾ \text{sw}^{\uparrow R} ⨾ \text{ftn}_R\right) \\
\text{definition of } {}^{\uparrow R}: \quad &= q \triangleright \left(m \triangleright \text{sw} \triangleright (x \to \text{sw}^{\downarrow H} ⨾ x ⨾ \text{sw}) \triangleright \text{ftn}_R\right) \\
\text{apply to arguments}: \quad &= q \triangleright \left(\text{ftn}_R(\text{sw}^{\downarrow H} ⨾ \text{sw}(m) ⨾ \text{sw})\right) \\
\text{use Eq. (14.48)}: \quad &= q \triangleright \left(q \triangleright (x \to q \triangleright x)^{\downarrow H} ⨾ \text{sw}^{\downarrow H} ⨾ \text{sw}(m) ⨾ \text{sw}\right) \\
\text{composition under } {}^{\downarrow H}: \quad &= q \triangleright \left(q \triangleright \left(\text{sw} ⨾ (x \to q \triangleright x)\right)^{\downarrow H} ⨾ \text{sw}(m) ⨾ \text{sw}\right) \quad . \qquad (14.49)
\end{aligned}$$

To proceed, we simplify the sub-expression $\text{sw}(m) ⨾ \text{sw}$ by computing the function compositions:

$$\begin{aligned}
&\text{sw}(m) ⨾ \text{sw} \\
&= (q_1 \to m \triangleright (r \to q_1 \triangleright \text{pu}_M^{\downarrow H} ⨾ r)^{\uparrow M}) ⨾ (y \to q_2 \to y \triangleright (r \to q_2 \triangleright \text{pu}_M^{\downarrow H} ⨾ r)^{\uparrow M}) \\
&= q_1 \to q_2 \to (m \triangleright (r \to q_1 \triangleright \text{pu}_M^{\downarrow H} ⨾ r)^{\uparrow M}) \triangleright (r \to q_2 \triangleright \text{pu}_M^{\downarrow H} ⨾ r)^{\uparrow M} \\
&= q_1 \to q_2 \to m \triangleright \left((r \to q_1 \triangleright \text{pu}_M^{\downarrow H} ⨾ r) ⨾ (r \to q_2 \triangleright \text{pu}_M^{\downarrow H} ⨾ r)\right)^{\uparrow M} \quad .
\end{aligned}$$

Using this formula, we can write, for any z of a suitable type:

$$\begin{aligned}
&q \triangleright (z \triangleright \text{sw}(m) ⨾ \text{sw}) = m \triangleright \left((r \to z \triangleright \text{pu}_M^{\downarrow H} ⨾ r) ⨾ (r \to q \triangleright \text{pu}_M^{\downarrow H} ⨾ r)\right)^{\uparrow M} \\
\text{compute composition}: \quad &= m \triangleright \left(r \to q \triangleright \text{pu}_M^{\downarrow H} ⨾ (z \triangleright \text{pu}_M^{\downarrow H} ⨾ r)\right)^{\uparrow M} \quad . \qquad (14.50)
\end{aligned}$$

Now we can substitute this into Eq. (14.49):

$$
\begin{aligned}
& q \triangleright \big(q \triangleright \big(\mathrm{sw} \fatsemi (x \to q \triangleright x)\big)^{\downarrow H} \triangleright \underline{\mathrm{sw}(m) \fatsemi \mathrm{sw}}\big) \\
\text{use Eq. (14.50)}: \quad & = m \triangleright \big(r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi (q \triangleright \underline{\big(\mathrm{sw} \fatsemi (x \to q \triangleright x)\big)^{\downarrow H} \triangleright \mathrm{pu}_M^{\downarrow H}} \fatsemi r)\big)^{\uparrow M} \\
\text{composition under } {}^{\downarrow H}: \quad & = m \triangleright \big(r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi (q \triangleright \big(\underline{\mathrm{pu}_M \fatsemi \mathrm{sw}} \fatsemi (x \to q \triangleright x)\big)^{\downarrow H} \fatsemi r)\big)^{\uparrow M} \\
\text{outer identity law of sw}: \quad & = m \triangleright \big(r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi (q \triangleright \big(\underline{\mathrm{pu}_M^{\uparrow R} \fatsemi (x \to q \triangleright x)}\big)^{\downarrow H} \fatsemi r)\big)^{\uparrow M} \\
\text{compute composition}: \quad & = m \triangleright \big(r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi (q \triangleright \big(x \to q \triangleright \mathrm{pu}_M^{\uparrow R}(x)\big)^{\downarrow H} \fatsemi r)\big)^{\uparrow M} \quad .
\end{aligned}
$$

We arrived at the same expression as the left-hand side of the law.
The outer interchange law is written as:

$$
\mathrm{ftn}_M \fatsemi \mathrm{sw} = \mathrm{sw}^{\uparrow M} \fatsemi \mathrm{sw} \fatsemi \mathrm{ftn}_M^{\uparrow R} \quad . \tag{14.51}
$$

We will apply both sides of the law to arbitrary $m^{:M^{M^{R^A}}}$ and $q^{:H^{M^A}}$, and transform both sides to the same expression. We begin with the more complicated right-hand side and use the pipe notation:

$$
\begin{aligned}
& q \triangleright \big(m \triangleright \mathrm{sw}^{\uparrow M} \underline{\fatsemi} \mathrm{sw} \underline{\fatsemi} \mathrm{ftn}_M^{\uparrow R}\big) = q \triangleright \big((m \triangleright \mathrm{sw}^{\uparrow M} \triangleright \mathrm{sw}) \triangleright \underline{\mathrm{ftn}_M^{\uparrow R}}\big) \\
\text{definition of } {}^{\uparrow R}: \quad & = q \triangleright \big(\mathrm{ftn}_M^{\downarrow H} \underline{\fatsemi} (m \triangleright \mathrm{sw}^{\uparrow M} \triangleright \mathrm{sw}) \underline{\fatsemi} \mathrm{ftn}_M\big) \\
\triangleright\text{-notation}: \quad & = \big(\underline{q \triangleright \mathrm{ftn}_M^{\downarrow H}} \triangleright (\underline{m \triangleright \mathrm{sw}^{\uparrow M}} \triangleright \underline{\mathrm{sw}})\big) \triangleright \mathrm{ftn}_M \\
\text{use Eq. (14.39)}: \quad & = \big(m \triangleright \underline{\mathrm{sw}^{\uparrow M}} \triangleright \big(r \to q \triangleright \underline{\mathrm{ftn}_M^{\downarrow H} \triangleright \mathrm{pu}_M^{\downarrow H}} \fatsemi r\big)^{\uparrow M}\big) \triangleright \mathrm{ftn}_M \\
\text{composition under } {}^{\downarrow H} \text{ and } {}^{\uparrow M}: \quad & = m \triangleright \big(\mathrm{sw} \fatsemi \big(r \to q \triangleright (\underline{\mathrm{pu}_M \fatsemi \mathrm{ftn}_M})^{\downarrow H} \fatsemi r\big)\big)^{\uparrow M} \fatsemi \mathrm{ftn}_M \\
\text{left identity law of } M: \quad & = m \triangleright \big(\mathrm{sw} \fatsemi (r \to q \triangleright r)\big)^{\uparrow M} \fatsemi \mathrm{ftn}_M \quad .
\end{aligned} \tag{14.52}
$$

Let us simplify the sub-expression $\mathrm{sw} \fatsemi (r \to q \triangleright r)$ separately:

$$
\begin{aligned}
& \underline{\mathrm{sw}} \fatsemi \big(r \to q \triangleright r\big) = (x \to x \triangleright \mathrm{sw}) \underline{\fatsemi} (r \to q \triangleright r) = x \to \underline{q \triangleright (x \triangleright \mathrm{sw})} \\
\text{use Eq. (14.39)}: \quad & = \underline{x \to x \triangleright} \big(r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r\big)^{\uparrow M} = \big(r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r\big)^{\uparrow M} \quad .
\end{aligned}
$$

Substituting this expression into Eq. (14.52), we get:

$$
\begin{aligned}
& m \triangleright \big(\underline{\mathrm{sw} \fatsemi (r \to q \triangleright r)}\big)^{\uparrow M} \fatsemi \mathrm{ftn}_M = m \triangleright \big(r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r\big)^{\underline{\uparrow M \uparrow M} \fatsemi \mathrm{ftn}_M} \\
\text{naturality law of } \mathrm{ftn}_M: \quad & = m \triangleright \mathrm{ftn}_M \fatsemi \big(r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r\big)^{\uparrow M} \quad .
\end{aligned}
$$

Now write the left-hand side of the law:

$$
\begin{aligned}
& q \triangleright \big(m \triangleright \mathrm{ftn}_M \underline{\fatsemi} \mathrm{sw}\big) = q \triangleright \big(m \triangleright \mathrm{ftn}_M \triangleright \underline{\mathrm{sw}}\big) \\
\text{use Eq. (14.39)}: \quad & = m \triangleright \mathrm{ftn}_M \triangleright \big(r \to q \triangleright \mathrm{pu}_M^{\downarrow H} \fatsemi r\big)^{\uparrow M} \quad .
\end{aligned}
$$

The two sides of the law are now equal.
The monadic naturality laws are written as:

$$
\mathrm{sw}_{R,\mathrm{Id}} = \mathrm{id} \quad , \qquad \mathrm{sw}_{R,M} \fatsemi \phi^{\uparrow R} = \phi \fatsemi \mathrm{sw}_{R,N} \quad , \qquad \mathrm{sw}_{R,M} \fatsemi \theta = \theta^{\uparrow M} \quad ,
$$

where $\phi : M \rightsquigarrow N$ and $\theta : R \rightsquigarrow \mathrm{Id}$ are arbitrary monad morphisms.

To verify the first law, set $M = \text{Id}$ in Eq. (14.39) and get:

$$\begin{aligned}
& q \triangleright (m \triangleright \text{sw}_{R,\text{Id}}) \\
\text{use Eq. (14.39)}: \quad & = m \triangleright \left(r \to q \triangleright \underline{\text{pu}_M^{\downarrow H} \fatsemi r}\right)^{\underline{\uparrow M}} \\
\text{use } M = \text{Id and } \text{pu}_M = \text{id}: \quad & = \underline{m \triangleright (r \to q \triangleright r)} \\
\text{apply to argument } m: \quad & = q \triangleright \underline{m} = q \triangleright (m \triangleright \text{id}) \quad .
\end{aligned}$$

So, $\text{sw}_{R,\text{Id}} = \text{id}$ when applied to arbitrary argument values m and q.

To verify the second law, apply both sides to arbitrary m and q. The left-hand side:

$$\begin{aligned}
& q \triangleright \left(m \triangleright \text{sw}_{R,M} \underline{\fatsemi} \phi^{\uparrow R}\right) = q \triangleright \left(m \triangleright \text{sw}_{R,M} \triangleright \underline{\phi^{\uparrow R}}\right) \\
\text{definition of } {}^{\uparrow R}: \quad & = q \triangleright \left(\phi^{\downarrow H} \fatsemi (m \triangleright \text{sw}_{R,M}) \fatsemi \phi\right) \\
\triangleright\text{-notation}: \quad & = \underline{(q \triangleright \phi^{\downarrow H})} \triangleright (m \triangleright \underline{\text{sw}_{R,M}}) \triangleright \phi \\
\text{use Eq. (14.39)}: \quad & = m \triangleright \left(r \to q \triangleright \underline{\phi^{\downarrow H} \triangleright \text{pu}_M^{\downarrow H}} \fatsemi r\right)^{\uparrow M} \triangleright \phi \\
\text{composition law of } H: \quad & = m \triangleright \left(r \to q \triangleright \underline{(\text{pu}_M \fatsemi \phi)}^{\downarrow H} \fatsemi r\right)^{\uparrow M} \fatsemi \phi \\
\text{identity law of } \phi: \quad & = m \triangleright \left(r \to q \triangleright \text{pu}_N^{\downarrow H} \fatsemi r\right)^{\underline{\uparrow M}} \underline{\fatsemi \phi} \\
\text{naturality law of } \phi: \quad & = m \triangleright \phi \fatsemi \left(r \to q \triangleright \text{pu}_N^{\downarrow H} \fatsemi r\right)^{\uparrow N} \quad .
\end{aligned}$$

The right-hand side, when applied to m and q, gives the same expression:

$$\begin{aligned}
& q \triangleright (m \triangleright \phi \fatsemi \text{sw}_{R,N}) = q \triangleright (m \triangleright \phi \triangleright \underline{\text{sw}_{R,N}}) \\
\text{use Eq. (14.39)}: \quad & = m \triangleright \phi \triangleright \left(r \to q \triangleright \text{pu}_N^{\downarrow H} \fatsemi r\right)^{\uparrow N} \quad .
\end{aligned}$$

To argue that the third law holds,[17] apply the left-hand side to m and q:

$$\begin{aligned}
& q \triangleright (m \triangleright \text{sw}_{R,M} \fatsemi \theta) = q \triangleright (m \triangleright \text{sw}_{R,M} \triangleright \theta) \\
& = q \triangleright \left((q_1 \to m \triangleright \left(r \to q_1 \triangleright \text{pu}_M^{\downarrow H} \fatsemi r\right)^{\uparrow M}) \triangleright \theta\right) \quad .
\end{aligned} \tag{14.53}$$

This expression cannot be simplified any further; and neither can the right-hand side, $q \triangleright (m \triangleright \theta^{\uparrow M})$. We need more detailed information about the function θ. The type of θ is:

$$\theta : \forall A. (H^A \to A) \to A \quad .$$

To implement a function of this type, we need to write code that takes an argument of type $H^A \to A$ and returns a value of type A. Since the type A is arbitrary, the code of θ cannot store a fixed value of type A to use as the return value. It appears that the only possibility of implementing a function θ is by substituting a value of type H^A into the given argument of type $H^A \to A$. So,[18] we need to produce a value of type H^A for an arbitrary type A, that is, a value of type $\forall A. H^A$. Using the contravariant Yoneda identity, we can simplify this type expression to the type $H^{\mathbb{1}}$:

$$\begin{aligned}
& \forall A. H^A \cong \forall A. \underline{\mathbb{1}} \to H^A \\
\text{use identity } (A \to \mathbb{1}) \cong \mathbb{1}: \quad & \cong \forall A. (A \to \mathbb{1}) \to H^A \\
\text{contravariant Yoneda identity}: \quad & \cong H^{\mathbb{1}} \quad .
\end{aligned}$$

[17] The present proof needs an additional assumption about θ, which will be indicated below.

[18] We did not prove that the type $\forall A. (H^A \to A) \to A$ is *equivalent* to $\forall A. H^A$. Later steps in the proof assume that θ has the form shown in Eq. (14.54). Up to that assumption, the proof is rigorous.

So, we can construct a θ if we take a value h_1 of type $H^{\mathbb{1}}$ and compute $h : H^A$ as:

$$h^{:H^A} = h_1^{:H^1} \triangleright (a^{:A} \to 1)^{\downarrow H} \quad .$$

For any given value $h_1 : H^{\mathbb{1}}$, we can implement a runner θ as:

$$\left(r^{:H^A \to A}\right) \triangleright \theta \stackrel{\text{def}}{=} h_1 \triangleright (_ \to 1)^{\downarrow H} \triangleright r \quad . \tag{14.54}$$

Let us check whether this θ is a monad morphism $R \rightsquigarrow \text{Id}$. We need to verify the laws:

$$\text{pu}_R \fatsemi \theta = \text{id} \quad , \qquad \text{ftn}_R \fatsemi \theta = \theta^{\uparrow R} \fatsemi \theta = \theta \fatsemi \theta \quad .$$

To verify the identity law, apply its both sides to an arbitrary $x^{:A}$ and simplify:

$$\begin{aligned}
\text{expect to equal } x: \quad & x \triangleright \text{pu}_R \fatsemi \theta = (x \triangleright \text{pu}_R) \triangleright \theta \\
\text{definition of } r \triangleright \theta: \quad & = h_1 \triangleright (_ \to 1)^{\downarrow H} \triangleright (x \triangleright \text{pu}_R) \\
\text{definition of } x \triangleright \text{pu}_R: \quad & = \left(h_1 \triangleright (_ \to 1)^{\downarrow H}\right) \triangleright (_ \to x) \\
\text{compute composition}: \quad & = x \quad .
\end{aligned}$$

The composition law, applied to an arbitrary $r : R^{R^A}$, simplifies to:

$$\begin{aligned}
& r \triangleright \text{ftn}_R \fatsemi \theta = r \triangleright \text{ftn}_R \triangleright \theta \\
\text{definition of } \text{ftn}_R: \quad & = r \triangleright \left(t \to q \to q \triangleright \left(q \triangleright (x \to q \triangleright x)^{\downarrow H} \fatsemi t\right)\right) \triangleright \theta \\
\text{apply to } r: \quad & = \left(q \to q \triangleright \left(q \triangleright (x \to q \triangleright x)^{\downarrow H} \fatsemi r\right)\right) \triangleright \theta \\
\text{definition (14.54) of } \theta: \quad & = h_1 \triangleright (_ \to 1)^{\downarrow H} \triangleright \left(q \to q \triangleright \left(q \triangleright (x \to q \triangleright x)^{\downarrow H} \fatsemi r\right)\right) \\
\text{apply to } h_1 \triangleright (_ \to 1)^{\downarrow H}: \quad & = h_1 \triangleright (_ \to 1)^{\downarrow H} \triangleright \left(h_1 \triangleright (_ \to 1)^{\downarrow H} \fatsemi (x \to \ldots)^{\downarrow H} \fatsemi r\right) \\
\text{composition under } {}^{\downarrow H}: \quad & = h_1 \triangleright (_ \to 1)^{\downarrow H} \triangleright \left(h_1 \triangleright (_ \to 1)^{\downarrow H} \triangleright r\right) \\
\text{definition (14.54) of } \theta: \quad & = r \triangleright \theta \triangleright \theta = r \triangleright \theta \fatsemi \theta \quad .
\end{aligned}$$

This verifies the composition law. So, θ is indeed a monad morphism.

Using the code of θ defined in Eq. (14.54), we can now verify the monadic naturality law of $\text{sw}_{R,M}$ (with respect to the runners θ of that form). The left-hand side of the law is given by Eq. (14.53):

$$\begin{aligned}
& q \triangleright \left((q_1 \to m \triangleright \left(r \to q_1 \triangleright \text{pu}_M^{\downarrow H} \fatsemi r\right)^{\uparrow M}) \triangleright \theta\right) \\
\text{definition of } \theta: \quad & = q \triangleright \left(h_1 \triangleright (_ \to 1)^{\downarrow H} \triangleright (q_1 \to m \triangleright \left(r \to q_1 \triangleright \text{pu}_M^{\downarrow H} \fatsemi r\right)^{\uparrow M})\right) \\
\text{apply to argument}: \quad & = q \triangleright \left(m \triangleright \left(r \to h_1 \triangleright (_ \to 1)^{\downarrow H} \triangleright \text{pu}_M^{\downarrow H} \fatsemi r\right)^{\uparrow M}\right) \\
\text{composition under } {}^{\downarrow H}: \quad & = q \triangleright \left(m \triangleright \left(r \to h_1 \triangleright (\text{pu}_M \fatsemi (_ \to 1))^{\downarrow H} \fatsemi r\right)^{\uparrow M}\right) \\
\text{compute composition}: \quad & = q \triangleright \left(m \triangleright \left(r \to h_1 \triangleright (_ \to 1)^{\downarrow H} \triangleright r\right)^{\uparrow M}\right) \quad .
\end{aligned}$$

The right-hand side is:

$$\begin{aligned}
& q \triangleright (m \triangleright \theta^{\uparrow M}) \\
\text{expand function}: \quad & = q \triangleright \left(m \triangleright \left(r \to r \triangleright \theta\right)^{\uparrow M}\right) \\
\text{definition (14.54) of } \theta: \quad & = q \triangleright \left(m \triangleright \left(r \to h_1 \triangleright (_ \to 1)^{\downarrow H} \triangleright r\right)^{\uparrow M}\right) \quad .
\end{aligned}$$

This expression is now the same as the left-hand side.

This concludes the proof of the laws for the "choice" monad's transformer.

14.6 Transformers for some monad constructions

14.6.1 Transformer for products of monads. Proofs

The product of any two monads is again a monad (Statement 10.2.8.2). If the two monads have known transformers, the product of the two transformers is the transformer for the product monad.

For instance, the product of `Either` and `Reader` monads, $L^A \stackrel{\text{def}}{=} (E+A)\times(R\to A)$, has the transformer:

$$T_L^{M,A} \stackrel{\text{def}}{=} M^{E+A}\times(R\to M^A) \quad .$$

Note that this monad transformer is neither composed-outside nor composed-inside.

Statement 14.6.1.1 The product of two lawful monad transformers satisfies all transformer laws.

Proof Given two monads G and H whose transformers T_G and T_H are known, we need to show that a lawful transformer for the monad $L^\bullet \stackrel{\text{def}}{=} G^\bullet\times H^\bullet$ is given by:

$$T_L^{M,A} \stackrel{\text{def}}{=} T_G^{M,A}\times T_H^{M,A} \quad .$$

We will assume that all monad transformer laws already hold for T_G and T_H.

Monad laws Since T_G^M and T_H^M are (by assumption) lawful monads, T_L^M is a lawful monad by the product construction (Statement 10.2.8.2).

Identity laws With $M=\text{Id}$, we have $T_L^{\text{Id}} = T_G^{\text{Id}}\times T_H^{\text{Id}}$. The equivalence $T_G^{\text{Id}}\times T_H^{\text{Id}}\cong G\times H$ is found via the pair product of the monad morphisms $T_G^{\text{Id}}\cong G$ and $T_H^{\text{Id}}\cong H$. The pair product of two monad morphisms is again a monad morphism (Statement 14.9.1.2).

Lifting laws We define the foreign lift function (`flift`) via the pair product:

$$\text{flift}_L^{:M\rightsquigarrow T_G^M\times T_H^M} \stackrel{\text{def}}{=} \Delta\,⨾\,(\text{flift}_G\boxtimes\text{flift}_H) \quad .$$

To show that flift_L is a monad morphism, we use the properties that $\Delta : M\rightsquigarrow M\times M$ is a monad morphism (Statement 14.9.1.1), that the pair product preserves monad morphisms (Statement 14.9.1.2), and that function composition also preserves monad morphisms (Statement 10.3.4.5).

Runner laws We need to define the foreign runner (`frun`):

$$\text{frun}_L(\phi^{:M\rightsquigarrow N}) : T_G^{M,A}\times T_H^{M,A}\rightsquigarrow T_G^{N,A}\times T_H^{N,A} \quad , \qquad \text{frun}_L(\phi)\stackrel{\text{def}}{=}\text{frun}_G(\phi)\boxtimes\text{frun}_H(\phi) \quad .$$

The function returned by $\text{frun}_L(\phi)$ is a monad morphism by Statement 14.9.1.2.

The functor laws hold due to the properties of the pair product:

$$\begin{aligned}
&\text{expect to equal id}: && \text{frun}_L(\text{id}) = \text{frun}_G(\text{id})\boxtimes\text{frun}_H(\text{id}) = \text{id}\boxtimes\text{id} = \text{id} \quad ,\\
&\text{expect to equal frun}_L(\phi\,⨾\,\chi): && \text{frun}_L(\phi)\,⨾\,\text{frun}_L(\chi) = \big(\underline{\text{frun}_G(\phi)\,⨾\,\text{frun}_G(\chi)}\big)\boxtimes\big(\underline{\text{frun}_H(\phi)\,⨾\,\text{frun}_H(\chi)}\big)\\
&\text{functor laws of frun}_G,\ \text{frun}_H: && = \text{frun}_G(\phi\,⨾\,\chi)\boxtimes\text{frun}_H(\phi\,⨾\,\chi) = \text{frun}_L(\phi\,⨾\,\chi) \quad .
\end{aligned}$$

Base runner laws We may define a base runner in two different ways, by dropping either the first or the second part of the product. Both base runners are lawful. Consider the first one:

$$\text{brun}_L : T_G^M\times T_H^M\rightsquigarrow M \quad , \qquad \text{brun}_L\stackrel{\text{def}}{=}\pi_1\,⨾\,\text{brun}_G \quad .$$

It is a monad morphism because it is a composition of a projection (Statement 14.9.1.3) and a monad morphism brun_G. Function composition preserves monad morphisms (Statement 10.3.4.5).

The nondegeneracy law holds because it holds for brun_G:

$$\text{flift}_L\,⨾\,\text{brun}_L = \Delta\,⨾\,\underline{(\text{flift}_G\boxtimes\text{flift}_H)\,⨾\,\pi_1}\,⨾\,\text{brun}_G = \underline{\Delta\,⨾\,\pi_1}\,⨾\,\underline{\text{flift}_G\,⨾\,\text{brun}_G} = \text{id} \quad .$$

Analogous proofs work for the other definition, $\text{brun}_L\stackrel{\text{def}}{=}\pi_2\,⨾\,\text{brun}_H$.

A deficiency of these base runners is that the effects of one of the monads (G or H) are being ignored, which loses information. If we would like to run the effects of both monads G and H, we can instead implement a base runner into the monad $M \times M$. This corresponds to running the monad $G \times H$ into the "double-valued" monad $D^A \stackrel{\text{def}}{=} A \times A$, since $T_D^M = M \times M$. Define the base runner as:

$$\text{brun}_L : T_G^M \times T_H^M \rightsquigarrow M \times M \quad , \qquad \text{brun}_L \stackrel{\text{def}}{=} \text{brun}_G \boxtimes \text{brun}_H \quad .$$

This is a monad morphism due to Statement 14.9.1.2. The nondegeneracy law has the form:

$$\text{flift}_L \fatsemi \text{brun}_L = \Delta^{:M \rightsquigarrow M \times M} \quad ,$$

since the target monad is now $M \times M$. To verify the nondegeneracy law:

$$\begin{aligned}
&\text{flift}_L \fatsemi \text{brun}_L = \Delta \fatsemi (\text{flift}_G \boxtimes \text{flift}_H) \fatsemi (\text{brun}_G \boxtimes \text{brun}_H) \\
&= \Delta \fatsemi (\underline{\text{flift}_G \fatsemi \text{brun}_G}) \boxtimes (\underline{\text{flift}_H \fatsemi \text{brun}_H}) = \Delta \fatsemi (\text{id} \boxtimes \text{id}) = \Delta \quad .
\end{aligned}$$

Monadic naturality laws To verify the monadic naturality law of flift_L:

$$\begin{aligned}
&\text{flift}_L \fatsemi \text{frun}_L(\phi) = \Delta \fatsemi (\text{flift}_G \boxtimes \text{flift}_H) \fatsemi (\text{frun}_G(\phi) \boxtimes \text{frun}_H(\phi)) \\
&= \Delta \fatsemi (\underline{\text{flift}_G \fatsemi \text{frun}_G(\phi)}) \boxtimes (\underline{\text{flift}_H \fatsemi \text{frun}_H(\phi)}) = \Delta \fatsemi (\underline{\phi} \fatsemi \text{flift}_G) \boxtimes (\underline{\phi} \fatsemi \text{flift}_H) \\
&= \underline{\Delta \fatsemi (\phi \boxtimes \phi)} \fatsemi (\text{flift}_G \boxtimes \text{flift}_H) = \phi \fatsemi \underline{\Delta \fatsemi (\text{flift}_G \boxtimes \text{flift}_H)} = \phi \fatsemi \text{flift}_L \quad .
\end{aligned}$$

To verify the monadic naturality law of an information-losing definition of brun_L:

$$\begin{aligned}
&\text{frun}_L(\phi) \fatsemi \text{brun}_L = \underline{(\text{frun}_G(\phi) \boxtimes \text{frun}_H(\phi)) \fatsemi \pi_1} \fatsemi \text{brun}_G = \pi_1 \fatsemi \text{frun}_G(\phi) \fatsemi \text{brun}_G \quad , \\
&\text{brun}_L \fatsemi \phi = \pi_1 \fatsemi \underline{\text{brun}_G \fatsemi \phi} = \pi_1 \fatsemi \text{frun}_G(\phi) \fatsemi \text{brun}_G \quad .
\end{aligned}$$

The runner $\text{brun}_L : T_L^M \rightsquigarrow M \times M$ also obeys a monadic naturality law involving $\phi \boxtimes \phi$ instead of ϕ:

$$\begin{aligned}
&\text{frun}_L(\phi) \fatsemi \text{brun}_L = (\text{frun}_G(\phi) \boxtimes \text{frun}_H(\phi)) \fatsemi (\text{brun}_G \boxtimes \text{brun}_H) \\
&= (\underline{\text{frun}_G(\phi) \fatsemi \text{brun}_G}) \boxtimes (\underline{\text{frun}_H(\phi) \fatsemi \text{brun}_H}) = (\text{brun}_G \fatsemi \phi) \boxtimes (\text{brun}_H \fatsemi \phi) \\
&= (\text{brun}_G \boxtimes \text{brun}_H) \fatsemi (\phi \boxtimes \phi) \quad .
\end{aligned}$$

14.6.2 Transformer for free pointed monads. Proofs

For an arbitrary monad K, a **free pointed monad on** K is defined as the monad $L^A \stackrel{\text{def}}{=} A + K^A$ (see Statement 10.2.8.4). An example of a free pointed monad is the `Eval` monad (Section 10.1.8) having the type $A + (\mathbb{1} \to A)$, which is of the form $A + K^A$ with $K^A \stackrel{\text{def}}{=} \mathbb{1} \to A$ being a `Reader` monad.

Assuming that a given monad K has a lawful transformer, we will now show that the free pointed monad on K also has a monad transformer satisfying all laws.

Statement 14.6.2.1 The monad L's transformer is defined by the type formula:

$$T_L^{M,A} \stackrel{\text{def}}{=} M^{A + T_K^{M,A}} \quad ,$$

where T_K is the monad K's transformer, which is assumed to be known.

Proof We will define the necessary methods for T_L when we need them in the course of the proof.

To verify the monad transformer laws for T_L, we begin by observing that $A + T_K^{M,A}$ is a free pointed monad on T_K^M. Let us denote for brevity:

$$N^A \stackrel{\text{def}}{=} T_K^{M,A} \quad , \qquad P^A \stackrel{\text{def}}{=} A + T_K^{M,A} = A + N^A \quad , \qquad T_L^{M,A} \stackrel{\text{def}}{=} M^{P^A} = (M \circ P)^A \quad .$$

We know from Statement 10.2.8.4 that P's monad methods are defined by:

$$\text{pu}_P \stackrel{\text{def}}{=} a^{:A} \to a + \mathbb{0}^{:N^A} \quad , \quad \text{ftn}_P \stackrel{\text{def}}{=} \left\| \begin{array}{c|cc} & A & N^A \\ \hline A & \text{id} & \mathbb{0} \\ N^A & \mathbb{0} & \text{id} \\ N^{P^A} & \mathbb{0} & \gamma^{\uparrow N} ⨾ \text{ftn}_N \end{array} \right\| \quad , \quad \text{where} \quad \gamma^{:P^A \to N^A} \stackrel{\text{def}}{=} \left\| \begin{array}{c|c} & N^A \\ \hline A & \text{pu}_N \\ N^A & \text{id} \end{array} \right\| \quad .$$

The transformer T_L is a composition of the foreign monad M outside P. To show that T_L is a monad, we may use Statement 14.3.2.1, which requires a suitable `swap` function (denoted by "sw"):

$$\text{sw} : P^{M^A} \to M^{P^A} \quad , \quad \text{sw} \stackrel{\text{def}}{=} \left\| \begin{array}{c|c} & M^{A+N^A} \\ \hline M^A & \text{pu}_P^{\uparrow M} \\ N^{M^A} & (t \to \mathbb{0}^{:A} + \text{merge}\,(t)) ⨾ \text{pu}_M \end{array} \right\| \quad . \tag{14.55}$$

Here, `merge` is a helper function that combines all effects contained within N^{M^A} into one N-effect represented by a value of type N^A:

$$\text{merge} : N^{M^A} \to N^A \quad , \quad \text{merge} \stackrel{\text{def}}{=} \text{flift}_K^{\uparrow N} ⨾ \text{ftn}_N \quad .$$

We will now verify all monad transformer laws for T_L.

Monad laws The monad methods of T_L are defined as in Statement 14.3.2.1:

$$\text{pu}_T \stackrel{\text{def}}{=} \text{pu}_P ⨾ \text{pu}_M = (a^{:A} \to a + \mathbb{0}^{:N^A}) ⨾ \text{pu}_M \quad , \quad \text{ftn}_T \stackrel{\text{def}}{=} \text{sw}^{\uparrow M} ⨾ \text{ftn}_M ⨾ \text{ftn}_P^{\uparrow M} \quad .$$

The composition $T_L^M \stackrel{\text{def}}{=} M \circ P$ is a monad as long as the `swap` function obeys the naturality law and the four laws given in Statement 14.3.2.1, if we replace L and M by M and P. Let us verify those laws.

The `swap` function is a natural transformation because it is a combination of various natural transformations such as pu_M and `merge`.

The outer identity law is $\text{pu}_M^{\uparrow P} ⨾ \text{sw} = \text{pu}_M$. We rewrite the left-hand side, expecting to obtain pu_M:

$$\text{pu}_M^{\uparrow P} ⨾ \text{sw} = \left\| \begin{array}{c|cc} & M^A & N^{M^A} \\ \hline A & \text{pu}_M & \mathbb{0} \\ N^A & \mathbb{0} & \text{pu}_M^{\uparrow N} \end{array} \right\| ⨾ \left\| \begin{array}{c|c} & M^{A+N^A} \\ \hline M^A & (a \to a + \mathbb{0}^{:N^A})^{\uparrow M} \\ N^{M^A} & (t \to \mathbb{0}^{:A} + \text{merge}\,(t)) ⨾ \text{pu}_M \end{array} \right\|$$

$$= \left\| \begin{array}{c|c} & M^{A+N^A} \\ \hline A & \text{pu}_M ⨾ (a \to a + \mathbb{0}^{:N^A})^{\uparrow M} \\ N^A & \text{pu}_M^{\uparrow N} ⨾ (t \to \mathbb{0}^{:A} + \text{merge}\,(t)) ⨾ \text{pu}_M \end{array} \right\| = \left\| \begin{array}{c|c} & M^{A+N^A} \\ \hline A & (a \to a + \mathbb{0}^{:N^A}) ⨾ \text{pu}_M \\ N^A & (n \to \mathbb{0}^{:A} + \text{merge}\,(n ▷ \text{pu}_M^{\uparrow N})) ⨾ \text{pu}_M \end{array} \right\|$$

$$= \left\| \begin{array}{c|c} & A + N^A \\ \hline A & a \to a + \mathbb{0}^{:N^A} \\ N^A & n \to \mathbb{0}^{:A} + n ▷ \text{pu}_M^{\uparrow N} ⨾ \text{merge} \end{array} \right\| ⨾ \text{pu}_M = \left\| \begin{array}{c|cc} & A & N^A \\ \hline A & \text{id} & \mathbb{0} \\ N^A & \mathbb{0} & \text{pu}_M^{\uparrow N} ⨾ \text{merge} \end{array} \right\| ⨾ \text{pu}_M \quad .$$

It remains to show that the matrix element $\text{pu}_M^{\uparrow N} ⨾ \text{merge}$ is equal to the identity function:

$$\begin{aligned} \text{expect to equal id :} \quad & \text{pu}_M^{\uparrow N} ⨾ \text{merge} = \underline{\text{pu}_M^{\uparrow N} ⨾ \text{flift}_K^{\uparrow N}} ⨾ \text{ftn}_N \\ \text{monad morphism law of flift}_K : \quad & = \text{pu}_N^{\uparrow N} ⨾ \text{ftn}_N = \text{id} \quad . \end{aligned}$$

The inner identity law is $\text{pu}_P \fatsemi \text{sw} = \text{pu}_P^{\uparrow M}$. Begin with the left-hand side:

$$\text{pu}_P \fatsemi \text{sw} = (m^{:M^A} \to m + \mathbb{0}^{:N^{M^A}}) \fatsemi \text{sw}$$

$$= \left| \begin{array}{c|cc} & M^A & N^{M^A} \\ \hline M^A & \text{id} & \mathbb{0} \end{array} \right| \fatsemi \left| \begin{array}{c|c} & M^{A+N^A} \\ \hline M^A & \text{pu}_P^{\uparrow M} \\ N^{M^A} & (n \to \mathbb{0}^{:A} + \text{merge}\,(n)) \fatsemi \text{pu}_M \end{array} \right| = \text{pu}_P^{\uparrow M} \quad .$$

The outer interchange law is:

$$\text{ftn}_M^{\uparrow P} \fatsemi \text{sw} = \text{sw} \fatsemi \text{sw}^{\uparrow M} \fatsemi \text{ftn}_M \quad .$$

Begin with the right-hand side since it is more complicated, and use the identity law we just proved:

$$\text{sw} \fatsemi \text{sw}^{\uparrow M} \fatsemi \text{ftn}_M = \left| \begin{array}{c|c} & M^{A+N^A} \\ \hline M^{M^A} & \underline{\text{pu}_P^{\uparrow M} \fatsemi \text{sw}^{\uparrow M}} \fatsemi \text{ftn}_M \\ N^{M^{M^A}} & (n \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \underline{\text{pu}_M \fatsemi \text{sw}^{\uparrow M}} \fatsemi \text{ftn}_M \end{array} \right|$$

$$= \left| \begin{array}{c} \underline{\text{pu}_P^{\uparrow M \uparrow M} \fatsemi \text{ftn}_M} \\ (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{sw} \fatsemi \underline{\text{pu}_M \fatsemi \text{ftn}_M} \end{array} \right| = \left| \begin{array}{c} \text{ftn}_M \fatsemi \text{pu}_P^{\uparrow M} \\ (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{sw} \end{array} \right| \quad .$$

The left-hand side of the law is written as:

$$\text{ftn}_M^{\uparrow P} \fatsemi \text{sw} = \left| \begin{array}{c|cc} & M^A & N^{M^A} \\ \hline M^{M^A} & \text{ftn}_M & \mathbb{0} \\ N^{M^{M^A}} & \mathbb{0} & \text{ftn}_M^{\uparrow N} \end{array} \right| \fatsemi \left| \begin{array}{c|c} & M^{A+N^A} \\ \hline M^A & \text{pu}_P^{\uparrow M} \\ N^{M^A} & (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{pu}_M \end{array} \right|$$

$$= \left| \begin{array}{c|c} & M^{A+N^A} \\ \hline M^{M^A} & \text{ftn}_M \fatsemi \text{pu}_P^{\uparrow M} \\ N^{M^{M^A}} & \text{ftn}_M^{\uparrow N} \fatsemi (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{pu}_M \end{array} \right| \quad .$$

The first rows of the matrices for the two sides of the law are now equal (to $\text{ftn}_M \fatsemi \text{pu}_P^{\uparrow M}$). It remains to show that the second rows of the matrices are also equal:

$$(t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{sw} \stackrel{?}{=} \text{ftn}_M^{\uparrow N} \fatsemi (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{pu}_M \quad ,$$

apply both sides to t : $\quad (\mathbb{0} + t \triangleright \text{merge}) \triangleright \underline{\text{sw}} \stackrel{?}{=} (\mathbb{0} + t \triangleright \text{ftn}_M^{\uparrow N} \triangleright \text{merge}) \triangleright \text{pu}_M \quad ,$

apply sw : $\quad (\mathbb{0} + t \triangleright \text{merge} \triangleright \text{merge}) \triangleright \text{pu}_M \stackrel{?}{=} (\mathbb{0} + t \triangleright \text{ftn}_M^{\uparrow N} \triangleright \text{merge}) \triangleright \text{pu}_M \quad .$

We will prove the last equality if we show that the function `merge` has the following property:

$$\text{merge} \fatsemi \text{merge} = \text{ftn}_M^{\uparrow N} \fatsemi \text{merge} \quad , \quad \text{or :} \quad \text{flift}_K^{\uparrow N} \fatsemi \text{ftn}_N \fatsemi \text{flift}_K^{\uparrow N} \fatsemi \text{ftn}_N = \text{ftn}_M^{\uparrow N} \fatsemi \text{flift}_K^{\uparrow N} \fatsemi \text{ftn}_N \quad .$$

$$\begin{array}{ccc} N^{M^{M^A}} & \xrightarrow{\text{merge}^{M^A}} & N^{M^A} \\ {\scriptstyle \text{ftn}_M^{\uparrow N}} \downarrow & & \downarrow {\scriptstyle \text{merge}^A} \\ N^{M^A} & \xrightarrow{\text{merge}^A} & N^A \end{array}$$

We know nothing about the transformer T_K's `flift` function except for its laws. The law connecting `flift` and `flatten` is the lifting law:

$$\text{flift}_K \fatsemi \text{flift}_K^{\uparrow N} \fatsemi \text{ftn}_N = \text{ftn}_M \fatsemi \text{flift}_K \quad . \tag{14.56}$$

To make use of that law, we use other laws to transform the expression merge ⨾ merge, so that the two `flift` functions are composed together and are followed by ftn_N:

$$
\begin{aligned}
\text{expect to equal } \text{ftn}_M^{\uparrow N} \fatsemi \text{merge}: \quad & \text{merge} \fatsemi \text{merge} = \text{flift}_K^{\uparrow N} \fatsemi \underline{\text{ftn}_N \fatsemi \text{flift}_K^{\uparrow N}} \fatsemi \text{ftn}_N \\
\text{naturality of } \text{ftn}_N: \quad & = \text{flift}_K^{\uparrow N} \fatsemi \text{flift}_K^{\uparrow N \uparrow N} \fatsemi \underline{\text{ftn}_N \fatsemi \text{ftn}_N} \\
\text{associativity law of } \text{ftn}_N: \quad & = \underline{\text{flift}_K^{\uparrow N} \fatsemi \text{flift}_K^{\uparrow N \uparrow N} \fatsemi \text{ftn}_N^{\uparrow N}} \fatsemi \text{ftn}_N = (\underline{\text{flift}_K \fatsemi \text{flift}_K^{\uparrow N} \fatsemi \text{ftn}_N})^{\uparrow N} \fatsemi \text{ftn}_N \\
\text{lifting law (14.56)}: \quad & = \underline{(\text{ftn}_M \fatsemi \text{flift}_K)^{\uparrow N}} \fatsemi \text{ftn}_N = \text{ftn}_M^{\uparrow N} \fatsemi \underline{\text{flift}_K^{\uparrow N} \fatsemi \text{ftn}_N} = \text{ftn}_M^{\uparrow N} \fatsemi \text{merge} \quad .
\end{aligned}
$$

This completes the proof of the outer interchange law of `swap`.

The inner interchange law is:

$$
\text{ftn}_P \fatsemi \text{sw} = \text{sw}^{\uparrow P} \fatsemi \text{sw} \fatsemi \text{ftn}_P^{\uparrow M} \quad .
$$

The left-hand side of that law is rewritten as:

$$
\text{ftn}_P \fatsemi \text{sw} = \begin{array}{|c||cc|}
\hline
 & M^A & N^{M^A} \\
\hline
M^A & \text{id} & \mathbb{0} \\
N^{M^A} & \mathbb{0} & \text{id} \\
N^{P^{M^A}} & \mathbb{0} & \gamma^{\uparrow N} \fatsemi \text{ftn}_N \\
\hline
\end{array} \fatsemi \begin{array}{|c||c|}
\hline
 & M^{P^A} \\
\hline
M^A & \text{pu}_P^{\uparrow M} \\
N^{M^A} & (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{pu}_M \\
\hline
\end{array}
$$

$$
= \begin{array}{|c||c|}
\hline
 & M^{P^A} \\
\hline
M^A & \text{pu}_P^{\uparrow M} \\
N^{M^A} & (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{pu}_M \\
N^{P^{M^A}} & \gamma^{\uparrow N} \fatsemi \text{ftn}_N \fatsemi (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{pu}_M \\
\hline
\end{array} \quad .
$$

The right-hand side is rewritten using the identity law of ftn_P and the naturality law of pu_P:

$$
\text{sw}^{\uparrow P} \fatsemi \text{sw} \fatsemi \text{ftn}_P^{\uparrow M}
$$

$$
= \begin{array}{|c||cc|}
\hline
 & M^{P^A} & N^{M^{P^A}} \\
\hline
M^A & \text{pu}_P^{\uparrow M} & \mathbb{0} \\
N^{M^A} & (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{pu}_M & \mathbb{0} \\
N^{P^{M^A}} & \mathbb{0} & \text{sw}^{\uparrow N} \\
\hline
\end{array} \fatsemi \begin{array}{|c||c|}
\hline
 & M^{P^A} \\
\hline
M^{P^A} & \underline{\text{pu}_P^{\uparrow M} \fatsemi \text{ftn}_P^{\uparrow M}} \\
N^{M^{P^A}} & (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \underline{\text{pu}_M \fatsemi \text{ftn}_P^{\uparrow M}} \\
\hline
\end{array}
$$

$$
= \begin{array}{|c||c|}
\hline
 & M^{P^A} \\
\hline
M^A & \text{pu}_P^{\uparrow M} \\
N^{M^A} & (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{pu}_M \\
N^{P^{M^A}} & \text{sw}^{\uparrow N} \fatsemi (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{ftn}_P \fatsemi \text{pu}_M \\
\hline
\end{array} \quad .
$$

The difference between the two sides is only in the third rows of the matrices:

$$
\gamma^{\uparrow N} \fatsemi \text{ftn}_N \fatsemi (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{pu}_M \stackrel{?}{=} \text{sw}^{\uparrow N} \fatsemi (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{ftn}_P \fatsemi \text{pu}_M \quad .
$$

We can omit the composition with pu_M if we can prove a stronger condition:

$$
\gamma^{\uparrow N} \fatsemi \text{ftn}_N \fatsemi (t \to \mathbb{0} + \text{merge}\,(t)) \stackrel{?}{=} \text{sw}^{\uparrow N} \fatsemi (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{ftn}_P \quad . \tag{14.57}
$$

Apply both sides of Eq. (14.57) to an arbitrary $n : N^{P^{M^A}}$. The left-hand side becomes:

$$n \triangleright \gamma^{\uparrow N} \fatsemi \text{ftn}_N \fatsemi (t \to \mathbb{0} + \text{merge}\,(t)) = \mathbb{0} + n \triangleright \gamma^{\uparrow N} \triangleright \text{ftn}_N \triangleright \text{merge} \quad .$$

The right-hand side of Eq. (14.57) applied to n is:

$$n \triangleright \text{sw}^{\uparrow N} \fatsemi (t \to \mathbb{0} + \text{merge}\,(t)) \fatsemi \text{ftn}_P = (\mathbb{0} + n \triangleright \text{sw}^{\uparrow N} \triangleright \text{merge}) \triangleright \text{ftn}_P$$

$$= (\mathbb{0} + n \triangleright \text{sw}^{\uparrow N} \fatsemi \text{merge}) \triangleright \left| \begin{array}{c|c|c} & A & N^A \\ \hline A & \text{id} & \mathbb{0} \\ N^A & \mathbb{0} & \text{id} \\ N^{P^A} & \mathbb{0} & \gamma^{\uparrow N} \fatsemi \text{ftn}_N \end{array} \right| = \mathbb{0} + n \triangleright \text{sw}^{\uparrow N} \fatsemi \text{merge} \fatsemi \gamma^{\uparrow N} \fatsemi \text{ftn}_N \quad .$$

The remaining difference is between two functions of type $N^{P^{M^A}} \to N^A$:

$$\gamma^{\uparrow N} \fatsemi \text{ftn}_N \fatsemi \text{merge} \stackrel{?}{=} \text{sw}^{\uparrow N} \fatsemi \text{merge} \fatsemi \gamma^{\uparrow N} \fatsemi \text{ftn}_N \quad . \tag{14.58}$$

The left-hand side of Eq. (14.58) can be rewritten as:

$$\begin{aligned} & \gamma^{\uparrow N} \fatsemi \text{ftn}_N \fatsemi \text{merge} = \gamma^{\uparrow N} \fatsemi \underline{\text{ftn}_N \fatsemi \text{flift}_K^{\uparrow N}} \fatsemi \text{ftn}_N \\ \text{naturality of ftn}_N : \quad & = \gamma^{\uparrow N} \fatsemi \text{flift}_K^{\uparrow N \uparrow N} \fatsemi \underline{\text{ftn}_N \fatsemi \text{ftn}_N} \\ \text{associativity law of ftn}_N : \quad & = \gamma^{\uparrow N} \fatsemi \text{flift}_K^{\uparrow N \uparrow N} \fatsemi \text{ftn}_N^{\uparrow N} \fatsemi \text{ftn}_N = \left(\gamma \fatsemi \text{flift}_K^{\uparrow N} \fatsemi \text{ftn}_N\right)^{\uparrow N} \fatsemi \text{ftn}_N \quad . \end{aligned}$$

The right-hand side of Eq. (14.58) is

$$\begin{aligned} & \text{sw}^{\uparrow N} \fatsemi \text{merge} \fatsemi \gamma^{\uparrow N} \fatsemi \text{ftn}_N \\ \text{naturality of merge} : \quad & = \text{sw}^{\uparrow N} \fatsemi \gamma^{\uparrow M \uparrow N} \fatsemi \text{merge} \fatsemi \text{ftn}_N = \text{sw}^{\uparrow N} \fatsemi \gamma^{\uparrow M \uparrow N} \fatsemi \text{flift}_K^{\uparrow N} \fatsemi \underline{\text{ftn}_N \fatsemi \text{ftn}_N} \\ \text{associativity law of ftn}_N : \quad & = \left(\text{sw} \fatsemi \gamma^{\uparrow M} \fatsemi \text{flift}_K \fatsemi \text{ftn}_N\right)^{\uparrow N} \fatsemi \text{ftn}_N \quad . \end{aligned}$$

The remaining difference is now between two functions of type $P^{M^A} \to N^A$:

$$\gamma \fatsemi \text{flift}_K^{\uparrow N} \fatsemi \text{ftn}_N \stackrel{?}{=} \text{sw} \fatsemi \gamma^{\uparrow M} \fatsemi \text{flift}_K \fatsemi \text{ftn}_N \quad . \tag{14.59}$$

The left-hand side of Eq. (14.59) is:

$$\gamma \fatsemi \text{flift}_K^{\uparrow N} \fatsemi \text{ftn}_N = \left| \begin{array}{c|c} & N^{M^A} \\ \hline M^A & \text{pu}_N \\ N^{M^A} & \text{id} \end{array} \right| \fatsemi \text{flift}_K^{\uparrow N} \fatsemi \text{ftn}_N = \left| \begin{array}{c|c} & N^A \\ \hline M^A & \underline{\text{pu}_N \fatsemi \text{flift}_K^{\uparrow N}} \fatsemi \text{ftn}_N \\ N^{M^A} & \text{id} \fatsemi \text{flift}_K^{\uparrow N} \fatsemi \text{ftn}_N \end{array} \right|$$

$$= \left| \begin{array}{c|c} & N^A \\ \hline M^A & \text{flift}_K \fatsemi \underline{\text{pu}_N \fatsemi \text{ftn}_N} \\ N^{M^A} & \text{id} \fatsemi \underline{\text{flift}_K^{\uparrow N} \fatsemi \text{ftn}_N} \end{array} \right| = \left| \begin{array}{c|c} & N^A \\ \hline M^A & \text{flift}_K \\ N^{M^A} & \text{merge} \end{array} \right| \quad .$$

To proceed with the right-hand side of Eq. (14.59), we compute some intermediate expressions:

$$\mathrm{sw} \fatsemi \gamma^{\uparrow M} = \left| \begin{array}{c|c} & M^{N^A} \\ \hline M^A & \mathrm{pu}_P^{\uparrow M} \fatsemi \gamma^{\uparrow M} \\ N^{M^A} & (t \to \mathbb{0} + \mathrm{merge}\,(t)) \fatsemi \mathrm{pu}_M \fatsemi \gamma^{\uparrow M} \end{array} \right| \quad ,$$

$$\mathrm{pu}_P \fatsemi \gamma = (a^{:A} \to a + \mathbb{0}) \fatsemi \left| \begin{array}{c|c} & N^A \\ \hline A & \mathrm{pu}_N \\ N^A & \mathrm{id} \end{array} \right| = a^{:A} \to \mathrm{pu}_N\,(a) = \mathrm{pu}_N \quad ,$$

$$(t \to \mathbb{0} + \mathrm{merge}\,(t)) \fatsemi \mathrm{pu}_M \fatsemi \gamma^{\uparrow M} = (t \to \mathbb{0} + \mathrm{merge}\,(t)) \fatsemi \gamma \fatsemi \mathrm{pu}_M$$

$$= (t \to \mathbb{0} + \mathrm{merge}\,(t)) \fatsemi \left| \begin{array}{c|c} & N^A \\ \hline A & \mathrm{pu}_N \\ N^A & \mathrm{id} \end{array} \right| \fatsemi \mathrm{pu}_M = t \to \mathrm{merge}\,(t) \triangleright \mathrm{pu}_M = \mathrm{merge} \fatsemi \mathrm{pu}_M \quad .$$

Now we can reduce the right-hand side of Eq. (14.59) to:

$$\mathrm{sw} \fatsemi \gamma^{\uparrow M} \fatsemi \mathrm{flift}_K \fatsemi \mathrm{ftn}_N = \left| \begin{array}{c|c} & N^A \\ \hline M^A & \underline{\mathrm{pu}_N^{\uparrow M} \fatsemi \mathrm{flift}_K} \fatsemi \mathrm{ftn}_N \\ N^{M^A} & \mathrm{merge} \fatsemi \underline{\mathrm{pu}_M \fatsemi \mathrm{flift}_K} \fatsemi \mathrm{ftn}_N \end{array} \right|$$

$$= \left| \begin{array}{c|c} & N^A \\ \hline M^A & \mathrm{flift}_K \fatsemi \underline{\mathrm{pu}_N^{\uparrow N} \fatsemi \mathrm{ftn}_N} \\ N^{M^A} & \mathrm{merge} \fatsemi \underline{\mathrm{pu}_N \fatsemi \mathrm{ftn}_N} \end{array} \right| = \left| \begin{array}{c|c} & N^A \\ \hline M^A & \mathrm{flift}_K \\ N^{M^A} & \mathrm{merge} \end{array} \right| \quad .$$

This equals the left-hand side of Eq. (14.59).

Identity laws The identity laws hold: setting $M = \mathrm{Id}$ in the type of T_L^M gives the isomorphism:

$$T_L^{\mathrm{Id},A} \stackrel{\mathrm{def}}{=} \mathrm{Id}^{A+T_K^{\mathrm{Id},A}} \cong A + K^A \stackrel{\mathrm{def}}{=} L^A \quad .$$

The two directions of the isomorphism $A + T_K^{\mathrm{Id},A} \cong A + K^A$ are monad morphisms due to Statement 14.3.4.1, assuming that the identity law already holds for the transformer T_K. We also need the property that the free pointed construction preserves monad morphisms (Exercise 14.9.5.16). Then the assumed monad isomorphism $T_K^{\mathrm{Id}} \cong K$ extends to a monad isomorphism $A + T_K^{\mathrm{Id},A} \cong A + K^A$.

Lifting laws The function `flift` for T_L is defined by:

$$\mathrm{flift}_L : M^A \to M^{P^A} \quad , \qquad \mathrm{flift}_L \stackrel{\mathrm{def}}{=} \mathrm{pu}_P^{\uparrow M} \quad .$$

To show that `flift` is a monad morphism, we refer to Statement 14.3.5.1, where we need to use `blift` instead of `flift` and consider P as the base monad and M as the foreign monad.

Implementation of runners We cannot use Statement 14.3.6.1 to verify the runner laws for T_L, because the transformer $T_L^M = M \circ P$ is not compositional: the monad P uses in its definition the monad M as $P^A \stackrel{\mathrm{def}}{=} A + T_K^{M,A}$. Mapping the monad M to M' requires us also to map P to $P' \stackrel{\mathrm{def}}{=} A + T_K^{M',A}$. So, the runners (`frun` and `brun`) must be defined differently from those in Statement 14.3.6.1:

$$\mathrm{frun}_L^{M,M'} : (M \rightsquigarrow M') \to M^{P^A} \to M'^{P'^A} \quad , \quad \mathrm{frun}_L^{M,M'}(\phi^{:M \rightsquigarrow M'}) \stackrel{\mathrm{def}}{=} \left| \begin{array}{c|cc} & A & T_K^{M',A} \\ \hline A & \mathrm{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \mathrm{frun}_K^{M,M'}(\phi) \end{array} \right|^{\uparrow M} \fatsemi \phi \quad ,$$

$$\mathrm{brun}_L : (L \rightsquigarrow \mathrm{Id}) \to M^{P^A} \to M^A \quad , \quad \mathrm{brun}_L(\theta^{:L\rightsquigarrow\mathrm{Id}}) \stackrel{\mathrm{def}}{=} \left|\begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \\ T_K^{M,A} & \mathrm{brun}_K((k \to \mathbb{0}+k) \fatsemi \theta) \end{array}\right|^{\uparrow M} \fatsemi \mathrm{ftn}_M \quad .$$

The function $(k \to \mathbb{0}+k) \fatsemi \theta$ is a runner $K \rightsquigarrow \mathrm{Id}$ as shown by Exercise 14.9.5.18(d). If the transformer T_K^M does not have a general base runner, the definition of brun_L above will need to use any available base runner brun_K instead of $\mathrm{brun}_K((k \to \mathbb{0}+k) \fatsemi \theta)$.

If the transformer T_K has no base runner, we will be unable to define a base runner for T_L.

Monadic naturality of `swap` The code of `swap` is the same for any foreign monad M; it means that `swap` is natural in the monad parameter M. However, since the monad P depends on M, the monadic naturality of $\mathrm{sw}_{M,P}$ with respect to the monad M is not simply the monadic naturality of the function `swap[M[_], P[_], A]` with respect to the type parameter `M[_]`. For any monad morphism $\phi : M \rightsquigarrow M'$, we need to use the corresponding monad morphism between $P \rightsquigarrow P'$ (see Exercise 14.9.5.16), which we will temporarily denote by ψ:

$$\psi \stackrel{\mathrm{def}}{=} \left|\begin{array}{c||cc} & A & T_K^{M',A} \\ \hline A & \mathrm{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \mathrm{frun}_K^{M,M'}(\phi) \end{array}\right| \quad .$$

We now write the monadic naturality law relating $\mathrm{sw}_{M,P}$ and $\mathrm{sw}_{M',P'}$ like this:

$$\begin{array}{ccccc} P^{M^A} & \xrightarrow{\phi^{\uparrow P}} & P^{M'^A} & \xrightarrow{\psi} & P'^{M'^A} \\ \downarrow{\scriptstyle \mathrm{sw}_{M,P}} & & & & \downarrow{\scriptstyle \mathrm{sw}_{M',P'}} \\ M^{P^A} & \xrightarrow{\psi^{\uparrow M}} & M^{P'^A} & \xrightarrow{\phi} & M'^{P'^A} \end{array}$$

$$\mathrm{sw}_{M,P} \fatsemi \psi^{\uparrow M} \fatsemi \phi = \phi^{\uparrow P} \fatsemi \psi \fatsemi \mathrm{sw}_{M',P'} \quad . \tag{14.60}$$

To verify this law, substitute the definition of $\mathrm{sw}_{M,P}$ into its two sides:

$$\mathrm{sw}_{M,P} \fatsemi \psi^{\uparrow M} \fatsemi \phi = \left|\begin{array}{c||c} & M'^{P'^A} \\ \hline M^A & \mathrm{pu}_P^{\uparrow M} \fatsemi \psi^{\uparrow M} \fatsemi \phi \\ N^{M^A} & (t \to \mathbb{0} + \mathrm{merge}\,(t)) \fatsemi \mathrm{pu}_M \fatsemi \psi^{\uparrow M} \fatsemi \phi \end{array}\right| = \left\|\begin{array}{c} \mathrm{pu}_P^{\uparrow M} \fatsemi \psi^{\uparrow M} \fatsemi \phi \\ (t \to \mathbb{0} + \mathrm{merge}\,(t)) \fatsemi \mathrm{pu}_M \fatsemi \psi^{\uparrow M} \fatsemi \phi \end{array}\right|$$

$$= \left\|\begin{array}{c} \phi \fatsemi (\mathrm{pu}_P \fatsemi \psi)^{\uparrow M'} \\ (t \to \mathbb{0} + \mathrm{merge}\,(t)) \fatsemi \psi \fatsemi \mathrm{pu}_M \fatsemi \phi \end{array}\right| = \left\|\begin{array}{c} \phi \fatsemi \mathrm{pu}_{P'}^{\uparrow M'} \\ (t \to \mathbb{0} + \mathrm{merge}\,(t)) \fatsemi \psi \fatsemi \mathrm{pu}_{M'} \end{array}\right| \quad ,$$

$$\phi^{\uparrow P} \fatsemi \psi \fatsemi \mathrm{sw}_{M',P'} = \left|\begin{array}{c||cc} & M'^A & N^{M'^A} \\ \hline M^A & \phi & \mathbb{0} \\ N^{M^A} & \mathbb{0} & \phi^{\uparrow N} \end{array}\right| \fatsemi \left|\begin{array}{c||cc} & M'^A & T_K^{M',M'^A} \\ \hline M'^A & \mathrm{id} & \mathbb{0} \\ T_K^{M,M'^A} & \mathbb{0} & \mathrm{frun}_K^{M,M'}(\phi) \end{array}\right| \fatsemi \mathrm{sw}_{M',P'}$$

$$= \left|\begin{array}{c||cc} & M'^A & T_K^{M',M'^A} \\ \hline M^A & \phi & \mathbb{0} \\ T_K^{M,M^A} & \mathbb{0} & \phi^{\uparrow N} \fatsemi \mathrm{frun}_K^{M,M'}(\phi) \end{array}\right| \fatsemi \left|\begin{array}{c||c} & M'^{P'^A} \\ \hline M'^A & \mathrm{pu}_{P'}^{\uparrow M'} \\ T_K^{M',M'^A} & (t \to \mathbb{0} + \mathrm{merge}\,(t)) \fatsemi \mathrm{pu}_{M'} \end{array}\right|$$

$$= \left|\begin{array}{c||c} & M'^{P'^A} \\ \hline M^A & \phi \fatsemi \mathrm{pu}_{P'}^{\uparrow M'} \\ T_K^{M,M^A} & \phi^{\uparrow N} \fatsemi \mathrm{frun}_K^{M,M'}(\phi) \fatsemi (t \to \mathbb{0} + \mathrm{merge}\,(t)) \fatsemi \mathrm{pu}_{M'} \end{array}\right| \quad .$$

It remains to show the equality of two functions of type $T_K^{M,M^A} \to A + T_K^{M',A} = N^{M^A} \to A + T_K^{M',A}$:

$$\phi^{\uparrow N} ⨾ \text{frun}_K^{M,M'}(\phi) ⨾ (t \to \mathbb{0} + \text{merge}\,(t)) \stackrel{?}{=} (t \to \mathbb{0} + \text{merge}\,(t)) ⨾ \psi$$

$$= (t \to \mathbb{0} + \text{merge}\,(t)) ⨾ \left| \begin{array}{c||cc} & A & T_K^{M',A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \text{frun}_K^{M,M'}(\phi) \end{array} \right| = t \to \mathbb{0} + t ▷ \text{merge} ▷ \text{frun}_K^{M,M'}(\phi) \quad .$$

Since both functions produce results of type $\mathbb{0}^{:A} + T_K^{M',A}$, we need to compare the following two functions of type $N^{M^A} \to N'^A$, where we denote for brevity $N'^A \stackrel{\text{def}}{=} T_K^{M',A}$:

$$\phi^{\uparrow N} ⨾ \text{frun}_K^{M,M'}(\phi) ⨾ \text{merge} \stackrel{?}{=} \text{merge} ⨾ \text{frun}_K^{M,M'}(\phi) \quad ,$$

$$\text{or equivalently}: \quad \phi^{\uparrow N} ⨾ \text{frun}_K^{M,M'}(\phi) ⨾ \text{flift}_K^{\uparrow N'} ⨾ \text{ftn}_{N'} \stackrel{?}{=} \text{flift}_K^{\uparrow N} ⨾ \text{ftn}_N ⨾ \text{frun}_K^{M,M'}(\phi) \quad . \tag{14.61}$$

Rewrite the left-hand side of Eq. (14.61) so that frun (ϕ) is moved to the right in the composition:

$$\phi^{\uparrow N} ⨾ \underline{\text{frun}_K^{M,M'}(\phi) ⨾ \text{flift}_K^{\uparrow N'}} ⨾ \text{ftn}_{N'} = (\underline{\phi ⨾ \text{flift}_K})^{\uparrow N} ⨾ \text{frun}_K^{M,M'}(\phi) ⨾ \text{ftn}_{N'}$$

$$\text{use law (14.5)}: \quad = \text{flift}_K^{\uparrow N} ⨾ \underline{\left(\text{frun}_K^{M,M'}(\phi)\right)^{\uparrow N} ⨾ \text{frun}_K^{M,M'}(\phi) ⨾ \text{ftn}_{N'}}$$

$$\text{composition law of frun}_K: \quad = \text{flift}_K^{\uparrow N} ⨾ \text{ftn}_N ⨾ \text{frun}_K^{M,M'}(\phi) \quad .$$

This equals the right-hand side of Eq. (14.61), proving the monadic naturality law (14.60).

Runner laws To show that $\text{frun}_L^{M,M'}(\phi)$ is a monad morphism, first verify the identity law:

$$\text{expect to equal pu}_{M' \circ P'}: \quad \text{pu}_{M \circ P} ⨾ \text{frun}_L^{M,M'}(\phi) = \text{pu}_P ⨾ \underline{\text{pu}_M ⨾ \psi^{\uparrow M}} ⨾ \phi$$

$$\text{naturality law of pu}_M: \quad = \underline{\text{pu}_P ⨾ \psi} ⨾ \underline{\text{pu}_M ⨾ \phi}$$

$$\text{identity laws of } \phi \text{ and } \psi: \quad = \text{pu}_{P'} ⨾ \text{pu}_{M'} = \text{pu}_{M' \circ P'} \quad .$$

To verify the composition law, express $\text{frun}_L^{M,M'} = \psi^{\uparrow M} ⨾ \phi$ and recall that $f^{\uparrow(M \circ P)} = f^{\uparrow P \uparrow M}$:

$$\text{left-hand side}: \quad (\psi^{\uparrow M} ⨾ \phi)^{\uparrow(M \circ P)} ⨾ \psi^{\uparrow M} ⨾ \phi ⨾ \text{ftn}_{M' \circ P'}$$

$$\text{definition of ftn}_{M' \circ P'}: \quad = (\psi^{\uparrow M} ⨾ \phi)^{\uparrow(M \circ P)} ⨾ \psi^{\uparrow M} ⨾ \phi ⨾ \text{sw}_{M',P'}^{\uparrow M'} ⨾ \text{ftn}_{M'} ⨾ \text{ftn}_{P'}^{\uparrow M'} \quad ,$$

$$\text{right-hand side}: \quad \text{ftn}_{M \circ P} ⨾ \psi^{\uparrow M} ⨾ \phi = \text{sw}_{M,P}^{\uparrow M} ⨾ \text{ftn}_M ⨾ \text{ftn}_P^{\uparrow M} ⨾ \psi^{\uparrow M} ⨾ \phi \quad .$$

We need to use various laws to rewrite the left-hand side so that the functions ψ and ϕ are moved to the right in the function composition:

$$\text{left-hand side}: \quad (\psi^{\uparrow M} ⨾ \phi)^{\uparrow(M \circ P)} ⨾ \psi^{\uparrow M} ⨾ \phi ⨾ \text{ftn}_{M' \circ P'}$$

$$\text{definition of ftn}_{M' \circ P'}: \quad = (\psi^{\uparrow M} ⨾ \phi)^{\uparrow(M \circ P)} ⨾ \psi^{\uparrow M} ⨾ \underline{\phi ⨾ \text{sw}_{M',P'}^{\uparrow M'}} ⨾ \text{ftn}_{M'} ⨾ \text{ftn}_{P'}^{\uparrow M'}$$

$$\text{naturality law of } \phi: \quad = \underline{(\psi^{\uparrow M} ⨾ \phi)^{\uparrow P \uparrow M} ⨾ \psi^{\uparrow M} ⨾ \text{sw}_{M',P'}^{\uparrow M}} ⨾ \phi ⨾ \text{ftn}_{M'} ⨾ \text{ftn}_{P'}^{\uparrow M'}$$

$$\text{composition under } {}^{\uparrow M}: \quad = \left(\psi^{\uparrow M \uparrow P} ⨾ \underline{\phi^{\uparrow P} ⨾ \psi ⨾ \text{sw}_{M',P'}}\right)^{\uparrow M} ⨾ \phi ⨾ \text{ftn}_{M'} ⨾ \text{ftn}_{P'}^{\uparrow M'}$$

$$\text{use Eq. (14.60) for sw}_{M',P'}: \quad = \left(\psi^{\uparrow M \uparrow P} ⨾ \text{sw}_{M,P} ⨾ \psi^{\uparrow M} ⨾ \phi\right)^{\uparrow M} ⨾ \phi ⨾ \underline{\text{ftn}_{M'}} ⨾ \text{ftn}_{P'}^{\uparrow M'}$$

$$\text{composition law of } \phi: \quad = \left(\underline{\psi^{\uparrow M \uparrow P} ⨾ \text{sw}_{M,P}} ⨾ \underline{\psi^{\uparrow M}}\right)^{\uparrow M} ⨾ \text{ftn}_M ⨾ \underline{\phi ⨾ \text{ftn}_{P'}^{\uparrow M'}}$$

$$\text{naturality of sw, } \phi \text{, and ftn}_M: \quad = \left(\text{sw}_{M,P} ⨾ \underline{\psi^{\uparrow P \uparrow M}}\right)^{\uparrow M} ⨾ \text{ftn}_M ⨾ \psi^{\uparrow M} ⨾ \text{ftn}_{P'}^{\uparrow M} ⨾ \phi$$

$$\text{naturality law of ftn}_M: \quad = \text{sw}_{M,P}^{\uparrow P} ⨾ \text{ftn}_M ⨾ \left(\underline{\psi^{\uparrow P} ⨾ \psi ⨾ \text{ftn}_{P'}}\right)^{\uparrow M} ⨾ \phi$$

$$\text{composition law of } \psi: \quad = \text{sw}_{M,P}^{\uparrow P} ⨾ \text{ftn}_M ⨾ (\text{ftn}_P ⨾ \psi)^{\uparrow M} ⨾ \phi = \text{sw}_{M,P}^{\uparrow P} ⨾ \text{ftn}_M ⨾ \text{ftn}_P^{\uparrow M} ⨾ \psi^{\uparrow M} ⨾ \phi \quad .$$

Both sides are now equal, which concludes the proof of the monad morphism laws of frun_L.

Functor laws of `frun` To verify the functor laws of frun_L, we assume that the functor laws already hold for frun_K. The identity law of frun_L is:

$$\text{frun}_L^{M,M}(\text{id}) = \left|\begin{array}{c||cc} & A & T_K^{M,A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \text{frun}_K^{M,M}(\text{id}) \end{array}\right|^{\uparrow M} \fatsemi \text{id} = \left|\begin{array}{c||cc} & A & T_K^{M,A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \text{id} \end{array}\right|^{\uparrow M} = \text{id}^{\uparrow M} = \text{id} \quad .$$

To verify the composition law (14.4) for any monad morphisms $\phi_1 : Q \rightsquigarrow R$ and $\phi_2 : R \rightsquigarrow S$, where Q, R, S are some monads, denote temporarily by ψ_1 and ψ_2 the following monad morphisms:

$$\psi_1 : A + T_K^{Q,A} \rightsquigarrow A + T_K^{R,A} \quad , \qquad \psi_1 \stackrel{\text{def}}{=} \left|\begin{array}{c||cc} & A & T_K^{R,A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{Q,A} & \mathbb{0} & \text{frun}_K^{Q,R}(\phi_1) \end{array}\right| \quad ,$$

$$\psi_2 : A + T_K^{R,A} \rightsquigarrow A + T_K^{S,A} \quad , \qquad \psi_2 \stackrel{\text{def}}{=} \left|\begin{array}{c||cc} & A & T_K^{S,A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{R,A} & \mathbb{0} & \text{frun}_K^{R,S}(\phi_2) \end{array}\right| \quad .$$

With these notations, we write the left side of the functor composition law:

$$\begin{aligned} \text{composition law of frun}_L : \quad & \text{frun}_L^{Q,R}(\phi_1) \fatsemi \text{frun}_L^{R,S}(\phi_2) = \psi_1^{\uparrow Q} \fatsemi \underline{\phi_1 \fatsemi \psi_2^{\uparrow R}} \fatsemi \phi_2 \\ \text{naturality of } \phi_1 : \quad & = \psi_1^{\uparrow Q} \fatsemi \psi_2^{\uparrow Q} \fatsemi \phi_1 \fatsemi \phi_2 = (\psi_1 \fatsemi \psi_2)^{\uparrow Q} \fatsemi \phi \fatsemi \phi_2 \quad . \end{aligned}$$

Compute the composition $\psi_1 \fatsemi \psi_2$ separately, using the composition law of frun_K:

$$\psi_1 \fatsemi \psi_2 = \left|\begin{array}{c||cc} & A & T_K^{R,A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{Q,A} & \mathbb{0} & \text{frun}_K^{Q,R}(\phi_1) \end{array}\right| \fatsemi \left|\begin{array}{c||cc} & A & T_K^{S,A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{R,A} & \mathbb{0} & \text{frun}_K^{R,S}(\phi_2) \end{array}\right|$$

$$\text{matrix composition :} \quad = \left|\begin{array}{c||cc} & A & T_K^{S,A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{Q,A} & \mathbb{0} & \underline{\text{frun}_K^{Q,R}(\phi_1) \fatsemi \text{frun}_K^{R,S}(\phi_2)} \end{array}\right| = \left|\begin{array}{c||cc} & A & T_K^{S,A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{Q,A} & \mathbb{0} & \text{frun}_K^{Q,S}(\phi_1 \fatsemi \phi_2) \end{array}\right| \quad .$$

We can now finish verifying the composition law:

$$\begin{aligned} & \text{frun}_L^{Q,R}(\phi_1) \fatsemi \text{frun}_L^{R,S}(\phi_2) = (\psi_1 \fatsemi \psi_2)^{\uparrow Q} \fatsemi \phi_1 \fatsemi \phi_2 \\ & = \left|\begin{array}{c||cc} & A & T_K^{S,A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{Q,A} & \mathbb{0} & \text{frun}_K^{Q,S}(\phi_1 \fatsemi \phi_2) \end{array}\right|^{\uparrow Q} \fatsemi \phi_1 \fatsemi \phi_2 = \text{frun}_L^{Q,S}(\phi_1 \fatsemi \phi_2) \quad . \end{aligned}$$

□

The derivation of the base runner laws used the following property:

Statement 14.6.2.2 Assume M and N are two monads such that the functor composition $M \circ N$ is a monad defined via a `swap` function ($\text{sw}_{M,N} : N \circ M \rightsquigarrow M \circ N$) according to Statement 14.3.2.1 (where the two monads are denoted by M, N instead of L, M). Let $\phi : N \rightsquigarrow M$ be a monad morphism such that the `swap` function obeys the law (14.62) shown below. That law can be seen as a weaker version of a monadic naturality law of $\text{sw}_{M,N}$ with respect to the parameter N (where the monad N is transformed by ϕ to the fixed monad M rather than to arbitrary new monad):

$$\phi \,\text{⨾}\, \text{ftn}_M = \text{sw}_{M,N} \,\text{⨾}\, \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \quad . \tag{14.62}$$

$$\begin{tikzcd}
N^{M^A} \arrow[r, "\phi"] \arrow[d, "\text{sw}_{M,N}"] & M^{M^A} \arrow[dr, "\text{ftn}_M"] & \\
M^{N^A} \arrow[r, "\phi^{\uparrow M}"] & M^{M^A} \arrow[r, "\text{ftn}_M"] & M^A
\end{tikzcd}$$

Then the function $\phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M$ is a monad morphism $M \circ N \rightsquigarrow M$.

Proof We need to check the laws of the monad morphism:

$$\text{identity law}: \quad \text{pu}_{M\circ N} \,\text{⨾}\, \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \stackrel{?}{=} \text{pu}_M \quad ,$$

$$\text{composition law}: \quad \text{ftn}_{M\circ N} \,\text{⨾}\, \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \stackrel{?}{=} \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \,\text{⨾}\, (\phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M)^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \quad .$$

To verify the identity law:

$$\begin{aligned}
\text{expect to equal } \text{pu}_M: \quad & \text{pu}_{M\circ N} \,\text{⨾}\, \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M = \text{pu}_N \,\text{⨾}\, \underline{\text{pu}_M \,\text{⨾}\, \phi^{\uparrow M}} \,\text{⨾}\, \text{ftn}_M \\
& = \underline{\text{pu}_N \,\text{⨾}\, \phi} \,\text{⨾}\, \underline{\text{pu}_M \,\text{⨾}\, \text{ftn}_M} = \text{pu}_M \,\text{⨾}\, \text{id} = \text{pu}_M \quad .
\end{aligned}$$

To verify the composition law, begin with its left-hand side:

$$\begin{aligned}
\text{left-hand side}: \quad & \text{ftn}_{M\circ N} \,\text{⨾}\, \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M = \text{sw}^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \,\text{⨾}\, \underline{\text{ftn}_N^{\uparrow M} \,\text{⨾}\, \phi^{\uparrow M}} \,\text{⨾}\, \text{ftn}_M \\
\text{composition law of } \phi: \quad & = \text{sw}^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \,\text{⨾}\, \left(\phi \,\text{⨾}\, \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M\right)^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \quad .
\end{aligned}$$

In order to use the law (14.62), we need to move ϕ towards $\text{sw}^{\uparrow M}$ in the composition and to replace ftn_M by $\text{ftn}_M^{\uparrow M}$. To achieve that, we repeatedly use ftn_M's laws of naturality and associativity:

$$\begin{aligned}
& \text{sw}^{\uparrow M} \,\text{⨾}\, \underline{\text{ftn}_M \,\text{⨾}\, (\phi \,\text{⨾}\, \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M)^{\uparrow M}} \,\text{⨾}\, \text{ftn}_M = \text{sw}^{\uparrow M} \,\text{⨾}\, \phi^{\uparrow M \uparrow M} \,\text{⨾}\, \text{ftn}_M \,\text{⨾}\, \underline{\phi^{\uparrow M \uparrow M} \,\text{⨾}\, \text{ftn}_M} \,\text{⨾}\, \text{ftn}_M \\
& = (\text{sw} \,\text{⨾}\, \phi^{\uparrow M})^{\uparrow M} \,\text{⨾}\, \underline{\text{ftn}_M \,\text{⨾}\, \text{ftn}_M} \,\text{⨾}\, \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M = \underline{(\text{sw} \,\text{⨾}\, \phi^{\uparrow M})^{\uparrow M} \,\text{⨾}\, \text{ftn}_M^{\uparrow M}} \,\text{⨾}\, \text{ftn}_M \,\text{⨾}\, \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \\
\text{use Eq. (14.62)}: \quad & = (\phi \,\text{⨾}\, \underline{\text{ftn}_M})^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \,\text{⨾}\, \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M = \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \,\text{⨾}\, \text{ftn}_M \,\text{⨾}\, \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \quad .
\end{aligned}$$

Now we can rewrite the right-hand side of the law and make it equal to the last expression:

$$\begin{aligned}
\text{right-hand side}: \quad & \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \,\text{⨾}\, (\phi^{\uparrow M} \,\text{⨾}\, \underline{\text{ftn}_M})^{\uparrow M} \,\text{⨾}\, \text{ftn}_M = \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \,\text{⨾}\, \underline{\phi^{\uparrow M \uparrow M} \,\text{⨾}\, \text{ftn}_M} \,\text{⨾}\, \text{ftn}_M \\
& = \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \,\text{⨾}\, \text{ftn}_M \,\text{⨾}\, \phi^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \quad .
\end{aligned}$$

Base runner laws Given any base runner brun_K for the monad transformer T_K, we define the base runner brun_L through the special runner $\theta_P : P^A \to M^A$ that we implement through brun_K:

$$\text{brun}_L : M^{A+T_K^{M,A}} \to M^A \quad , \qquad \text{brun}_L \stackrel{\text{def}}{=} \theta_P^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \quad , \qquad \theta_P \stackrel{\text{def}}{=} \begin{array}{|c||c|} & M^A \\ \hline A & \text{pu}_M \\ T_K^{M,A} & \text{brun}_K \end{array} \quad . \tag{14.63}$$

The runner brun_L is not purely compositional since $L \stackrel{\text{def}}{=} M \circ P$ contains the monad P that again depends on M. So, we cannot use the proofs in Statement 14.3.6.1, and the monad morphism laws and the nondegeneracy law of brun_L need to be verified separately.

Note that $\theta_P : P \rightsquigarrow M$ is a monad morphism by Exercise 14.9.5.18.

To show that brun_L is a monad morphism, we will use Statement 14.6.2.2 where we use the monad $P^A \stackrel{\text{def}}{=} A + N^A$ instead of L^A. It remains to check the properties required by Statement 14.6.2.2. The required monad morphism ϕ is $\theta_P : P \rightsquigarrow M$. The special law of `swap` has the following form:

$$\theta_P \,\text{⨾}\, \text{ftn}_M \stackrel{?}{=} \text{sw} \,\text{⨾}\, \theta_P^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \quad .$$

Let us verify this law using Eq. (14.55) and the definition (14.63) of θ_P:

$$\text{expect to equal } \theta_P \,⨾\, \text{ftn}_M : \quad \text{sw} \,⨾\, \theta_P^{\uparrow M} \,⨾\, \text{ftn}_M = \begin{array}{c||c|} & M^A \\ \hline M^A & \underline{\text{pu}_P^{\uparrow M} \,⨾\, \theta_P^{\uparrow M}} \,⨾\, \text{ftn}_M \\ N^{M^A} & (t \to \mathbb{0}^{:A} + \text{merge}\,(t)) \,⨾\, \underline{\text{pu}_M \,⨾\, \theta_P^{\uparrow M}} \,⨾\, \text{ftn}_M \end{array}$$

$$\text{identity law of } \theta_P : \quad = \begin{array}{c||c|} & M^A \\ \hline M^A & \underline{\text{pu}_M^{\uparrow M} \,⨾\, \text{ftn}_M} \\ N^{M^A} & (t \to \mathbb{0}^{:A} + \text{merge}\,(t)) \,⨾\, \theta_P \,⨾\, \underline{\text{pu}_M \,⨾\, \text{ftn}_M} \end{array}$$

$$\text{identity laws of } M : \quad = \begin{array}{c||c|} & M^A \\ \hline M^A & \text{id} \\ N^{M^A} & (t \to \underline{\mathbb{0}^{:A} + \text{merge}\,(t)} \,⨾\, \theta_P \end{array} = \begin{array}{c||c|} & M^A \\ \hline M^A & \text{id} \\ N^{M^A} & \text{merge} \,⨾\, \text{brun}_K \end{array} .$$

Write out $\theta_P \,⨾\, \text{ftn}_M$ in matrix form:

$$\theta_P \,⨾\, \text{ftn}_M = \begin{array}{c||c|} & M^A \\ \hline M^A & \text{pu}_M \,⨾\, \text{ftn}_M \\ N^{M^A} & \text{brun}_K \,⨾\, \text{ftn}_M \end{array} = \begin{array}{c||c|} & M^A \\ \hline M^A & \text{id} \\ N^{M^A} & \text{brun}_K \,⨾\, \text{ftn}_M \end{array} .$$

The remaining difference is between the following functions of type $N^{M^A} \to M^A$:

$$\text{merge} \,⨾\, \text{brun}_K \stackrel{?}{=} \text{brun}_K \,⨾\, \text{ftn}_M \quad ,$$

$$\text{or, using the definition of merge :} \quad \text{flift}_K^{\uparrow N} \,⨾\, \text{ftn}_N \,⨾\, \text{brun}_K \stackrel{?}{=} \text{brun}_K \,⨾\, \text{ftn}_M \quad .$$

The monad morphism laws of brun_K allow us to prove this equation:

$$\text{expect to equal } \text{brun}_K \,⨾\, \text{ftn}_M : \quad \text{flift}_K^{\uparrow N} \,⨾\, \underline{\text{ftn}_N \,⨾\, \text{brun}_K} = \underline{\text{flift}_K^{\uparrow N} \,⨾\, \text{brun}_K^{\uparrow N}} \,⨾\, \text{brun}_K \,⨾\, \text{ftn}_M$$

$$\text{non-degeneracy law of } \text{brun}_K : \quad = \text{brun}_K \,⨾\, \text{ftn}_M \quad .$$

This concludes the proof of monad morphism laws of brun_L.

To verify the nondegeneracy law, which is an equation between functions of type $M^A \to M^A$:

$$\text{expect to equal id :} \quad \text{flift}_L \,⨾\, \text{brun}_L = \text{pu}_P^{\uparrow M} \,⨾\, \text{brun}_L = \underline{\text{pu}_P^{\uparrow M} \,⨾\, \theta_P^{\uparrow M}} \,⨾\, \text{ftn}_M = \underline{\text{pu}_M^{\uparrow M} \,⨾\, \text{ftn}_M} = \text{id} \quad .$$

Monadic naturality laws The monadic naturality law of flift_L is written as:

$$\text{flift}_L \,⨾\, \text{frun}_L(\phi) \stackrel{?}{=} \phi \,⨾\, \text{flift}_L \quad ,$$

$$\text{or, using definitions of flift}_L \text{ and frun}_L : \quad \text{pu}_P^{\uparrow M} \,⨾\, \begin{array}{c||cc|} & A & T_K^{M',A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \text{frun}_K^{M,M'}(\phi) \end{array}^{\uparrow M} \,⨾\, \phi \stackrel{?}{=} \phi \,⨾\, \text{pu}_P^{\uparrow M'} \quad .$$

This should hold for any monad morphism $\phi^{:M \rightsquigarrow M'}$. Simplify the expression under $^{\uparrow M}$ separately:

$$\text{pu}_P \,⨾\, \begin{array}{c||cc|} & A & T_K^{M',A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \text{frun}_K^{M,M'}(\phi) \end{array} = (a \to a + \mathbb{0}^{:T_K^{M,A}}) \,⨾\, \begin{array}{c||cc|} & A & T_K^{M',A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \text{frun}_K^{M,M'}(\phi) \end{array} = a \to a + \mathbb{0} \quad .$$

So, we can transform the left-hand side of the law and obtain the right-hand side:

$$\text{flift}_L \fatsemi \text{frun}_L(\phi) = (a \to a + \mathbb{0})^{\uparrow M} \fatsemi \phi = \underline{\text{pu}_P^{\uparrow M} \fatsemi \phi} = \phi \fatsemi \text{pu}_P^{\uparrow M'} \quad .$$

The monadic naturality law of brun_L is:

$$\text{brun}_L \fatsemi \phi \stackrel{?}{=} \text{frun}_L(\phi) \fatsemi \text{brun}_L \quad .$$

Rewrite the right-hand side using the definitions of brun_L and frun_L:

$$\text{expect to equal } \text{brun}_L \fatsemi \phi : \quad \text{frun}_L(\phi) \fatsemi \text{brun}_L = \left| \begin{array}{c||cc} & A & T_K^{M',A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \text{frun}_K^{M,M'}(\phi) \end{array} \right|^{\uparrow M} \fatsemi \underline{\phi \fatsemi \theta_{P'}^{\uparrow M'}} \fatsemi \text{ftn}_{M'}$$

$$\text{naturality law of } \phi : \quad = \left| \begin{array}{c||cc} & A & T_K^{M',A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \text{frun}_K^{M,M'}(\phi) \end{array} \right|^{\uparrow M} \fatsemi \theta_{P'}^{\uparrow M} \fatsemi \phi \fatsemi \text{ftn}_{M'} \quad .$$

Rewrite the expression under $^{\uparrow M}$ separately:

$$\left| \begin{array}{c||cc} & A & T_K^{M',A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \text{frun}_K^{M,M'}(\phi) \end{array} \right| \fatsemi \theta_{P'} = \left| \begin{array}{c||cc} & A & T_K^{M',A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \text{frun}_K^{M,M'}(\phi) \end{array} \right| \fatsemi \left| \begin{array}{c||c} & M'^A \\ \hline A & \text{pu}_{M'} \\ T_K^{M',A} & \text{brun}_K \end{array} \right|$$

$$= \left| \begin{array}{c||c} & M'^A \\ \hline A & \text{pu}_{M'} \\ T_K^{M,A} & \underline{\text{frun}_K^{M,M'}(\phi) \fatsemi \text{brun}_K} \end{array} \right| = \left| \begin{array}{c||c} & M'^A \\ \hline A & \underline{\text{pu}_{M'}} \\ T_K^{M,A} & \text{brun}_K \fatsemi \phi \end{array} \right|$$

$$= \left| \begin{array}{c||c} & M'^A \\ \hline A & \text{pu}_M \fatsemi \phi \\ T_K^{M,A} & \text{brun}_K \fatsemi \phi \end{array} \right| = \left| \begin{array}{c||c} & M^A \\ \hline A & \text{pu}_M \\ T_K^{M,A} & \text{brun}_K \end{array} \right| \fatsemi \phi = \theta_P \fatsemi \phi \quad .$$

Here we assumed that brun_K obeys its monadic naturality law. So, we can now write:

$$\left| \begin{array}{c||cc} & A & T_K^{M',A} \\ \hline A & \text{id} & \mathbb{0} \\ T_K^{M,A} & \mathbb{0} & \text{frun}_K^{M,M'}(\phi) \end{array} \right|^{\uparrow M} \fatsemi \theta_{P'}^{\uparrow M} \fatsemi \phi \fatsemi \text{ftn}_{M'} = \theta_P^{\uparrow M} \fatsemi \underline{\phi^{\uparrow M} \fatsemi \phi \fatsemi \text{ftn}_{M'}}$$

$$\text{composition law of } \phi : \quad = \underline{\theta_P^{\uparrow M} \fatsemi \text{ftn}_M} \fatsemi \phi = \text{brun}_L \fatsemi \phi \quad .$$

This derivation concludes the proof of the laws of the free pointed monad transformer.

14.7 Transformers for recursive monads

14.7.1 Transformer for free monads. Proofs

The free monad (denoted by $\text{Free}^{F,\bullet}$) on a given functor F is the following recursive type constructor:

$$\text{Free}^{F,A} \stackrel{\text{def}}{=} A + F^{\text{Free}^{F,A}} \quad .$$

Its monad transformer (denoted by $T^{M,\bullet}_{\text{Free}}$) is also defined recursively:

$$T^{M,A}_{\text{Free}} \stackrel{\text{def}}{=} M^{A+F^{T^{M,A}_{\text{Free}}}} \quad .$$

For brevity, we omit the superscript F in Free^F, since the functor F will be fixed throughout the derivations. When the foreign monad M is also fixed, we will denote $T^{M,A}_{\text{Free}}$ simply by T^A.

Statement 14.7.1.1 T_{Free} is a lawful monad transformer.

Proof We need to define the methods required of a monad transformer and verify all the laws.

Monad laws The recursively defined type of $T^{M,A}_{\text{Free}}$ contains infinitely many nested layers of the monad M composed with the functor F. We use the "unrolling trick" to express T_{Free} through simpler types. Write T_{Free} as an "infinite" type expression:

$$T^M_{\text{Free}} = M \circ (\text{Id} + F \circ M \circ (\text{Id} + F \circ M \circ (\text{Id} + ...))) \quad .$$

If we denote by L the type constructor defined recursively by:

$$L^A \stackrel{\text{def}}{=} A + F^{M^{L^A}} = A + F^{M^{A+F^{M^{A+...}}}} \quad ,$$

or equivalently by $L^A \stackrel{\text{def}}{=} \text{Free}^{F\circ M,A}$, we may express $T^{M,A}_{\text{Free}}$ as:

$$T^{M,A}_{\text{Free}} = M^{L^A} \quad , \qquad T^M_{\text{Free}} = M \circ L = M \circ \text{Free}^{F\circ M} \quad .$$

Since Free^P is a monad for any functor P, in particular for the functor $P \stackrel{\text{def}}{=} F \circ M$, it follows that L is a monad. It remains to show that the composition $M \circ L$ is also a monad. This resembles the situation with the composed-inside monad transformers having a base monad L and a foreign monad M. A significant difference is that the monad L also depends on the foreign monad M. That difference will play a role in proofs of laws that involve changing the foreign monad. But in this part of the proof, the foreign monad M is fixed throughout the derivation.

Similarly to the case of composed-inside transformers, we define a `swap` function:

$$\text{sw} : L^{M^A} \to M^{L^A} \quad , \qquad \text{sw} \stackrel{\text{def}}{=} \left| \begin{array}{c|c} & M^{A+(F\circ M\circ L)^A} \\ \hline M^A & (a^{:A} \to a + \mathbb{0}^{:(F\circ M\circ L)^A})^{\uparrow M} \\ (F\circ M\circ L\circ M)^A & \overline{\text{sw}}^{\uparrow M\uparrow F} ; \text{ftn}^{\uparrow F}_M ; (p \to \mathbb{0}^{:A} + p) ; \text{pu}_M \end{array} \right| \quad . \tag{14.64}$$

The upper row of the matrix contains the function $a \to a + \mathbb{0}$, which is the `pure` method of the monad L (denoted by pu_L). The lower row of the matrix contains a composition of four functions, the first of which being $\overline{\text{sw}}$, the recursive call to `swap`. To illustrate the types involved in the composition:

$$F^{M^{L^{M^A}}} \xrightarrow{\overline{\text{sw}}^{\uparrow M\uparrow F}} F^{M^{M^{L^A}}} \xrightarrow{\text{ftn}^{\uparrow F}_M} F^{M^{L^A}} \xrightarrow{p\to\mathbb{0}^{:A}+p} A + F^{M^{L^A}} \xrightarrow{\text{pu}_M} M^{A+F^{M^{L^A}}}$$

The required methods `pure` and `flatten` of the monad $T \stackrel{\text{def}}{=} M \circ L$ are defined by:

$$\text{pu}_T \stackrel{\text{def}}{=} \text{pu}_L ; \text{pu}_M \quad , \qquad \text{ftn}_T \stackrel{\text{def}}{=} \text{sw}^{\uparrow M} ; \text{ftn}_M ; \text{ftn}^{\uparrow M}_L \quad .$$

To prove that the composition $T \stackrel{\text{def}}{=} M \circ L$ is a lawful monad, we use Statement 14.3.2.1 with the monads L and M exchanged. That statement requires `swap` to satisfy its four laws:

$$\begin{aligned} \text{outer identity}: &\quad \text{pu}^{\uparrow L}_M ; \text{sw} = \text{pu}_M \quad , \\ \text{inner identity}: &\quad \text{pu}_L ; \text{sw} = \text{pu}^{\uparrow M}_L \quad , \\ \text{outer interchange}: &\quad \text{ftn}^{\uparrow L}_M ; \text{sw} = \text{sw} ; \text{sw}^{\uparrow M} ; \text{ftn}_M \quad , \\ \text{inner interchange}: &\quad \text{ftn}_L ; \text{sw} = \text{sw}^{\uparrow L} ; \text{sw} ; \text{ftn}^{\uparrow M}_L \quad . \end{aligned}$$

To verify these laws, we need the code for the lifting $(^{\uparrow L})$ as well as for the methods pu_L and ftn_L. Since L is a free monad on the functor $F \circ M$, we can adapt the code from Statement 10.2.8.10:

$$(f^{:A\to B})^{\uparrow L} = \left\| \begin{array}{c|cc} & B & F^{M^{L^B}} \\ \hline A & f & \mathbb{0} \\ F^{M^{L^A}} & \mathbb{0} & (f^{\overline{\uparrow L}})^{\uparrow M\uparrow F} \end{array} \right| \quad ,$$

$$\text{pu}_L \stackrel{\text{def}}{=} \left| \begin{array}{c|cc} & A & F^{M^{L^A}} \\ \hline A & \text{id} & \mathbb{0} \end{array} \right| \quad , \qquad \text{ftn}_L^{:L^{L^A}\to L^A} \stackrel{\text{def}}{=} \left\| \begin{array}{c|cc} & A & F^{M^{L^A}} \\ \hline A & \text{id} & \mathbb{0} \\ F^{M^{L^A}} & \mathbb{0} & \text{id} \\ F^{M^{L^{L^A}}} & \mathbb{0} & \overline{\text{ftn}}_L^{\uparrow M\uparrow F} \end{array} \right| \quad .$$

In the calculations, we will omit type annotations for brevity. To verify the outer identity law:

$$\text{pu}_M^{\uparrow L} ⨾ \text{sw} = \left\| \begin{array}{cc} \text{pu}_M & \mathbb{0} \\ \mathbb{0} & (\text{pu}_M^{\overline{\uparrow L}})^{\uparrow M\uparrow F} \end{array} \right\| ⨾ \left\| \begin{array}{c} (a^{:A} \to a + \mathbb{0}^{:(F\circ M\circ L)^A})^{\uparrow M} \\ \overline{\text{sw}}^{\uparrow M\uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \text{pu}_M \end{array} \right\|$$

$$= \left\| \begin{array}{c} \underline{\text{pu}_M ⨾ (a^{:A} \to a + \mathbb{0}^{:(F\circ M\circ L)^A})^{\uparrow M}} \\ \underline{(\text{pu}_M^{\overline{\uparrow L}})^{\uparrow M\uparrow F} ⨾ \overline{\text{sw}}^{\uparrow M\uparrow F}} ⨾ \text{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0} + p) ⨾ \text{pu}_M \end{array} \right\| = \left\| \begin{array}{c} (a^{:A} \to a + \mathbb{0}^{:(F\circ M\circ L)^A}) ⨾ \underline{\text{pu}_M} \\ \underline{(\text{pu}_M^{\overline{\uparrow L}} ⨾ \overline{\text{sw}})}^{\uparrow M\uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0} + p) ⨾ \underline{\text{pu}_M} \end{array} \right\|$$

$$= \left\| \begin{array}{c} a^{:A} \to a + \mathbb{0}^{:(F\circ M\circ L)^A} \\ \underline{(\text{pu}_M^{\uparrow M} ⨾ \text{ftn}_M)}^{\uparrow F} ⨾ (p \to \mathbb{0} + p) \end{array} \right\| ⨾ \text{pu}_M = \left| \begin{array}{c|c} & A + F^{M^{L^A}} \\ \hline A & a \to a + \mathbb{0} \\ F^{M^{L^A}} & p \to \mathbb{0}^{:A} + p \end{array} \right| ⨾ \text{pu}_M = \text{id} ⨾ \text{pu}_M = \text{pu}_M \quad .$$

To verify the inner identity law:

$$\text{pu}_L ⨾ \text{sw} = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \end{array} \right\| ⨾ \left\| \begin{array}{c} (a^{:A} \to a + \mathbb{0}^{:(F\circ M\circ L)^A})^{\uparrow M} \\ \overline{\text{sw}}^{\uparrow M\uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \text{pu}_M \end{array} \right\| = (a^{:A} \to a + \mathbb{0}^{:(F\circ M\circ L)^A})^{\uparrow M} = \text{pu}_L^{\uparrow M} \quad .$$

To verify the outer interchange law, write the two sides of the law separately:

$$\text{left-hand side}: \quad \text{ftn}_M^{\uparrow L} ⨾ \text{sw} = \left\| \begin{array}{cc} \text{ftn}_M & \mathbb{0} \\ \mathbb{0} & (\text{ftn}_M^{\overline{\uparrow L}})^{\uparrow M\uparrow F} \end{array} \right\| ⨾ \left\| \begin{array}{c} (a^{:A} \to a + \mathbb{0}^{:(F\circ M\circ L)^A})^{\uparrow M} \\ \overline{\text{sw}}^{\uparrow M\uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \text{pu}_M \end{array} \right\|$$

$$= \left\| \begin{array}{c} \text{ftn}_M ⨾ \text{pu}_L^{\uparrow M} \\ (\text{ftn}_M^{\overline{\uparrow L}})^{\uparrow M\uparrow F} ⨾ \overline{\text{sw}}^{\uparrow M\uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \text{pu}_M \end{array} \right\| \quad ,$$

$$\text{right-hand side}: \quad \text{sw} ⨾ \text{sw}^{\uparrow M} ⨾ \text{ftn}_M = \left\| \begin{array}{c} \underline{\text{pu}_L^{\uparrow M} ⨾ \text{sw}^{\uparrow M}} ⨾ \text{ftn}_M \\ \overline{\text{sw}}^{\uparrow M\uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \underline{\text{pu}_M ⨾ \text{sw}^{\uparrow M}} ⨾ \text{ftn}_M \end{array} \right\|$$

$$= \left\| \begin{array}{c} \underline{\text{pu}_L^{\uparrow M\uparrow M} ⨾ \text{ftn}_M} \\ \overline{\text{sw}}^{\uparrow M\uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \text{sw} ⨾ \underline{\text{pu}_M ⨾ \text{ftn}_M} \end{array} \right\|$$

$$= \left\| \begin{array}{c} \text{ftn}_M ⨾ \text{pu}_L^{\uparrow M} \\ \overline{\text{sw}}^{\uparrow M\uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \text{sw} \end{array} \right\| \quad .$$

The remaining difference is in the second rows of the matrices:

$$(\text{ftn}_M^{\overline{\uparrow L}})^{\uparrow M\uparrow F} ⨾ \overline{\text{sw}}^{\uparrow M\uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \text{pu}_M \stackrel{?}{=} \overline{\text{sw}}^{\uparrow M\uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \text{sw} \quad . \tag{14.65}$$

Let us rewrite both sides of this expression step by step. By the inductive assumption, we may use the outer interchange law for the recursive call to `swap`:

$$\text{ftn}_M^{\overline{\uparrow L}} \fatsemi \overline{\text{sw}} = \text{sw} \fatsemi \text{sw}^{\uparrow M} \fatsemi \text{ftn}_M \quad .$$

We can then rewrite the left-hand side of Eq. (14.65) as:

$$\begin{aligned}
& \underline{\left(\text{sw} \fatsemi \text{sw}^{\uparrow M} \fatsemi \text{ftn}_M\right)^{\uparrow M \uparrow F} \fatsemi \text{ftn}_M^{\uparrow F}} \fatsemi (p \to \mathbb{0}^{:A} + p) \fatsemi \text{pu}_M \\
\text{composition under } {}^{\uparrow F} : \quad & = \left(\text{sw}^{\uparrow M} \fatsemi \text{sw}^{\uparrow M \uparrow M} \fatsemi \underline{\text{ftn}_M^{\uparrow M} \fatsemi \text{ftn}_M}\right)^{\uparrow F} \fatsemi (p \to \mathbb{0}^{:A} + p) \fatsemi \text{pu}_M \\
\text{associativity law of } \text{ftn}_M : \quad & = \left(\text{sw}^{\uparrow M} \fatsemi \underline{\text{sw}^{\uparrow M \uparrow M} \fatsemi \text{ftn}_M} \fatsemi \text{ftn}_M\right)^{\uparrow F} \fatsemi (p \to \mathbb{0}^{:A} + p) \fatsemi \text{pu}_M \\
\text{naturality law of } \text{ftn}_M : \quad & = \left(\text{sw}^{\uparrow M} \fatsemi \text{ftn}_M \fatsemi \text{sw}^{\uparrow M} \fatsemi \text{ftn}_M\right)^{\uparrow F} \fatsemi (p \to \mathbb{0}^{:A} + p) \fatsemi \text{pu}_M \quad .
\end{aligned}$$

We can also compute the expression $(p \to \mathbb{0}^{:A} + p) \fatsemi \text{sw}$ in the right-hand side of Eq. (14.65):

$$\begin{aligned}
(p \to \mathbb{0} + p) \fatsemi \text{sw} &= \left\| \begin{array}{cc} \mathbb{0} & \text{id} \end{array} \right| \fatsemi \left\| \begin{array}{c} \text{pu}_L^{\uparrow M} \\ \overline{\text{sw}}^{\uparrow M \uparrow F} \fatsemi \text{ftn}_M^{\uparrow F} \fatsemi (p \to \mathbb{0} + p) \fatsemi \text{pu}_M \end{array} \right| \\
&= \left(\overline{\text{sw}}^{\uparrow M} \fatsemi \text{ftn}_M\right)^{\uparrow F} \fatsemi (p \to \mathbb{0} + p) \fatsemi \text{pu}_M \quad .
\end{aligned}$$

Then Eq. (14.65) becomes:

$$\begin{aligned}
& \left(\text{sw}^{\uparrow M} \fatsemi \text{ftn}_M \fatsemi \text{sw}^{\uparrow M} \fatsemi \text{ftn}_M\right)^{\uparrow F} \fatsemi (p \to \mathbb{0} + p) \fatsemi \text{pu}_M \\
& \stackrel{?}{=} \underline{\overline{\text{sw}}^{\uparrow M \uparrow F} \fatsemi \text{ftn}_M^{\uparrow F}} \fatsemi \left(\overline{\text{sw}}^{\uparrow M} \fatsemi \text{ftn}_M\right)^{\uparrow F} \fatsemi (p \to \mathbb{0} + p) \fatsemi \text{pu}_M \\
& = \left(\text{sw}^{\uparrow M} \fatsemi \text{ftn}_M \fatsemi \text{sw}^{\uparrow M} \fatsemi \text{ftn}_M\right)^{\uparrow F} \fatsemi (p \to \mathbb{0} + p) \fatsemi \text{pu}_M \quad .
\end{aligned}$$

Both sides are now equal.

To verify the inner interchange law, write its two sides separately:

$$\text{left-hand side}: \quad \text{ftn}_L \fatsemi \text{sw} = \left| \begin{array}{c|cc} & M^A & F^{M^{L^{M^A}}} \\ \hline M^A & \text{id} & \mathbb{0} \\ F^{M^{L^{M^A}}} & \mathbb{0} & \text{id} \\ F^{M^{L^{L^{M^A}}}} & \mathbb{0} & \overline{\text{ftn}}_L^{\uparrow M \uparrow F} \end{array} \right| \fatsemi \left\| \begin{array}{c} \text{pu}_L^{\uparrow M} \\ \overline{\text{sw}}^{\uparrow M \uparrow F} \fatsemi \text{ftn}_M^{\uparrow F} \fatsemi (p \to \mathbb{0} + p) \fatsemi \text{pu}_M \end{array} \right|$$

$$= \left| \begin{array}{c|c} & M^{A + (F \circ M \circ L \circ M)^A} \\ \hline M^A & \text{pu}_L^{\uparrow M} \\ F^{M^{L^{M^A}}} & \overline{\text{sw}}^{\uparrow M \uparrow F} \fatsemi \text{ftn}_M^{\uparrow F} \fatsemi (p \to \mathbb{0} + p) \fatsemi \text{pu}_M \\ F^{M^{L^{L^{M^A}}}} & \overline{\text{ftn}}_L^{\uparrow M \uparrow F} \fatsemi \overline{\text{sw}}^{\uparrow M \uparrow F} \fatsemi \text{ftn}_M^{\uparrow F} \fatsemi (p \to \mathbb{0} + p) \fatsemi \text{pu}_M \end{array} \right| ,$$

$$\text{right-hand side}: \quad \text{sw}^{\uparrow L} \fatsemi \text{sw} \fatsemi \text{ftn}_L^{\uparrow M} = \left| \begin{array}{c|cc} & M^{L^A} & F^{M^{L^{M^{L^A}}}} \\ \hline L^{M^A} & \text{sw} & \mathbb{0} \\ F^{M^{L^{L^{M^A}}}} & \mathbb{0} & \left(\text{sw}^{\overline{\uparrow L}}\right)^{\uparrow M \uparrow F} \end{array} \right| \fatsemi \text{sw} \fatsemi \text{ftn}_L^{\uparrow M} \quad .$$

In order to compare the resulting matrices, we need to expand the first row of the last matrix, so that the disjunctive type L^{M^A} is represented as $M^A + (F \circ M \circ L)^{M^A}$. We can then fill in the matrix

elements of `swap`:

$$\mathrm{sw}^{\uparrow L} = \left\| \begin{array}{c|cc} & M^{L^A} & F^{M^{L^{M^{L^A}}}} \\ \hline L^{M^A} & \mathrm{sw} & \mathbb{0} \\ F^{M^{L^{L^{M^A}}}} & \mathbb{0} & (\mathrm{sw}^{\overline{\uparrow L}})^{\uparrow M \uparrow F} \end{array} \right\|$$

$$= \left\| \begin{array}{c|cc} & M^{L^A} & F^{M^{L^{M^{L^A}}}} \\ \hline M^A & \mathrm{pu}_L^{\uparrow M} & \mathbb{0} \\ F^{M^{L^{M^A}}} & \overline{\mathrm{sw}}^{\uparrow M \uparrow F} ; \mathrm{ftn}_M^{\uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{pu}_M & \mathbb{0} \\ F^{M^{L^{L^{M^A}}}} & \mathbb{0} & (\mathrm{sw}^{\overline{\uparrow L}})^{\uparrow M \uparrow F} \end{array} \right\| \quad .$$

With this matrix, we continue rewriting the right-hand side of the inner interchange law:

$$\begin{aligned}
&\mathrm{sw}^{\uparrow L} ; \mathrm{sw} ; \mathrm{ftn}_L^{\uparrow M} \\
&= \left\| \begin{array}{cc} \mathrm{pu}_L^{\uparrow M} & \mathbb{0} \\ \overline{\mathrm{sw}}^{\uparrow M \uparrow F} ; \mathrm{ftn}_M^{\uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{pu}_M & \mathbb{0} \\ \mathbb{0} & (\mathrm{sw}^{\overline{\uparrow L}})^{\uparrow M \uparrow F} \end{array} \right\| ; \left\| \begin{array}{c} \mathrm{pu}_L^{\uparrow M} \\ \overline{\mathrm{sw}}^{\uparrow M \uparrow F} ; \mathrm{ftn}_M^{\uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{pu}_M \end{array} \right\| ; \mathrm{ftn}_L^{\uparrow M} \\
&= \left\| \begin{array}{c} \mathrm{pu}_L^{\uparrow M} ; \underline{\mathrm{pu}_L^{\uparrow M} ; \mathrm{ftn}_L^{\uparrow M}} \\ \overline{\mathrm{sw}}^{\uparrow M \uparrow F} ; \mathrm{ftn}_M^{\uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{pu}_M ; \underline{\mathrm{pu}_L^{\uparrow M} ; \mathrm{ftn}_L^{\uparrow M}} \\ \underline{(\mathrm{sw}^{\overline{\uparrow L}})^{\uparrow M \uparrow F} ; \overline{\mathrm{sw}}^{\uparrow M \uparrow F}} ; \mathrm{ftn}_M^{\uparrow F} ; (p \to \mathbb{0} + p) ; \underline{\mathrm{pu}_M ; \mathrm{ftn}_L^{\uparrow M}} \end{array} \right\| \\
&= \left\| \begin{array}{c} \mathrm{pu}_L^{\uparrow M} \\ \overline{\mathrm{sw}}^{\uparrow M \uparrow F} ; \mathrm{ftn}_M^{\uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{pu}_M \\ (\mathrm{sw}^{\overline{\uparrow L}} ; \overline{\mathrm{sw}})^{\uparrow M \uparrow F} ; \mathrm{ftn}_M^{\uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{ftn}_L ; \mathrm{pu}_M \end{array} \right\| \quad .
\end{aligned}$$

The remaining difference between the two sides of the law is only in the last rows of the matrices:

$$\overline{\mathrm{ftn}}_L^{\uparrow M \uparrow F} ; \overline{\mathrm{sw}}^{\uparrow M \uparrow F} ; \mathrm{ftn}_M^{\uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{pu}_M \stackrel{?}{=} (\mathrm{sw}^{\overline{\uparrow L}} ; \overline{\mathrm{sw}})^{\uparrow M \uparrow F} ; \mathrm{ftn}_M^{\uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{ftn}_L ; \mathrm{pu}_M \quad .$$

By the inductive assumption, the law already holds for the recursive call to `swap`:

$$\overline{\mathrm{ftn}}_L ; \overline{\mathrm{sw}} = \mathrm{sw}^{\uparrow L} ; \mathrm{sw} ; \mathrm{ftn}_L^{\uparrow M} \quad .$$

This gives the following code for the left-hand side:

$$\begin{aligned}
& \underline{\overline{\mathrm{ftn}}_L^{\uparrow M \uparrow F} ; \overline{\mathrm{sw}}^{\uparrow M \uparrow F}} ; \mathrm{ftn}_M^{\uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{pu}_M \\
\text{inductive assumption :} \quad & = (\mathrm{sw}^{\uparrow L} ; \mathrm{sw} ; \underline{\mathrm{ftn}_L^{\uparrow M}})^{\uparrow M \uparrow F} ; \mathrm{ftn}_M^{\uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{pu}_M \\
\text{naturality of } \mathrm{ftn}_M \text{ :} \quad & = (\mathrm{sw}^{\uparrow L} ; \mathrm{sw})^{\uparrow M \uparrow F} ; \mathrm{ftn}_M^{\uparrow F} ; \mathrm{ftn}_L^{\uparrow M \uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{pu}_M \quad .
\end{aligned}$$

The remaining difference between the two sides is:

$$\begin{aligned}
& (\mathrm{sw}^{\uparrow L} ; \mathrm{sw})^{\uparrow M \uparrow F} ; \mathrm{ftn}_M^{\uparrow F} ; \mathrm{ftn}_L^{\uparrow M \uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{pu}_M \\
& \stackrel{?}{=} (\mathrm{sw}^{\overline{\uparrow L}} ; \overline{\mathrm{sw}})^{\uparrow M \uparrow F} ; \mathrm{ftn}_M^{\uparrow F} ; (p \to \mathbb{0} + p) ; \mathrm{ftn}_L ; \mathrm{pu}_M \quad .
\end{aligned}$$

Both sides are functions of type $(F \circ M \circ L \circ L \circ M)^A \to M^{L^A}$. We will prove the equality above if we show that the following two functions of type $(F \circ M \circ L \circ L)^A \to L^A$ are equal:

$$(p \to \mathbb{0} + p) \,⨾\, \text{ftn}_L \stackrel{?}{=} \text{ftn}_L^{\uparrow M \uparrow F} \,⨾\, (p \to \mathbb{0} + p) \quad .$$

We compute the left-hand side and show that it equals the right-hand side:

$$(p \to \mathbb{0}^{:L^A} + p) \,⨾\, \text{ftn}_L = \left| \begin{array}{c||cc} & L^A & (F \circ M \circ L \circ L)^A \\ \hline (F \circ M \circ L \circ L)^A & \mathbb{0} & \text{id} \end{array} \right| ⨾ \left| \begin{array}{c||cc} & A & F^{M^{L^A}} \\ \hline A & \text{id} & \mathbb{0} \\ F^{M^{L^A}} & \mathbb{0} & \text{id} \\ F^{M^{L^{L^A}}} & \mathbb{0} & \overline{\text{ftn}}_L^{\uparrow M \uparrow F} \end{array} \right|$$

$$= \left| \begin{array}{c||ccc} & A & F^{M^{L^A}} & (F \circ M \circ L \circ L)^A \\ \hline (F \circ M \circ L \circ L)^A & \mathbb{0} & \mathbb{0} & \text{id} \end{array} \right| ⨾ \left| \begin{array}{c||cc} & A & F^{M^{L^A}} \\ \hline A & \text{id} & \mathbb{0} \\ F^{M^{L^A}} & \mathbb{0} & \text{id} \\ F^{M^{L^{L^A}}} & \mathbb{0} & \overline{\text{ftn}}_L^{\uparrow M \uparrow F} \end{array} \right|$$

$$= \left| \begin{array}{c||cc} & A & (F \circ M \circ L)^A \\ \hline (F \circ M \circ L \circ L)^A & \mathbb{0} & \overline{\text{ftn}}_L^{\uparrow M \uparrow F} \end{array} \right| = \overline{\text{ftn}}_L^{\uparrow M \uparrow F} \,⨾\, (p^{:(F \circ M \circ L)^A} \to \mathbb{0}^{:A} + p) \quad .$$

This concludes the proof of the four laws of `swap`. It follows that $M \circ L$ is a lawful monad.

Identity laws Setting $M = \text{Id}$ in the type of $T_{\text{Free}}^{M,A}$, we get the isomorphism $\text{Id} \circ \text{Free}^{F \circ \text{Id}} \cong \text{Free}^F$.

Lifting laws The function `flift` for T_{Free} is defined by:

$$\text{flift} : M^A \to M^{L^A} \quad , \qquad \text{flift} \stackrel{\text{def}}{=} \text{pu}_L^{\uparrow M} = (a^{:A} \to a + \mathbb{0}^{:(F \circ M \circ L)^A})^{\uparrow M} \quad .$$

To show that `flift` is a monad morphism, we refer to Statement 14.3.5.1, where we need to use `blift` instead of `flift` and exchange the monads L and M.

Implementation of runners Since the transformer $T_{\text{Free}} = M \circ L$ is not purely compositional (the monad L itself depends on M), mapping the monad M to M' via a monad morphism $\phi : M \rightsquigarrow M'$ requires us also to map L to $L' \stackrel{\text{def}}{=} \text{Free}^{F \circ M'}$. Denote that map temporarily by ψ; it is a function of ϕ, but we will write simply ψ rather than $\psi(\phi)$ for brevity, since ϕ is fixed throughout the derivation. We can implement $\psi : L^A \to L'^A$ by the following code matrix:

$$\psi : \text{Free}^{F \circ M, A} \to \text{Free}^{F \circ M', A} \quad , \qquad \psi \stackrel{\text{def}}{=} \left| \begin{array}{c||cc} & A & (F \circ M' \circ L')^A \\ \hline A & \text{id} & \mathbb{0} \\ (F \circ M \circ L)^A & \mathbb{0} & \overline{\psi}^{\uparrow M \uparrow F} \,⨾\, \phi^{\uparrow F} \end{array} \right| \quad . \tag{14.66}$$

Since ϕ is a natural transformation, Exercise 7.2.5.1(a) indicates that the map $\phi^{\uparrow F} : F^{M^A} \rightsquigarrow F^{M'^A}$ is also a natural transformation, and then Statement 13.6.1.1 shows that $\psi : \text{Free}^{F \circ M} \rightsquigarrow \text{Free}^{F \circ M'}$ is a monad morphism. The foreign runner $\text{frun}^{M,M'}(\phi) : T_{\text{Free}} \rightsquigarrow T'_{\text{Free}}$ is then implemented by:

$$\text{frun}^{M,M'} : (M \rightsquigarrow M') \to M^{L^A} \to M'^{L'^A} \quad , \quad \text{frun}^{M,M'}(\phi^{:M \rightsquigarrow M'}) \stackrel{\text{def}}{=} \psi^{\uparrow M} \,⨾\, \phi \quad .$$

To implement the base runner, we use the similarity between the two disjunctive types:

$$L^A \stackrel{\text{def}}{=} \text{Free}^{F \circ M, A} = A + F^{M^{\text{Free}^{F \circ M, A}}} = A + (F \circ M \circ L)^A \quad \text{and} \quad \text{Free}^{F, M^A} = M^A + F^{\text{Free}^{F, M^A}} \quad .$$

The code for the base runner is:

$$\text{brun} : (\text{Free}^F \rightsquigarrow \text{Id}) \to M^{L^A} \to M^A \quad ,$$

$$\text{brun}\,(\theta^{:\text{Free}^F \rightsquigarrow \text{Id}}) \stackrel{\text{def}}{=} \left|\begin{array}{c|c} & M^A \quad F^{\text{Free}^{F,M^A}} \\ \hline A & \text{pu}_M \quad \mathbb{0} \\ (F\circ M\circ L)^A & \mathbb{0} \quad (\overline{\text{brun}}\,(\theta) \,;\, (x^{:M^A} \to x + \mathbb{0}^{:F^{\text{Free}^{F,M^A}}}))^{\uparrow F} \end{array}\right|^{\uparrow M} \,;\, \theta^{\uparrow M} \,;\, \text{ftn}_M \quad .$$

Monadic naturality of `swap` When the monad M is replaced by another monad M' via a monad morphism $\phi : M \rightsquigarrow M'$, the `swap` function with type signature $\text{sw}_{M,L} : L\circ M \rightsquigarrow M\circ L$ is replaced by the corresponding function $\text{sw}_{M',L'} : L'\circ M' \rightsquigarrow M'\circ L'$. The function $\text{sw}_{M',L'}$ has the same code as $\text{sw}_{M,L}$ except for using the monad parameter M' instead of M. This monadic naturality property is expressed by the law analogous to Eq. (14.60):

$$\text{sw}_{M,L} \,;\, \psi^{\uparrow M} \,;\, \phi = \phi^{\uparrow L} \,;\, \psi \,;\, \text{sw}_{M',L'} \quad . \tag{14.67}$$

To verify this law, compute both sides of Eq. (14.67) separately. Both sides are functions of type $L^{M^A} \to M'^{L'^A}$:

$$\text{left-hand side}: \quad \text{sw}_{M,L} \,;\, \psi^{\uparrow M} \,;\, \phi = \left|\begin{array}{c|c} & M'^{A+(F\circ M'\circ L')^A} \\ \hline M^A & \underline{\text{pu}_L^{\uparrow M} \,;\, \psi^{\uparrow M}} \,;\, \phi \\ F^{M^{L^{M^A}}} & \overline{\text{sw}}_{M,L}^{\uparrow M\uparrow F} \,;\, \text{ftn}_M^{\uparrow F} \,;\, (p \to \mathbb{0}^{:A} + p) \,;\, \underline{\text{pu}_M \,;\, \psi^{\uparrow M}} \,;\, \phi \end{array}\right|$$

$$= \left|\begin{array}{c|c} & M'^{A+(F\circ M'\circ L')^A} \\ \hline M^A & \text{pu}_{L'}^{\uparrow M} \,;\, \phi \\ F^{M^{L^{M^A}}} & \overline{\text{sw}}_{M,L}^{\uparrow M\uparrow F} \,;\, \text{ftn}_M^{\uparrow F} \,;\, (p \to \mathbb{0}^{:A} + p) \,;\, \psi \,;\, \text{pu}_M \,;\, \phi \end{array}\right| \quad ,$$

$$\text{right-hand side}: \quad \phi^{\uparrow L} \,;\, \psi \,;\, \text{sw}_{M',L'}$$

$$= \left|\begin{array}{c|cc} & M'^A & F^{M^{L^{M'^A}}} \\ \hline M^A & \phi & \mathbb{0} \\ F^{M^{L^{M^A}}} & \mathbb{0} & (\phi^{\overline{\uparrow L}})^{\uparrow M\uparrow F} \end{array}\right| \,;\, \left|\begin{array}{c|cc} & M'^A & F^{M'^{L'^{M'^A}}} \\ \hline M'^A & \text{id} & \mathbb{0} \\ F^{M^{L^{M'^A}}} & \mathbb{0} & \overline{\psi}^{\uparrow M\uparrow F} \,;\, \phi^{\uparrow F} \end{array}\right| \,;\, \text{sw}_{M',L'}$$

$$= \left|\begin{array}{c|cc} & M'^A & F^{M'^{L'^{M'^A}}} \\ \hline M^A & \phi & \mathbb{0} \\ F^{M^{L^{M^A}}} & \mathbb{0} & (\phi^{\overline{\uparrow L}})^{\uparrow M\uparrow F} \,;\, \overline{\psi}^{\uparrow M\uparrow F} \,;\, \phi^{\uparrow F} \end{array}\right| \,;\, \left|\begin{array}{c|c} & M'^{A+(F\circ M'\circ L')^A} \\ \hline M'^A & \text{pu}_{L'}^{\uparrow M'} \\ F^{M'^{L'^{M'^A}}} & \overline{\text{sw}}_{M',L'}^{\uparrow M'\uparrow F} \,;\, \text{ftn}_{M'}^{\uparrow F} \,;\, (p \to \mathbb{0}^{:A} + p) \,;\, \text{pu}_{M'} \end{array}\right|$$

$$= \left|\begin{array}{c|c} & M'^{A+(F\circ M'\circ L')^A} \\ \hline M^A & \phi \,;\, \text{pu}_{L'}^{\uparrow M'} \\ F^{M^{L^{M^A}}} & (\phi^{\overline{\uparrow L}})^{\uparrow M\uparrow F} \,;\, \overline{\psi}^{\uparrow M\uparrow F} \,;\, \phi^{\uparrow F} \,;\, \overline{\text{sw}}_{M',L'}^{\uparrow M'\uparrow F} \,;\, \text{ftn}_{M'}^{\uparrow F} \,;\, (p \to \mathbb{0}^{:A} + p) \,;\, \text{pu}_{M'} \end{array}\right| \quad .$$

The first rows of the two matrices are equal due to naturality of ϕ:

$$\text{pu}_{L'}^{\uparrow M} \,;\, \phi = \phi \,;\, \text{pu}_{L'}^{\uparrow M'} \quad .$$

The remaining difference is in the second rows:

$$\begin{aligned} &\overline{\text{sw}}_{M,L}^{\uparrow M\uparrow F} \,;\, \text{ftn}_M^{\uparrow F} \,;\, (p \to \mathbb{0}^{:A} + p) \,;\, \psi \,;\, \text{pu}_M \,;\, \phi \\ &\stackrel{?}{=} (\phi^{\overline{\uparrow L}})^{\uparrow M\uparrow F} \,;\, \overline{\psi}^{\uparrow M\uparrow F} \,;\, \phi^{\uparrow F} \,;\, \overline{\text{sw}}_{M',L'}^{\uparrow M'\uparrow F} \,;\, \text{ftn}_{M'}^{\uparrow F} \,;\, (p \to \mathbb{0}^{:A} + p) \,;\, \text{pu}_{M'} \quad . \end{aligned} \tag{14.68}$$

Transform the left-hand side of Eq. (14.68):

$$
\begin{aligned}
&\overline{\mathrm{sw}}^{\uparrow M \uparrow F}_{M,L} \fatsemi \mathrm{ftn}^{\uparrow F}_{M} \fatsemi (p \to \mathbb{0} + p) \fatsemi \psi \fatsemi \underline{\mathrm{pu}_M \fatsemi \phi} \\
&= \overline{\mathrm{sw}}^{\uparrow M \uparrow F}_{M,L} \fatsemi \mathrm{ftn}^{\uparrow F}_{M} \fatsemi \left\| \begin{array}{cc} \mathbb{0} & \mathrm{id} \end{array} \right| \fatsemi \left\| \begin{array}{cc} \mathrm{id} & \mathbb{0} \\ \mathbb{0} & \overline{\psi}^{\uparrow M \uparrow F} \fatsemi \phi^{\uparrow F} \end{array} \right| \fatsemi \mathrm{pu}_{M'} \\
&= \overline{\mathrm{sw}}^{\uparrow M \uparrow F}_{M,L} \fatsemi \mathrm{ftn}^{\uparrow F}_{M} \fatsemi \underline{\left\| \begin{array}{cc} \mathbb{0} & \overline{\psi}^{\uparrow M \uparrow F} \fatsemi \phi^{\uparrow F} \end{array} \right|} \fatsemi \mathrm{pu}_{M'} \\
&= \overline{\mathrm{sw}}^{\uparrow M \uparrow F}_{M,L} \fatsemi \mathrm{ftn}^{\uparrow F}_{M} \fatsemi \underline{\overline{\psi}^{\uparrow M \uparrow F} \fatsemi \phi^{\uparrow F} \fatsemi (p \to \mathbb{0} + p)} \fatsemi \mathrm{pu}_{M'} \quad .
\end{aligned}
$$

The remaining difference in Eq. (14.68) is:

$$
\overline{\mathrm{sw}}^{\uparrow M \uparrow F}_{M,L} \fatsemi \mathrm{ftn}^{\uparrow F}_{M} \fatsemi \overline{\psi}^{\uparrow M \uparrow F} \fatsemi \phi^{\uparrow F} \stackrel{?}{=} \left(\phi^{\overline{\uparrow L}}\right)^{\uparrow M \uparrow F} \fatsemi \overline{\psi}^{\uparrow M \uparrow F} \fatsemi \phi^{\uparrow F} \fatsemi \overline{\mathrm{sw}}^{\uparrow M' \uparrow F}_{M',L'} \fatsemi \mathrm{ftn}^{\uparrow F}_{M'} \quad ,
$$

which we can simplify by omitting the common lifting $^{\uparrow F}$:

$$
\overline{\mathrm{sw}}^{\uparrow M}_{M,L} \fatsemi \mathrm{ftn}_{M} \fatsemi \overline{\psi}^{\uparrow M} \fatsemi \phi \stackrel{?}{=} \left(\phi^{\overline{\uparrow L}}\right)^{\uparrow M} \fatsemi \overline{\psi}^{\uparrow M} \fatsemi \phi \fatsemi \overline{\mathrm{sw}}^{\uparrow M'}_{M',L'} \fatsemi \mathrm{ftn}_{M'} \quad . \tag{14.69}
$$

To transform the right-hand side of Eq. (14.69), we need the inductive assumption that Eq. (14.67) already holds for $\overline{\mathrm{sw}}$. To use that assumption, we need to move $\overline{\mathrm{sw}}$ next to $\phi^{\uparrow L} \fatsemi \psi$ under $^{\uparrow M}$:

$$
\begin{aligned}
& \left(\phi^{\overline{\uparrow L}}\right)^{\uparrow M} \fatsemi \overline{\psi}^{\uparrow M} \fatsemi \underline{\phi \fatsemi \overline{\mathrm{sw}}^{\uparrow M'}_{M',L'}} \fatsemi \mathrm{ftn}_{M'} \\
\text{naturality of } \phi : \quad & = \left(\phi^{\overline{\uparrow L}} \fatsemi \overline{\psi} \fatsemi \overline{\mathrm{sw}}_{M',L'}\right)^{\uparrow M} \fatsemi \phi \fatsemi \mathrm{ftn}_{M'} \\
\text{inductive assumption, Eq. (14.67) for } \overline{\mathrm{sw}} : \quad & = \underline{\left(\mathrm{sw}_{M,L} \fatsemi \psi^{\uparrow M} \fatsemi \phi\right)^{\uparrow M} \fatsemi \phi \fatsemi \mathrm{ftn}_{M'}} \\
\text{composition law of } \phi : \quad & = \mathrm{sw}^{\uparrow M}_{M,L} \fatsemi \underline{\psi^{\uparrow M \uparrow M} \fatsemi \mathrm{ftn}_M} \fatsemi \phi \\
\text{naturality of } \mathrm{ftn}_M : \quad & = \mathrm{sw}^{\uparrow M}_{M,L} \fatsemi \mathrm{ftn}_M \fatsemi \psi^{\uparrow M} \fatsemi \phi \quad .
\end{aligned}
$$

The two sides of Eq. (14.69) are now equal, which completes the proof of the law.

Runner laws To show that $\mathrm{frun}^{M,M'}(\phi)$ is a monad morphism $T \rightsquigarrow T'$, first check the identity law:

$$
\begin{aligned}
\text{expect to equal } \mathrm{pu}_{T'} : \quad & \mathrm{pu}_T \fatsemi \mathrm{frun}^{M,M'}(\phi) = \mathrm{pu}_L \fatsemi \underline{\mathrm{pu}_M \fatsemi \psi^{\uparrow M}} \fatsemi \phi \\
\text{naturality of } \mathrm{pu}_M : \quad & = \underline{\mathrm{pu}_L \fatsemi \psi} \fatsemi \underline{\mathrm{pu}_M \fatsemi \phi} \\
\text{identity laws of } \phi, \psi : \quad & = \mathrm{pu}_{L'} \fatsemi \mathrm{pu}_{M'} = \mathrm{pu}_{T'} \quad .
\end{aligned}
$$

To check the composition law, write the two sides separately:

$$
\begin{aligned}
\text{left-hand side} : \quad & \mathrm{ftn}_T \fatsemi \mathrm{frun}\,(\phi) = \mathrm{sw}^{\uparrow M} \fatsemi \mathrm{ftn}_M \fatsemi \mathrm{ftn}^{\uparrow M}_{L} \fatsemi \psi^{\uparrow M} \fatsemi \phi \quad , \\
\text{right-hand side} : \quad & (\mathrm{frun}\,(\phi))^{\uparrow T} \fatsemi \mathrm{frun}\,(\phi) \fatsemi \mathrm{ftn}_{T'} = (\psi^{\uparrow M} \fatsemi \phi)^{\uparrow L \uparrow M} \fatsemi \psi^{\uparrow M} \fatsemi \phi \fatsemi \mathrm{sw}^{\uparrow M'}_{M',L'} \fatsemi \mathrm{ftn}_{M'} \fatsemi \mathrm{ftn}^{\uparrow M'}_{L'} \quad .
\end{aligned}
$$

We begin with the right-hand side since it is more complicated. In order to show that it is equal to the left-hand side, we need somehow to move `swap` to the left in the function compositions, so that we may use Eq. (14.67):

$$
\begin{aligned}
\text{right-hand side} : \quad & (\psi^{\uparrow M} \fatsemi \phi)^{\uparrow L \uparrow M} \fatsemi \psi^{\uparrow M} \fatsemi \underline{\phi \fatsemi \mathrm{sw}^{\uparrow M'}_{M',L'}} \fatsemi \mathrm{ftn}_{M'} \fatsemi \mathrm{ftn}^{\uparrow M'}_{L'} \\
\text{naturality of } \phi : \quad & = \left(\underline{(\psi^{\uparrow M} \fatsemi \phi)^{\uparrow L} \fatsemi \psi \fatsemi \mathrm{sw}_{M',L'}}\right)^{\uparrow M} \fatsemi \phi \fatsemi \mathrm{ftn}_{M'} \fatsemi \mathrm{ftn}^{\uparrow M'}_{L'} \\
\text{use Eq. (14.67) under } ^{\uparrow M} : \quad & = \left(\underline{\psi^{\uparrow M \uparrow L} \fatsemi \mathrm{sw}_{M,L}} \fatsemi \psi^{\uparrow M} \fatsemi \phi\right)^{\uparrow M} \fatsemi \phi \fatsemi \mathrm{ftn}_{M'} \fatsemi \mathrm{ftn}^{\uparrow M'}_{L'} \\
\text{composition law of } \phi : \quad & = \mathrm{sw}_{M,L} \fatsemi \underline{\left(\psi^{\uparrow L} \fatsemi \psi\right)^{\uparrow M \uparrow M} \fatsemi \mathrm{ftn}_M} \fatsemi \underline{\phi \fatsemi \mathrm{ftn}^{\uparrow M'}_{L'}} \\
\text{naturality of } \mathrm{ftn}_M \text{ and } \phi : \quad & = \mathrm{sw}_{M,L} \fatsemi \mathrm{ftn}_M \fatsemi \underline{\left(\psi^{\uparrow L} \fatsemi \psi\right)^{\uparrow M} \fatsemi \mathrm{ftn}^{\uparrow M}_{L'}} \fatsemi \phi \\
\text{composition law of } \psi : \quad & = \mathrm{sw}_{M,L} \fatsemi \mathrm{ftn}_M \fatsemi \left(\mathrm{ftn}_L \fatsemi \psi\right)^{\uparrow M} \fatsemi \phi \quad .
\end{aligned}
$$

The last line equals the left-hand side of the law.

Functor laws of `frun` To verify the functor laws of frun, begin with the identity law. If $\phi = \text{id}$, Equation (14.66) defines the corresponding function ψ also as $\psi = \text{id}$, and so the identity law holds:

$$\text{expect to equal id}: \quad \text{frun}\,(\text{id}) = \psi^{\uparrow M} \fatsemi \phi = \text{id}^{\uparrow M} \fatsemi \text{id} = \text{id} \quad .$$

To verify the composition law, take any monad morphisms $\phi_1 : Q \rightsquigarrow R$ and $\phi_2 : R \rightsquigarrow S$, where Q, R, S are some foreign monads. Denote temporarily $L_Q \stackrel{\text{def}}{=} \text{Free}^{F\circ Q}$, $L_R \stackrel{\text{def}}{=} \text{Free}^{F\circ R}$, $L_S \stackrel{\text{def}}{=} \text{Free}^{F\circ S}$. Then denote the corresponding monad morphisms by $\psi_1 : L_Q \rightsquigarrow L_R$ and $\psi_2 : L_R \rightsquigarrow L_S$ and write:

$$\begin{aligned}
\text{expect to equal frun}\,(\phi_1 \fatsemi \phi_2): \quad & \text{frun}\,(\phi_1) \fatsemi \text{frun}(\phi_2) = \psi_1^{\uparrow Q} \fatsemi \underline{\phi_1 \fatsemi \psi_2^{\uparrow R}} \fatsemi \phi_2 \\
\text{naturality of } \phi_1: \quad & = \psi_1^{\uparrow Q} \fatsemi \psi_2^{\uparrow Q} \fatsemi \phi_1 \fatsemi \phi_2 = (\psi_1 \fatsemi \psi_2)^{\uparrow Q} \fatsemi (\phi_1 \fatsemi \phi_2) \quad .
\end{aligned}$$

It remains to show that $\psi_1 \fatsemi \psi_2$ is the monad morphism $L_Q \rightsquigarrow L_S$ corresponding to the composition $\phi_1 \fatsemi \phi_2$. By Eq. (14.66), we have:

$$\psi_1 \fatsemi \psi_2 = \left|\begin{array}{c|c|c}
 & A & (F\circ R\circ L_R)^A \\ \hline
A & \text{id} & \mathbb{0} \\
(F\circ Q\circ L_Q)^A & \mathbb{0} & \overline{\psi}_1^{\uparrow Q\uparrow F} \fatsemi \phi_1^{\uparrow F}
\end{array}\right| \fatsemi \left|\begin{array}{c|c|c}
 & A & (F\circ S\circ L_S)^A \\ \hline
A & \text{id} & \mathbb{0} \\
(F\circ R\circ L_R)^A & \mathbb{0} & \overline{\psi}_2^{\uparrow R\uparrow F} \fatsemi \phi_2^{\uparrow F}
\end{array}\right|$$

$$= \left|\begin{array}{c|c|c}
 & A & (F\circ S\circ L_S)^A \\ \hline
A & \text{id} & \mathbb{0} \\
(F\circ Q\circ L_Q)^A & \mathbb{0} & \overline{\psi}_1^{\uparrow Q\uparrow F} \fatsemi \phi_1^{\uparrow F} \fatsemi \overline{\psi}_2^{\uparrow R\uparrow F} \fatsemi \phi_2^{\uparrow F}
\end{array}\right| \quad .$$

We need to show that this equals the code matrix given by Eq. (14.66) for $\phi_1 \fatsemi \phi_2$:

$$\psi_1 \fatsemi \psi_2 \stackrel{?}{=} \psi \stackrel{\text{def}}{=} \left|\begin{array}{c|c|c}
 & A & (F\circ S\circ L_S)^A \\ \hline
A & \text{id} & \mathbb{0} \\
(F\circ Q\circ L_Q)^A & \mathbb{0} & \overline{\psi}^{\uparrow Q\uparrow F} \fatsemi (\phi_1 \fatsemi \phi_2)^{\uparrow F}
\end{array}\right| \quad .$$

Since ψ is a recursive function, we may use the inductive assumption that $\overline{\psi} = \overline{\psi}_1 \fatsemi \overline{\psi}_2$ for the recursive calls. The difference between the matrices is then reduced to:

$$\overline{\psi}_1^{\uparrow Q\uparrow F} \fatsemi \phi_1^{\uparrow F} \fatsemi \overline{\psi}_2^{\uparrow R\uparrow F} \fatsemi \phi_2^{\uparrow F} \stackrel{?}{=} (\psi_1 \fatsemi \psi_2)^{\uparrow Q\uparrow F} \fatsemi (\phi_1 \fatsemi \phi_2)^{\uparrow F} \quad .$$

The last equality holds since $\phi_1 \fatsemi \overline{\psi}_2^{\uparrow R} = \overline{\psi}_2^{\uparrow Q} \fatsemi \phi_1$ due to naturality of ϕ_1. This proves the law.

Base runner laws The base runner needs to satisfy the monad morphism laws and the nondegeneracy law. Let us temporarily denote by K the base monad, $K \stackrel{\text{def}}{=} \text{Free}^F$, and by ξ the function:

$$\xi : L^A \to K^{M^A} \quad , \qquad \xi \stackrel{\text{def}}{=} \left|\begin{array}{c|c|c}
 & M^A & F^{K^{M^A}} \\ \hline
A & \text{pu}_M & \mathbb{0} \\
F^{M^{L^A}} & \mathbb{0} & (\overline{\text{brun}}\,(\theta) \fatsemi (x^{:M^A} \to x + \mathbb{0}^{:F^{K^{M^A}}}))^{\uparrow F}
\end{array}\right| \quad .$$

We can then write the code for brun (θ) more conveniently:

$$\text{brun}\,(\theta) \stackrel{\text{def}}{=} \xi^{\uparrow M} \fatsemi \theta^{\uparrow M} \fatsemi \text{ftn}_M \quad .$$

To show that brun (θ) is a monad morphism, we will use Statement 14.6.2.2 with the monads M, L instead of M, N. The required monad morphism $\phi : L \rightsquigarrow M$ is defined as $\phi \stackrel{\text{def}}{=} \xi \fatsemi \theta$. It remains to verify that ϕ is indeed a monad morphism and that `swap` satisfies the special law with respect to ϕ.

Although $\theta : K \rightsquigarrow \mathrm{Id}$ is an arbitrary runner, its behavior is constrained by the laws of monad morphisms. To extract some useful information about θ, begin with its identity law:

$$\mathrm{pu}_K \,⨾\, \theta = \mathrm{id} \quad , \qquad \text{or equivalently:} \qquad (x \to x + \mathbb{0}) \,⨾\, \theta = \mathrm{id} \quad .$$

If we apply θ to a value of type $\mathbb{0} + F^{A+\mathbb{0}}$, the result must be of type A. Since the type $\mathbb{0} + F^{A+\mathbb{0}}$ is equivalent to F^A, we have obtained a transformation $r : F^A \to A$ that extracts values from F-wrappers. We can write this property as a definition of the function r through θ:

$$r : F^A \to A \quad , \qquad r \stackrel{\text{def}}{=} (x^{:A} \to x + \mathbb{0}^{:F^{K^A}})^{\uparrow F} \,⨾\, (y^{:F^{A+\mathbb{0}}} \to \mathbb{0}^{:A} + y) \,⨾\, \theta \quad .$$

Since θ is a natural transformation (because it is a monad morphism), r is also a natural transformation. In the rest of this derivation, we will only need to use the naturality property of r; we will not need to know the code of r.

With these equations, we can simplify the function $\phi \stackrel{\text{def}}{=} \xi \,⨾\, \theta$ by first rewriting the definition of ϕ in a way that does not explicitly depend on $\mathrm{brun}\,(\theta)$. Since $\mathrm{brun}\,(\theta) = \phi^{\uparrow M} \,⨾\, \mathrm{ftn}_M$, we have:

$$\phi \stackrel{\text{def}}{=} \xi \,⨾\, \theta = \left| \begin{array}{c||cc} & M^A & F^{K^{M^A}} \\ \hline A & \mathrm{pu}_M & \mathbb{0} \\ F^{M^{L^A}} & \mathbb{0} & (\overline{\phi}^{\uparrow M} \,⨾\, \mathrm{ftn}_M \,⨾\, (x^{:M^A} \to x + \mathbb{0}^{:F^{K^{M^A}}}))^{\uparrow F} \end{array} \right| ⨾\, \theta \quad .$$

Now we may merge the matrix columns of ξ and compute the composition with θ:

$$\phi = \xi \,⨾\, \theta = \left| \begin{array}{c||c} & M^A + F^{K^{M^A}} \\ \hline A & \mathrm{pu}_M \,⨾\, (x \to x + \mathbb{0}) \\ F^{M^{L^A}} & (\overline{\phi}^{\uparrow M} \,⨾\, \mathrm{ftn}_M \,⨾\, (x \to x + \mathbb{0}))^{\uparrow F} \,⨾\, (x \to \mathbb{0} + x) \end{array} \right| ⨾\, \theta$$

$$= \left| \begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \,⨾\, \underline{(x \to x + \mathbb{0}) \,⨾\, \theta} \\ F^{M^{L^A}} & (\overline{\phi}^{\uparrow M} \,⨾\, \mathrm{ftn}_M \,⨾\, \underline{(x \to x + \mathbb{0}))^{\uparrow F} \,⨾\, (x \to \mathbb{0} + x) \,⨾\, \theta} \end{array} \right| = \left| \begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \\ F^{M^{L^A}} & \overline{\phi}^{\uparrow M \uparrow F} \,⨾\, \mathrm{ftn}_M^{\uparrow F} \,⨾\, r \end{array} \right| \quad . \tag{14.70}$$

With the simplified code for ϕ, we proceed with the proof. To verify the identity law of ϕ:

$$\text{expect to equal } \mathrm{pu}_M : \quad \mathrm{pu}_L \,⨾\, \phi = \left\| \begin{array}{cc} \mathrm{id} & \mathbb{0} \end{array} \right| ⨾ \left\| \begin{array}{c} \mathrm{pu}_M \\ \overline{\phi}^{\uparrow M \uparrow F} \,⨾\, \mathrm{ftn}_M^{\uparrow F} \,⨾\, r \end{array} \right| = \mathrm{pu}_M \quad .$$

To verify the composition law of ϕ, begin with the left-hand side of that law:

$$\text{left-hand side :} \quad \mathrm{ftn}_L \,⨾\, \phi = \left| \begin{array}{c||cc} & A & F^{M^{L^A}} \\ \hline A & \mathrm{id} & \mathbb{0} \\ F^{M^{L^A}} & \mathbb{0} & \mathrm{id} \\ F^{M^{L^{L^A}}} & \mathbb{0} & \overline{\mathrm{ftn}}_L^{\uparrow M \uparrow F} \end{array} \right| ⨾ \left| \begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \\ F^{M^{L^A}} & \overline{\phi}^{\uparrow M \uparrow F} \,⨾\, \mathrm{ftn}_M^{\uparrow F} \,⨾\, r \end{array} \right|$$

$$= \left| \begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \\ F^{M^{L^A}} & \overline{\phi}^{\uparrow M \uparrow F} \,⨾\, \mathrm{ftn}_M^{\uparrow F} \,⨾\, r \\ F^{M^{L^{L^A}}} & \overline{\mathrm{ftn}}_L^{\uparrow M \uparrow F} \,⨾\, \overline{\phi}^{\uparrow M \uparrow F} \,⨾\, \mathrm{ftn}_M^{\uparrow F} \,⨾\, r \end{array} \right| \quad .$$

It is easier to work with the right-hand side of the law if we first simplify the function $\phi ⨾ \text{ftn}_M$:

$$\phi ⨾ \text{ftn}_M = \left|\begin{array}{c||c} & M^{M^A} \\ \hline M^A & \text{pu}_M \\ F^{M^{L^{M^A}}} & \overline{\phi}^{\uparrow M \uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ r \end{array}\right| ⨾ \text{ftn}_M = \left|\begin{array}{c||c} & M^A \\ \hline M^A & \underline{\text{pu}_M ⨾ \text{ftn}_M} \\ F^{M^{L^{M^A}}} & \overline{\phi}^{\uparrow M \uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ r ⨾ \underline{\text{ftn}_M} \end{array}\right| .$$

$$= \left|\begin{array}{c||c} & M^A \\ \hline M^A & \text{id} \\ F^{M^{L^{M^A}}} & (\overline{\phi}^{\uparrow M} ⨾ \text{ftn}_M ⨾ \text{ftn}_M)^{\uparrow F} ⨾ r \end{array}\right| .$$

Now we write the right-hand side of the composition law:

right-hand side : $\phi^{\uparrow L} ⨾ \phi ⨾ \text{ftn}_M$

$$= \left|\begin{array}{c||cc} & M^A & F^{M^{L^{M^A}}} \\ \hline A & \text{pu}_M & \mathbb{0} \\ F^{M^{L^A}} & \overline{\phi}^{\uparrow M \uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ r & \mathbb{0} \\ F^{M^{L^{L^A}}} & \mathbb{0} & \overline{\phi}^{\uparrow L \uparrow M \uparrow F} \end{array}\right| ⨾ \left|\begin{array}{c||c} & M^A \\ \hline M^A & \text{id} \\ F^{M^{L^{M^A}}} & (\overline{\phi}^{\uparrow M} ⨾ \text{ftn}_M ⨾ \text{ftn}_M)^{\uparrow F} ⨾ r \end{array}\right|$$

$$= \left|\begin{array}{c||c} & M^A \\ \hline A & \text{pu}_M \\ F^{M^{L^A}} & \overline{\phi}^{\uparrow M \uparrow F} ⨾ \text{ftn}_M^{\uparrow F} ⨾ r \\ F^{M^{L^{L^A}}} & (\overline{\phi}^{\uparrow L \uparrow M} ⨾ \overline{\phi}^{\uparrow M} ⨾ \text{ftn}_M ⨾ \text{ftn}_M)^{\uparrow F} ⨾ r \end{array}\right| .$$

The remaining difference is in the last rows of the matrices:

$$(\overline{\text{ftn}_L}^{\uparrow M} ⨾ \overline{\phi}^{\uparrow M} ⨾ \text{ftn}_M)^{\uparrow F} ⨾ r \stackrel{?}{=} (\overline{\phi}^{\uparrow L \uparrow M} ⨾ \overline{\phi}^{\uparrow M} ⨾ \text{ftn}_M ⨾ \text{ftn}_M)^{\uparrow F} ⨾ r \quad .$$

We will prove this equality if we show that:

$$\overline{\text{ftn}_L}^{\uparrow M} ⨾ \overline{\phi}^{\uparrow M} ⨾ \text{ftn}_M \stackrel{?}{=} \overline{\phi}^{\uparrow L \uparrow M} ⨾ \overline{\phi}^{\uparrow M} ⨾ \text{ftn}_M ⨾ \text{ftn}_M \quad .$$

Substitute the inductive assumption for the recursive calls to $\overline{\phi}$:

$$\text{ftn}_L ⨾ \overline{\phi} = \overline{\phi}^{\uparrow L} ⨾ \overline{\phi} ⨾ \text{ftn}_M \quad .$$

Then we only need to show the following equality:

$$\underline{\overline{\text{ftn}_L}^{\uparrow M} ⨾ \overline{\phi}^{\uparrow M}} ⨾ \text{ftn}_M = (\overline{\phi}^{\uparrow L} ⨾ \overline{\phi} ⨾ \text{ftn}_M)^{\uparrow M} ⨾ \text{ftn}_M$$
$$\stackrel{?}{=} \overline{\phi}^{\uparrow L \uparrow M} ⨾ \overline{\phi}^{\uparrow M} ⨾ \underline{\text{ftn}_M ⨾ \text{ftn}_M} = \overline{\phi}^{\uparrow L \uparrow M} ⨾ \overline{\phi}^{\uparrow M} ⨾ \text{ftn}_M^{\uparrow M} ⨾ \text{ftn}_M \quad .$$

The last expressions are now equal, which completes the derivation.

It remains to check the special law of `swap`:

$$\text{sw} ⨾ \phi^{\uparrow M} ⨾ \text{ftn}_M \stackrel{?}{=} \phi ⨾ \text{ftn}_M \quad . \tag{14.71}$$

Substitute the code of `swap` from Eq. (14.64) into the left-hand side:

$$\mathrm{sw} ⨾ \phi^{\uparrow M} ⨾ \mathrm{ftn}_M = \left|\begin{array}{c||c} & M^A \\ \hline M^A & \underline{\mathrm{pu}_L^{\uparrow M} ⨾ \phi^{\uparrow M}} ⨾ \mathrm{ftn}_M \\ F^{M^{L^{M^A}}} & \overline{\mathrm{sw}}^{\uparrow M \uparrow F} ⨾ \mathrm{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \underline{\mathrm{pu}_M ⨾ \phi^{\uparrow M}} ⨾ \mathrm{ftn}_M \end{array}\right|$$

$$= \left|\begin{array}{c||c} & M^A \\ \hline M^A & \underline{\mathrm{pu}_M^{\uparrow M} ⨾ \mathrm{ftn}_M} \\ F^{M^{L^{M^A}}} & \overline{\mathrm{sw}}^{\uparrow M \uparrow F} ⨾ \mathrm{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \phi ⨾ \underline{\mathrm{pu}_M ⨾ \mathrm{ftn}_M} \end{array}\right|$$

$$= \left|\begin{array}{c||c} & M^A \\ \hline M^A & \mathrm{id} \\ F^{M^{L^{M^A}}} & \overline{\mathrm{sw}}^{\uparrow M \uparrow F} ⨾ \mathrm{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \phi \end{array}\right| \quad .$$

We can further simplify the last matrix row if we substitute the code for ϕ from Eq. (14.70):

$$\begin{aligned}
&\overline{\mathrm{sw}}^{\uparrow M \uparrow F} ⨾ \mathrm{ftn}_M^{\uparrow F} ⨾ (p \to \mathbb{0}^{:A} + p) ⨾ \phi \\
&= \overline{\mathrm{sw}}^{\uparrow M \uparrow F} ⨾ \mathrm{ftn}_M^{\uparrow F} ⨾ \left\| \mathbb{0} \quad \mathrm{id} \right| ⨾ \left|\begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \\ F^{M^{L^A}} & \overline{\phi}^{\uparrow M \uparrow F} ⨾ \mathrm{ftn}_M^{\uparrow F} ⨾ r \end{array}\right| \\
&= \overline{\mathrm{sw}}^{\uparrow M \uparrow F} ⨾ \mathrm{ftn}_M^{\uparrow F} ⨾ \overline{\phi}^{\uparrow M \uparrow F} ⨾ \mathrm{ftn}_M^{\uparrow F} ⨾ r = \left(\overline{\mathrm{sw}}^{\uparrow M} ⨾ \underline{\mathrm{ftn}_M ⨾ \overline{\phi}^{\uparrow M}} ⨾ \mathrm{ftn}_M\right)^{\uparrow F} ⨾ r \\
&= \left(\overline{\mathrm{sw}}^{\uparrow M} ⨾ \phi^{\uparrow M \uparrow M} ⨾ \underline{\mathrm{ftn}_M ⨾ \mathrm{ftn}_M}\right)^{\uparrow F} ⨾ r = \left(\underline{\overline{\mathrm{sw}}^{\uparrow M} ⨾ \phi^{\uparrow M \uparrow M} ⨾ \mathrm{ftn}_M^{\uparrow M}} ⨾ \mathrm{ftn}_M\right)^{\uparrow F} ⨾ r \\
&\text{inductive assumption}: \quad \left(\left(\overline{\phi} ⨾ \underline{\mathrm{ftn}_M}\right)^{\uparrow M} ⨾ \mathrm{ftn}_M\right)^{\uparrow F} ⨾ r = \left(\overline{\phi}^{\uparrow M} ⨾ \mathrm{ftn}_M ⨾ \mathrm{ftn}_M\right)^{\uparrow F} ⨾ r \quad .
\end{aligned}$$

Now substitute Eq. (14.70) into the right-hand side of Eq. (14.71):

$$\phi ⨾ \mathrm{ftn}_M = \left|\begin{array}{c||c} & M^{M^A} \\ \hline M^A & \mathrm{pu}_M \\ F^{M^{L^{M^A}}} & \overline{\phi}^{\uparrow M \uparrow F} ⨾ \mathrm{ftn}_M^{\uparrow F} ⨾ r \end{array}\right| ⨾ \mathrm{ftn}_M = \left|\begin{array}{c||c} & M^A \\ \hline M^A & \mathrm{pu}_M ⨾ \mathrm{ftn}_M \\ F^{M^{L^{M^A}}} & \overline{\phi}^{\uparrow M \uparrow F} ⨾ \mathrm{ftn}_M^{\uparrow F} ⨾ r ⨾ \mathrm{ftn}_M \end{array}\right| \quad .$$

The difference between the two sides of Eq. (14.71) is now reduced to:

$$\left(\overline{\phi}^{\uparrow M} ⨾ \mathrm{ftn}_M ⨾ \underline{\mathrm{ftn}_M}\right)^{\uparrow F} ⨾ r \stackrel{?}{=} \overline{\phi}^{\uparrow M \uparrow F} ⨾ \mathrm{ftn}_M^{\uparrow F} ⨾ \underline{r ⨾ \mathrm{ftn}_M} \quad .$$

This holds due to naturality of $r : F \rightsquigarrow \mathrm{Id}$, which proves Eq. (14.71).

The requirements of Statement 14.6.2.2 are now satisfied, so brun (θ) is a monad morphism.

The nondegeneracy law of `brun` is an equation between functions of type $M^A \to M^A$:

$$\begin{aligned}
&\text{expect to equal id}: \quad \mathrm{flift} ⨾ \mathrm{brun}\,(\theta) = \mathrm{pu}_L^{\uparrow M} ⨾ \mathrm{brun}\,(\theta) = \mathrm{pu}_L^{\uparrow M} ⨾ \underline{\xi^{\uparrow M} ⨾ \theta^{\uparrow M}} ⨾ \mathrm{ftn}_M = \left(\underline{\mathrm{pu}_L ⨾ \phi}\right)^{\uparrow M} ⨾ \mathrm{ftn}_M \\
&\text{identity law of } \phi: \quad = \mathrm{pu}_M^{\uparrow M} ⨾ \mathrm{ftn}_M = \mathrm{id} \quad .
\end{aligned}$$

Monadic naturality laws The monadic naturality law of `flift` is:

$$\begin{aligned}
&\mathrm{flift} ⨾ \mathrm{frun}\,(\phi) \stackrel{?}{=} \phi ⨾ \mathrm{flift} \quad , \\
&\text{or, using definitions of flift and frun}: \quad \mathrm{pu}_L^{\uparrow M} ⨾ \psi^{\uparrow M} ⨾ \phi \stackrel{?}{=} \phi ⨾ \mathrm{pu}_{L'}^{\uparrow M'} \quad ,
\end{aligned}$$

which should hold for any natural transformation $\phi^{:M\rightsquigarrow M'}$. Here, we denoted $L' \stackrel{\text{def}}{=} \text{Free}^{F\circ M'}$. Since $\psi : L \rightsquigarrow L'$ is a monad morphism, we find:

$$\text{expect to equal } \phi\,\text{⨾}\,\text{pu}_{L'}^{\uparrow M'} : \quad \text{pu}_L^{\uparrow M}\,\text{⨾}\,\psi^{\uparrow M}\,\text{⨾}\,\phi = (\text{pu}_L\,\text{⨾}\,\psi)^{\uparrow M}\,\text{⨾}\,\phi = \phi\,\text{⨾}\,\underline{(\text{pu}_L\,\text{⨾}\,\psi)}^{\uparrow M'} = \phi\,\text{⨾}\,\text{pu}_{L'}^{\uparrow M'} \quad .$$

The monadic naturality law of brun_L is:

$$\text{brun}_L(\theta)\,\text{⨾}\,\phi \stackrel{?}{=} \text{frun}\,(\phi)\,\text{⨾}\,\text{brun}_{L'}(\theta) \quad .$$

This law should hold for any monad morphism $\phi : M \rightsquigarrow M'$. Write the two sides separately:

$$\begin{aligned}
\text{left-hand side}: \quad & \text{brun}_L(\theta)\,\text{⨾}\,\phi = \xi^{\uparrow M}\,\text{⨾}\,\theta^{\uparrow M}\,\text{⨾}\,\underline{\text{ftn}_M\,\text{⨾}\,\phi} \\
\text{composition law of } \phi: \quad & = \underline{\xi^{\uparrow M}\,\text{⨾}\,\theta^{\uparrow M}\,\text{⨾}\,\phi^{\uparrow M}}\,\text{⨾}\,\phi\,\text{⨾}\,\text{ftn}_{M'} = (\xi\,\text{⨾}\,\theta\,\text{⨾}\,\phi)^{\uparrow M}\,\text{⨾}\,\phi\,\text{⨾}\,\text{ftn}_{M'} \quad , \\
\text{right-hand side}: \quad & \text{frun}\,(\phi)\,\text{⨾}\,\text{brun}_{L'}(\theta) = \psi^{\uparrow M}\,\text{⨾}\,\underline{\phi\,\text{⨾}\,\xi^{\uparrow M'}\,\text{⨾}\,\theta^{\uparrow M'}}\,\text{⨾}\,\text{ftn}_{M'} \\
\text{naturality of } \phi: \quad & = (\psi\,\text{⨾}\,\xi\,\text{⨾}\,\theta)^{\uparrow M}\,\text{⨾}\,\phi\,\text{⨾}\,\text{ftn}_M \quad .
\end{aligned}$$

The remaining difference is a comparison of functions of type $L^A \to M'^A$:

$$\xi\,\text{⨾}\,\theta\,\text{⨾}\,\phi \stackrel{?}{=} \psi\,\text{⨾}\,\xi\,\text{⨾}\,\theta \quad . \tag{14.72}$$

Substituting Eq. (14.66) instead of ψ and Eq. (14.70) instead of $\xi\,\text{⨾}\,\theta$, we transform this equality to:

$$\left\| \begin{array}{c} \text{pu}_M \\ \overline{\xi}^{\uparrow M\uparrow F}\,\text{⨾}\,\theta^{\uparrow M\uparrow F}\,\text{⨾}\,\text{ftn}_M^{\uparrow F}\,\text{⨾}\,r \end{array} \right| \text{⨾}\,\phi = \left\| \begin{array}{c} \underline{\text{pu}_M\,\text{⨾}\,\phi} \\ \overline{\xi}^{\uparrow M\uparrow F}\,\text{⨾}\,\theta^{\uparrow M\uparrow F}\,\text{⨾}\,\text{ftn}_M^{\uparrow F}\,\text{⨾}\,r\,\text{⨾}\,\phi \end{array} \right| = \left\| \begin{array}{c} \text{pu}_{M'} \\ \overline{\xi}^{\uparrow M\uparrow F}\,\text{⨾}\,\theta^{\uparrow M\uparrow F}\,\text{⨾}\,\text{ftn}_M^{\uparrow F}\,\text{⨾}\,r\,\text{⨾}\,\phi \end{array} \right|$$

$$\stackrel{?}{=} \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \overline{\psi}^{\uparrow M\uparrow F}\,\text{⨾}\,\phi^{\uparrow F} \end{array} \right| \text{⨾} \left\| \begin{array}{c} \text{pu}_{M'} \\ \overline{\xi}^{\uparrow M'\uparrow F}\,\text{⨾}\,\theta^{\uparrow M'\uparrow F}\,\text{⨾}\,\text{ftn}_{M'}^{\uparrow F}\,\text{⨾}\,r \end{array} \right| = \left\| \begin{array}{c} \text{pu}_{M'} \\ \overline{\psi}^{\uparrow M\uparrow F}\,\text{⨾}\,\phi^{\uparrow F}\,\text{⨾}\,\overline{\xi}^{\uparrow M'\uparrow F}\,\text{⨾}\,\theta^{\uparrow M'\uparrow F}\,\text{⨾}\,\text{ftn}_{M'}^{\uparrow F}\,\text{⨾}\,r \end{array} \right| \quad .$$

The remaining difference is:

$$\overline{\xi}^{\uparrow M\uparrow F}\,\text{⨾}\,\theta^{\uparrow M\uparrow F}\,\text{⨾}\,\text{ftn}_M^{\uparrow F}\,\text{⨾}\,r\,\text{⨾}\,\phi \stackrel{?}{=} \overline{\psi}^{\uparrow M\uparrow F}\,\text{⨾}\,\phi^{\uparrow F}\,\text{⨾}\,\overline{\xi}^{\uparrow M'\uparrow F}\,\text{⨾}\,\theta^{\uparrow M'\uparrow F}\,\text{⨾}\,\text{ftn}_{M'}^{\uparrow F}\,\text{⨾}\,r \quad .$$

By the inductive assumption, Eq. (14.72) holds for recursive calls $\overline{\xi}$ and $\overline{\psi}$. To use that assumption, we need to move these functions towards each other in the compositions above:

$$\begin{aligned}
& \overline{\xi}^{\uparrow M\uparrow F}\,\text{⨾}\,\theta^{\uparrow M\uparrow F}\,\text{⨾}\,\text{ftn}_M^{\uparrow F}\,\text{⨾}\,\underline{r\,\text{⨾}\,\phi} = \overline{\xi}^{\uparrow M\uparrow F}\,\text{⨾}\,\theta^{\uparrow M\uparrow F}\,\text{⨾}\,\underline{\text{ftn}_M^{\uparrow F}\,\text{⨾}\,\phi^{\uparrow F}}\,\text{⨾}\,r \\
\text{composition law of } \phi: \quad & = \underline{\overline{\xi}^{\uparrow M\uparrow F}\,\text{⨾}\,\theta^{\uparrow M\uparrow F}\,\text{⨾}\,(\phi^{\uparrow M}}\,\text{⨾}\,\phi\,\text{⨾}\,\text{ftn}_{M'})^{\uparrow F}\,\text{⨾}\,r = (\underline{\overline{\xi}\,\text{⨾}\,\theta\,\text{⨾}\,\phi})^{\uparrow M\uparrow F}\,\text{⨾}\,\phi^{\uparrow F}\,\text{⨾}\,\text{ftn}_{M'}^{\uparrow F}\,\text{⨾}\,r \\
\text{inductive assumption}: \quad & = ((\overline{\psi}\,\text{⨾}\,\underline{\xi\,\text{⨾}\,\theta)^{\uparrow M}\,\text{⨾}\,\phi}\,\text{⨾}\,\text{ftn}_{M'})^{\uparrow F}\,\text{⨾}\,r = ((\overline{\psi}^{\uparrow M}\,\text{⨾}\,\phi\,\text{⨾}\,(\xi\,\text{⨾}\,\theta)^{\uparrow M'}\,\text{⨾}\,\text{ftn}_{M'})^{\uparrow F}\,\text{⨾}\,r \quad .
\end{aligned}$$

The two sides are now equal.

This concludes the proofs of the laws for the free monad's transformer.

14.7.2 Transformer for the `List` monad. Proofs

The `List` monad's transformer, usually named `ListT`, is defined as a recursive type:

$$T_{\text{List}}^{M,A} \stackrel{\text{def}}{=} M^{\mathbb{1}+A\times T_{\text{List}}^{M,A}} \quad ,$$

that wraps each partial tail of the list into a separate layer of an M-effect. The result is an "effectful list": obtaining each next element of the list requires running an M-effect. The `flatten` method (ftn_T) should sequence all M-effects of the nested lists in the order they occur in the outer list.

Despite a superficial similarity between T_{List} and the transformers for the free monad and the free pointed monad, proofs of the laws for T_{List} require a different technique, and the transformer's implementation is significantly more complicated. The reason is that the functor L defined by:

$$L^A \stackrel{\text{def}}{=} \mathbb{1} + A \times T_{\text{List}}^{M,A} = \mathbb{1} + A \times M^{L^A} \quad ,$$

does not admit a monad instance that flattens the "effectful lists" correctly (see Exercise 14.9.5.21). So, we cannot define $T_{\text{List}} = M \circ L$ as a functor composition of two monads. As a result, the `flatten` method of T_{List}, which we will denote for brevity by ftn_T, cannot be defined via a `swap` function, as we have done for the other monad transformers just mentioned. Instead, the `flatten` method of T_{List} is defined via a special helper function named `prod` as shown in the following statement.[19]

Statement 14.7.2.1 The type constructor T_{List} satisfies all the monad transformer laws.

Proof We will shortly implement a function called `prod` with the type signature:

$$\text{prod} : L^{M^{L^A}} \to M^{L^A} \cong \mathbb{1} + M^{L^A} \times M^{L^{M^{L^A}}} \to M^{L^A} \quad .$$

The monad methods of T_{List} are then defined by:

$$\begin{tikzcd} T^{T^A} = M^{L^{M^{L^A}}} \arrow[r, "\text{prod}^{\uparrow M}"] \arrow[dr, "\text{ftn}_T \stackrel{\text{def}}{=}"'] & M^{M^{L^A}} \arrow[d, "\text{ftn}_M"] \\ & M^{L^A} \end{tikzcd}$$

$$\text{pu}_T \stackrel{\text{def}}{=} \text{pu}_L \,\text{⨾}\, \text{pu}_M \quad , \qquad \text{ftn}_T \stackrel{\text{def}}{=} \text{prod}^{\uparrow M} \,\text{⨾}\, \text{ftn}_M \quad . \tag{14.73}$$

The method $\text{pu}_L : A \to L^A$ creates a list with a single element and an empty effect:

$$\text{pu}_L \stackrel{\text{def}}{=} a^{:A} \to \mathbb{0}^{:\mathbb{1}} + a \times \text{pu}_M(1 + \mathbb{0}^{:A \times M^{L^A}}) \quad .$$

Since the value $1 + \mathbb{0}^{:A \times M^{L^A}}$ will be used frequently, we denote it by $\text{Nil}^{:L^A}$. So, we write:

$$\text{pu}_L \stackrel{\text{def}}{=} a^{:A} \to \mathbb{0}^{:\mathbb{1}} + a \times (\text{Nil} \triangleright \text{pu}_M) = \left| \begin{array}{c|cc} & \mathbb{1} & A \times M^{L^A} \\ \hline A & \mathbb{0} & a \to a \times \text{pu}_M(\text{Nil}) \end{array} \right| \quad .$$

Implementing `prod` The code of `prod` uses a helper function that we temporarily call `comb`, with type:

$$\text{comb} : M^{L^A} \to M^{L^A} \to M^{L^A} \quad .$$

This function "combines" two values of type $M^{L^A} \stackrel{\text{def}}{=} T_{\text{List}}^A$ into a new value of the same type:

$$\text{comb}\,(m)(n) \stackrel{\text{def}}{=} m \triangleright \text{flm}_M \left(\left| \begin{array}{c|c} & M^{L^A} \\ \hline \mathbb{1} & 1 \to n \\ A \times M^{L^A} & a \times p \to \text{pu}_M(\mathbb{0}^{:\mathbb{1}} + a \times \overline{\text{comb}}\,(p)(n)) \end{array} \right| \right) \quad . \tag{14.74}$$

The code uses the monad M's `flatMap` method to sequence all M-effects from the first list before all M-effects of the second one. We will show in Statement 14.7.2.2 that the `comb` function is a monoidal operation for values of type T_{List}^A; that property will be used in the later derivations. The function `prod` is defined recursively via `comb`:

$$\text{prod} \stackrel{\text{def}}{=} \left| \begin{array}{c|c} & M^{L^A} \\ \hline \mathbb{1} & 1 \to \text{Nil} \triangleright \text{pu}_M \\ M^{L^A} \times M^{L^{M^{L^A}}} & m \times p \to \text{comb}\,(m)(p \triangleright \overline{\text{prod}}^{\uparrow M} \,\text{⨾}\, \text{ftn}_M) \end{array} \right| \quad . \tag{14.75}$$

[19] The function `prod` was studied in the paper "Composing monads" (http://web.cecs.pdx.edu/~mpj/pubs/RR-1004.pdf). However, that paper did not correctly define the `List` monad's transformer.

The function `prod` satisfies certain laws that we will prove in Statement 14.7.2.5 below. Besides a naturality law, `prod` satisfies two identity laws and an associativity law:

$$\text{identity laws}: \quad \text{pu}_L \,⨾\, \text{prod} = \text{id} \quad , \qquad \text{pu}_T^{\uparrow L} \,⨾\, \text{prod} = \text{pu}_M \quad , \tag{14.76}$$

$$\text{associativity law}: \quad \text{prod} \,⨾\, \text{prod}^{\uparrow M} \,⨾\, \text{ftn}_M = \text{prod}^{\uparrow M \uparrow L} \,⨾\, \text{ftn}_M^{\uparrow L} \,⨾\, \text{prod} \quad . \tag{14.77}$$

$$\begin{tikzcd}
M^{L^A} \arrow[d, "\text{pu}_L"'] \arrow[dr, "\text{id}"] & & L^A \arrow[dl, "\text{pu}_M"'] \arrow[d, "\text{pu}_T^{\uparrow L}"] \\
L^{M^{L^A}} \arrow[r, "\text{prod}"] & M^{L^A} & L^{M^{L^A}} \arrow[l, "\text{prod}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
L^{M^{L^{M^{L^A}}}} \arrow[r, "\text{prod}^{\uparrow M \uparrow L}"] \arrow[d, "\text{prod}"] & L^{M^{M^{L^A}}} \arrow[r, "\text{ftn}_M^{\uparrow L}"] & L^{M^{L^A}} \arrow[d, "\text{prod}"] \\
M^{L^{M^{L^A}}} \arrow[r, "\text{prod}^{\uparrow M}"] & M^{M^{L^A}} \arrow[r, "\text{ftn}_M"] & M^{L^A}
\end{tikzcd}$$

We will now verify that these laws make the monad laws for T_{List} hold automatically:

Monad laws of T_{List} need to be verified using the monad methods pu_T and ftn_T we defined above. To verify the left identity law of T:

$$\begin{aligned}
&\text{expect to equal id}: \quad \text{pu}_T \,⨾\, \text{ftn}_T = \text{pu}_L \,⨾\, \underline{\text{pu}_M \,⨾\, \text{prod}^{\uparrow M}} \,⨾\, \text{ftn}_M = \underline{\text{pu}_L \,⨾\, \text{prod}} \,⨾\, \text{pu}_M \,⨾\, \text{ftn}_M \\
&\text{identity law of prod}: \quad = \text{pu}_M \,⨾\, \text{ftn}_M = \text{id} \quad .
\end{aligned}$$

To verify the right identity law of T:

$$\begin{aligned}
&\text{expect to equal id}: \quad \text{pu}_T^{\uparrow T} \,⨾\, \text{ftn}_T = \underline{\text{pu}_T^{\uparrow L \uparrow M} \,⨾\, \text{prod}^{\uparrow M}} \,⨾\, \text{ftn}_M \\
&\text{identity law of prod under } {}^{\uparrow M}: \quad = \underline{\text{pu}_M^{\uparrow M} \,⨾\, \text{ftn}_M} = \text{id} \quad .
\end{aligned}$$

To verify the associativity law of T:

$$\begin{aligned}
&\text{expect to equal } \text{ftn}_T \,⨾\, \text{ftn}_T: \quad \text{ftn}_T^{\uparrow T} \,⨾\, \text{ftn}_T = \underline{(\text{prod}^{\uparrow M} \,⨾\, \text{ftn}_M)^{\uparrow L \uparrow M} \,⨾\, \text{prod}^{\uparrow M}} \,⨾\, \text{ftn}_M \\
&\qquad = (\underline{\text{prod}^{\uparrow M \uparrow L} \,⨾\, \text{ftn}_M^{\uparrow L} \,⨾\, \text{prod}})^{\uparrow M} \,⨾\, \text{ftn}_M \\
&\text{associativity law of prod}: \quad = (\text{prod} \,⨾\, \text{prod}^{\uparrow M} \,⨾\, \underline{\text{ftn}_M})^{\uparrow M} \,⨾\, \underline{\text{ftn}_M} = \text{prod}^{\uparrow M} \,⨾\, \underline{\text{prod}^{\uparrow M \uparrow M} \,⨾\, \text{ftn}_M} \,⨾\, \text{ftn}_M \\
&\qquad = \text{prod}^{\uparrow M} \,⨾\, \text{ftn}_M \,⨾\, \text{prod}^{\uparrow M} \,⨾\, \text{ftn}_M = \text{ftn}_T \,⨾\, \text{ftn}_T \quad .
\end{aligned}$$

Identity laws When $M = \text{Id}$, the type constructor T_{List} is equivalent to `List` since T_{List} is then defined by the recursive equation:

$$T_{\text{List}}^A = M^{\mathbb{1} + A \times T_{\text{List}}^A} \cong \mathbb{1} + A \times T_{\text{List}}^A \quad .$$

After setting $M = \text{Id}$, the code of pu_T, `comb`, and ftn_T becomes the same as the code of pu_{List}, `concat` (++), and ftn_{List}. So, the identity laws hold for T.

Lifting laws The foreign lift function is implemented as:

$$\text{flift} : M^A \to M^{L^A} \quad , \qquad \text{flift} \stackrel{\text{def}}{=} \text{pu}_L^{\uparrow M} \quad .$$

To show that `flift` is a monad morphism, begin with the identity law:

$$\begin{aligned}
&\text{expect to equal } \text{pu}_T: \quad \text{pu}_M \,⨾\, \text{flift} = \underline{\text{pu}_M \,⨾\, \text{pu}_L^{\uparrow M}} = \text{pu}_L \,⨾\, \text{pu}_M \\
&\text{by definition of } \text{pu}_T: \quad = \text{pu}_T \quad .
\end{aligned}$$

To verify the composition law, write its two sides separately:

$$\begin{aligned}
&\text{left-hand side}: \quad \text{ftn}_M \,⨾\, \text{flift} = \text{ftn}_M \,⨾\, \text{pu}_L^{\uparrow M} \quad , \\
&\text{right-hand side}: \quad \text{flift}^{\uparrow M} \,⨾\, \text{flift} \,⨾\, \text{ftn}_T = \text{pu}_L^{\uparrow M \uparrow M} \,⨾\, \underline{\text{pu}_L^{\uparrow M} \,⨾\, \text{prod}^{\uparrow M}} \,⨾\, \text{ftn}_M \\
&\text{identity law of prod}: \quad = \underline{\text{pu}_L^{\uparrow M \uparrow M} \,⨾\, \text{ftn}_M} = \text{ftn}_M \,⨾\, \text{pu}_L^{\uparrow M} \quad .
\end{aligned}$$

Implementation of `frun` Since the transformer $T_{\text{List}} = M \circ L$ is not purely compositional (L itself depends on M), mapping the monad M to M' via a monad morphism $\phi : M \rightsquigarrow M'$ requires us also to map L to L', where we define:

$$L'^A \stackrel{\text{def}}{=} \mathbb{1} + A \times M'^{L'^A} \quad .$$

We denote that map temporarily by ψ. To be precise, ψ is a function of ϕ and should be denoted by $\psi(\phi)$, but we will write simply ψ for brevity (since ϕ is fixed throughout most of the derivation). We can implement $\psi : L^A \to L'^A$ by the following code:

$$\psi : L^A \to L'^A \quad , \qquad \psi \stackrel{\text{def}}{=} \left| \begin{array}{c|cc} & \mathbb{1} & A \times M'^{L'^A} \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A \times M^{L^A} & \mathbb{0} & \text{id} \boxtimes (\overline{\psi}^{\uparrow M} \,;\, \phi) \end{array} \right| \quad . \tag{14.78}$$

The foreign runner $(\text{frun}^{M,M'}(\phi) : T_{\text{List}} \rightsquigarrow T'_{\text{List}})$ is then implemented as:

$$\text{frun}^{M,M'} : (M \rightsquigarrow M') \to M^{L^A} \to M'^{L'^A} \quad , \quad \text{frun}^{M,M'}(\phi^{:M \rightsquigarrow M'}) \stackrel{\text{def}}{=} \psi^{\uparrow M} \,;\, \phi \quad .$$

Monadic naturality of `prod` When the monad M is mapped to M' via a monad morphism $\phi : M \rightsquigarrow M'$, the `prod` function with type signature $\text{prod}_{M,L} : L \circ M \circ L \rightsquigarrow M \circ L$ is replaced by the corresponding function $\text{prod}_{M',L'} : L' \circ M' \circ L' \rightsquigarrow M' \circ L'$. The function $\text{prod}_{M',L'}$ has the same code as $\text{prod}_{M,L}$ except for using the monad parameter M' instead of M. This monadic naturality property is expressed by the law analogous to Eq. (14.60):

$$\begin{tikzcd} L^{M^{L^A}} \arrow[r, "\psi^{\uparrow M \uparrow L}"] \arrow[dr, "\text{prod}_{M,L}"'] & L^{M^{L'^A}} \arrow[r, "\phi^{\uparrow L}"] & L^{M'^{L'^A}} \arrow[r, "\psi"] & L'^{M'^{L'^A}} \arrow[d, "\text{prod}_{M',L'}"] \\ & M^{L^A} \arrow[r, "\psi^{\uparrow M}"] & M^{L'^A} \arrow[r, "\phi"] & M'^{L'^A} \end{tikzcd}$$

$$\text{prod}_{M,L} \,;\, \psi^{\uparrow M} \,;\, \phi = \psi^{\uparrow M \uparrow L} \,;\, \phi^{\uparrow L} \,;\, \psi \,;\, \text{prod}_{M',L'} \quad . \tag{14.79}$$

This law guarantees that the code of $\text{prod}_{M,L}$ works in the same way for all foreign monads M. We will verify this law in Statement 14.7.2.5 below.

Runner laws To show that the foreign runner is a monad morphism, begin with the identity law:

$$\begin{aligned}
&\text{expect } \text{pu}_{M' \circ L'} : && \text{pu}_{M \circ L} \,;\, \text{frun}(\phi) = \text{pu}_L \,;\, \underline{\text{pu}_M \,;\, \psi^{\uparrow M}} \,;\, \phi = \text{pu}_L \,;\, \psi \,;\, \underline{\text{pu}_M \,;\, \phi} \\
&\text{identity law of } \phi : && = \text{pu}_L \,;\, \psi \,;\, \text{pu}_{M'} = \left\| \begin{array}{cc} \mathbb{0} & a \to a \times \text{pu}_M(\text{Nil}) \end{array} \right| \,;\, \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \boxtimes (\overline{\psi}^{\uparrow M} \,;\, \phi) \end{array} \right| \,;\, \text{pu}_{M'} \\
& && = \left\| \begin{array}{cc} \mathbb{0} & a \to a \times (\underline{\text{Nil} \triangleright \text{pu}_M \,;\, \overline{\psi}^{\uparrow M} \,;\, \phi}) \end{array} \right| \,;\, \text{pu}_{M'} \\
& && = \left\| \begin{array}{cc} \mathbb{0} & a \to a \times (\text{Nil} \triangleright \text{pu}_{M'}) \end{array} \right| \,;\, \text{pu}_{M'} = \text{pu}_{L'} \,;\, \text{pu}_{M'} = \text{pu}_{M' \circ L'} \quad .
\end{aligned}$$

Here we used the following computation:

$$\text{Nil} \triangleright \underline{\text{pu}_M \,;\, \psi^{\uparrow M}} \,;\, \phi = \text{Nil} \triangleright \psi \,;\, \underline{\text{pu}_M \,;\, \phi} = \text{Nil} \triangleright \text{pu}_{M'} \quad .$$

As part of this derivation, we have found the property:

$$\text{pu}_L \,;\, \psi = \text{pu}_{L'} \quad , \tag{14.80}$$

which resembles the identity law of a monad morphism for $\psi : L \rightsquigarrow L'$. However, L and L' do not have the right monad structures we can use here.

To verify the composition law, write its two sides separately:

$$\begin{aligned}
&\text{left-hand side :} && \text{ftn}_T \,;\, \text{frun}(\phi) = \text{flm}_M(\text{prod}) \,;\, \psi^{\uparrow M} \,;\, \phi = \text{flm}_M(\text{prod} \,;\, \psi^{\uparrow M}) \,;\, \phi \\
&\phi\text{'s composition law :} && = \phi \,;\, \text{flm}_{M'}(\text{prod} \,;\, \psi^{\uparrow M} \,;\, \phi) \quad , \\
&\text{right-hand side :} && (\text{frun}(\phi))^{\uparrow T} \,;\, \text{frun}(\phi) \,;\, \text{ftn}_{T'} = \underline{(\psi^{\uparrow M} \,;\, \phi)^{\uparrow L \uparrow M} \,;\, \psi^{\uparrow M} \,;\, \phi} \,;\, \text{flm}_{M'}(\text{prod}) \\
&\text{naturality law of } \phi : && = \phi \,;\, ((\psi^{\uparrow M} \,;\, \phi)^{\uparrow L} \,;\, \psi)^{\uparrow M'} \,;\, \text{flm}_{M'}(\text{prod}) = \phi \,;\, \text{flm}_{M'}((\psi^{\uparrow M} \,;\, \phi)^{\uparrow L} \,;\, \psi \,;\, \text{prod}) \quad .
\end{aligned}$$

The remaining difference is:

$$\text{prod} \fatsemi \psi^{\uparrow M} \fatsemi \phi \stackrel{?}{=} (\psi^{\uparrow M} \fatsemi \phi)^{\uparrow L} \fatsemi \psi \fatsemi \text{prod} \quad .$$

The last equation is the same as the monadic naturality law (14.79) of `prod`.

Functor laws of `frun` To verify the functor laws of `frun`, begin with the identity law:

$$\text{frun}\,(\text{id}) = \psi(\text{id}) \fatsemi \text{id} = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \boxtimes (\overline{\psi(\text{id})}^{\uparrow M} \fatsemi \text{id}) \end{array} \right| \fatsemi \text{id} = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \boxtimes \text{id} \end{array} \right| = \text{id} \quad .$$

Then verify the composition law, assuming $\phi_1 : M \rightsquigarrow M'$ and $\phi_2 : M' \rightsquigarrow M''$ are monad morphisms:

$$\begin{aligned} & \text{frun}\,(\phi_1) \fatsemi \text{frun}\,(\phi_2) = \psi^{\uparrow M}(\phi_1) \fatsemi \underline{\phi_1 \fatsemi \psi^{\uparrow M'}(\phi_2)} \fatsemi \phi_2 \\ \text{naturality law of } \phi_1 : \quad & = \psi^{\uparrow M}(\phi_1) \fatsemi \psi^{\uparrow M}(\phi_2) \fatsemi \phi_1 \fatsemi \phi_2 = \big(\psi(\phi_1) \fatsemi \psi(\phi_2)\big)^{\uparrow M} \fatsemi \phi_1 \fatsemi \phi_2 \\ & \stackrel{?}{=} \text{frun}\,(\phi_1 \fatsemi \phi_2) = \psi(\phi_1 \fatsemi \phi_2)^{\uparrow M} \fatsemi \phi_1 \fatsemi \phi_2 \quad . \end{aligned}$$

The remaining difference is:

$$\psi(\phi_1) \fatsemi \psi(\phi_2) \stackrel{?}{=} \psi(\phi_1 \fatsemi \phi_2) \quad ,$$

which we verify by writing:

$$\begin{aligned} \psi(\phi_1) \fatsemi \psi(\phi_2) &= \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \boxtimes (\overline{\psi(\phi_1)}^{\uparrow M} \fatsemi \phi_1) \end{array} \right| \fatsemi \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \boxtimes (\overline{\psi(\phi_2)}^{\uparrow M} \fatsemi \phi_2) \end{array} \right| \\ \text{compute composition :} \quad &= \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \boxtimes (\underline{\overline{\psi(\phi_1)}^{\uparrow M} \fatsemi \phi_1 \fatsemi \overline{\psi(\phi_2)}^{\uparrow M}} \fatsemi \phi_2) \end{array} \right| \\ \text{inductive assumption :} \quad &= \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \boxtimes (\overline{\psi(\phi_1 \fatsemi \phi_2)}^{\uparrow M} \fatsemi \phi_1 \fatsemi \phi_2) \end{array} \right| = \psi(\phi_1 \fatsemi \phi_2) \quad . \end{aligned}$$

Base runner laws The base runner ($\text{brun} : T^A \to M^A$) cannot be implemented for arbitrary types A: since a list may be empty, it is not always possible to extract a value of type M^A out of a given value of type $T^{M,A}_{\text{List}}$. However, for a monoid type R with binary operation $\oplus_R$ and empty element e_R, the type signature $\text{List}^R \to R$ is implemented by the standard `reduce` operation:

$$\text{reduce} : \text{List}^R \to R \quad , \qquad \text{reduce} \stackrel{\text{def}}{=} \left| \begin{array}{c|c} & R \\ \hline \mathbb{1} & 1 \to e_R \\ R \times \text{List}^R & h \times t \to h \oplus_R \overline{\text{reduce}}\,(t) \end{array} \right| \quad .$$

We can similarly implement a base runner (`brun`) for the transformer $T^{M,A}_{\text{List}}$ if we restrict its usage to *monoid* types $A = R$. The function `brun` with the type signature $M^{L^R} \to M^R$ aggregates all elements of the effectful list into a single value of type M^R (which is also a monoid type):

$$\text{brun} : M^{L^R} \to M^R \quad , \qquad \text{brun} \stackrel{\text{def}}{=} \text{flm}_M \left(\left| \begin{array}{c|c} & M^R \\ \hline \mathbb{1} & 1 \to \text{pu}_M(e_R) \\ R \times M^{L^R} & h \times t \to \text{pu}_M(h) \oplus_M \overline{\text{brun}}\,(t) \end{array} \right| \right) \quad .$$

Here we use the monoid M^R's binary operation $\oplus_M$ defined by:

$$p^{:M^R} \oplus_M q^{:M^R} \stackrel{\text{def}}{=} p \triangleright \text{flm}_M \big(u^{:R} \to q \triangleright (v^{:R} \to u \oplus_R v)^{\uparrow M}\big) \quad . \tag{14.81}$$

It can be shown that `brun` satisfies the nondegeneracy law:

$$\text{expect to equal id}: \quad \text{flift} \,⨾\, \text{brun} = \text{pu}_L^{\uparrow M} \,⨾\, \text{flm}_M \left(\left\| \begin{array}{c} 1 \to \text{pu}_M(e_R) \\ r \times t \to \text{pu}_M(r) \oplus_M \overline{\text{brun}}(t) \end{array} \right| \right)$$

$$\text{naturality of flm}_M: \quad = \text{flm}_M \left(\underline{\text{pu}_L} \,⨾\, \left\| \begin{array}{c} 1 \to \text{pu}_M(e_R) \\ r \times t \to \text{pu}_M(r) \oplus_M \overline{\text{brun}}(t) \end{array} \right| \right)$$

$$\text{definition of pu}_L: \quad = \text{flm}_M \left(\left\| \mathbb{0} \quad a \to a \times (\text{Nil} \triangleright \text{pu}_M) \right| \,⨾\, \left\| \begin{array}{c} 1 \to \text{pu}_M(e_R) \\ r \times t \to \text{pu}_M(r) \oplus_M \overline{\text{brun}}(t) \end{array} \right| \right)$$

$$= \text{flm}_M (a \to \text{pu}_M(a) \oplus_M \underline{\text{brun}(\text{Nil} \triangleright \text{pu}_M)})$$

$$\text{use Eq. (14.82)}: \quad = \text{flm}_M (a \to \underline{\text{pu}_M(a) \oplus \text{pu}_M(e_R)}) = \text{flm}_M (\underline{a \to \text{pu}_M(a)}) = \text{flm}_M(\text{pu}_M) = \text{id} \quad .$$

Here we used the property that $\text{pu}_M(e_R)$ is the unit value for $\oplus_M$, and also computed:

$$\text{brun}(\text{Nil} \triangleright \text{pu}_M) = \text{Nil} \triangleright \underline{\text{pu}_M \,⨾\, \text{flm}_M} \left(\left\| \begin{array}{c} 1 \to \text{pu}_M(e_R) \\ r \times t \to \text{pu}_M(r) \oplus_M \overline{\text{brun}}(t) \end{array} \right| \right) = \text{pu}_M(e_R) \quad . \tag{14.82}$$

The monad morphism identity law holds for `brun`:

$$\text{expect to equal pu}_M: \quad \text{pu}_T \,⨾\, \text{brun} = \text{pu}_L \,⨾\, \underline{\text{pu}_M \,⨾\, \text{flm}_M} \left(\left\| \begin{array}{c} 1 \to \text{pu}_M(e_R) \\ r \times t \to \text{pu}_M(r) \oplus_M \overline{\text{brun}}(t) \end{array} \right| \right)$$

$$\text{identity law of flm}_M: \quad = \left\| \mathbb{0} \quad a^{:R} \to a \times (\text{Nil} \triangleright \text{pu}_M) \right| \,⨾\, \left\| \begin{array}{c} 1 \to \text{pu}_M(e_R) \\ r \times t \to \text{pu}_M(r) \oplus_M \overline{\text{brun}}(t) \end{array} \right|$$

$$\text{matrix composition}: \quad = a^{:R} \to \text{pu}_M(a) \oplus_M \underline{\overline{\text{brun}}(\text{Nil} \triangleright \text{pu}_M)}$$

$$\text{use Eq. (14.82)}: \quad = a \to \text{pu}_M(a) \underline{\oplus_M \text{pu}_M(e_R)} = a \to \text{pu}_M(a) = \text{pu}_M \quad .$$

It is not clear whether the monad morphism composition law of `brun` holds (Problem 15.3.2.11). If the composition law does not hold, the base runner `brun` must be applied to an entire monadic program. Running several parts of a program separately and composing the results as values of type M^A will not give the same value as running the entire program.

Monadic naturality laws For `flift`, the monadic naturality law is written as:

$$\text{flift}^M \,⨾\, \text{frun}(\phi) = \phi \,⨾\, \text{flift}^{M'} \quad ,$$

where $\phi : M \rightsquigarrow M'$ is an arbitrary natural transformation. Begin with the left-hand side:

$$\text{expect to equal } \phi \,⨾\, \text{flift}^{M'}: \quad \text{flift}^M \,⨾\, \text{frun}(\phi) = \text{pu}_L^{\uparrow M} \,⨾\, \psi^{\uparrow M} \,⨾\, \phi = (\text{pu}_L \,⨾\, \psi)^{\uparrow M} \,⨾\, \phi$$

$$\text{naturality law of } \phi: \quad = \phi \,⨾\, (\text{pu}_L \,⨾\, \psi)^{\uparrow M}$$

$$\text{use Eq. (14.80)}: \quad = \phi \,⨾\, \text{pu}_{L'}^{\uparrow M'} = \phi \,⨾\, \text{flift}^{M'} \quad .$$

The monadic naturality law of `brun` is written for any monad morphism $\phi : M \rightsquigarrow M'$:

$$\text{brun}^M \,⨾\, \phi = \text{frun}(\phi) \,⨾\, \text{brun}^{M'} \quad .$$

Begin with the left-hand side:

$$\text{flm}_M \left(\left\| \begin{array}{c} 1 \to \text{pu}_M(e_R) \\ r \times t \to \text{pu}_M(r) \oplus_M \overline{\text{brun}}(t) \end{array} \right| \right) \,⨾\, \phi = \phi \,⨾\, \text{flm}_{M'} \left(\left\| \begin{array}{c} 1 \to \text{pu}_M(e_R) \\ r \times t \to \text{pu}_M(r) \oplus_M \overline{\text{brun}}(t) \end{array} \right| \,⨾\, \phi \right) \quad .$$

Using the equality $\text{frun}(\phi) = \psi^{\uparrow M} ⨾ \phi = \phi ⨾ \psi^{\uparrow M'}$, we write the right-hand side as

$$\text{frun}(\phi) ⨾ \text{brun} = \phi ⨾ \underline{\psi^{\uparrow M'} ⨾ \text{flm}_{M'}}\left(\left\| \begin{array}{c} 1 \to \text{pu}_{M'}(e_R) \\ r \times t \to \text{pu}_{M'}(r) \oplus_{M'} \overline{\text{brun}}(t) \end{array} \right|\right)$$

$$\text{naturality law of } \text{flm}_{M'}: \quad = \phi ⨾ \text{flm}_{M'}\left(\psi ⨾ \left\| \begin{array}{c} 1 \to \text{pu}_{M'}(e_R) \\ r \times t \to \text{pu}_{M'}(r) \oplus_{M'} \overline{\text{brun}}(t) \end{array} \right|\right) \quad .$$

The remaining difference is:

$$\left\| \begin{array}{c} 1 \to \text{pu}_M(e_R) \\ r \times t \to \text{pu}_M(r) \oplus_M \overline{\text{brun}}^M(t) \end{array} \right| ⨾ \phi \stackrel{?}{=} \psi ⨾ \left\| \begin{array}{c} 1 \to \text{pu}_{M'}(e_R) \\ r \times t \to \text{pu}_{M'}(r) \oplus_{M'} \overline{\text{brun}}^{M'}(t) \end{array} \right| \quad .$$

Rewrite the left-hand side:

$$\left\| \begin{array}{c} 1 \to \text{pu}_M(e_R) \\ r \times t \to \text{pu}_M(r) \oplus_M \overline{\text{brun}}^M(t) \end{array} \right| ⨾ \phi = \left\| \begin{array}{c} 1 \to e_R \triangleright \underline{\text{pu}_M ⨾ \phi} \\ r \times t \to \phi\big(\text{pu}_M(r) \oplus_M \overline{\text{brun}}^M(t)\big) \end{array} \right|$$

$$= \left\| \begin{array}{c} 1 \to e_R \triangleright \text{pu}_{M'} \\ r \times t \to \phi\big(\text{pu}_M(r) \oplus_M \overline{\text{brun}}^M(t)\big) \end{array} \right| \quad .$$

Rewrite the right-hand side:

$$\psi ⨾ \left\| \begin{array}{c} 1 \to \text{pu}_{M'}(e_R) \\ r \times t \to \text{pu}_{M'}(r) \oplus_{M'} \overline{\text{brun}}^{M'}(t) \end{array} \right|$$

$$\text{definition of } \psi: \quad = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \boxtimes \text{frun}(\phi) \end{array} \right| ⨾ \left\| \begin{array}{c} 1 \to \text{pu}_{M'}(e_R) \\ r \times t \to \text{pu}_{M'}(r) \oplus_{M'} \overline{\text{brun}}^{M'}(t) \end{array} \right|$$

$$\text{matrix composition}: \quad = \left\| \begin{array}{c} 1 \to \text{pu}_{M'}(e_R) \\ r \times t \to \text{pu}_{M'}(r) \oplus_{M'} \big(t \triangleright \underline{\text{frun}(\phi) ⨾ \overline{\text{brun}}^{M'}}\big) \end{array} \right|$$

$$\text{inductive assumption}: \quad = \left\| \begin{array}{c} 1 \to \text{pu}_{M'}(e_R) \\ r \times t \to \text{pu}_{M'}(r) \oplus_{M'} \big(t \triangleright \text{brun}^M ⨾ \phi\big) \end{array} \right| \quad .$$

The remaining difference is in the last rows of the matrices:

$$\phi\big(\text{pu}_M(r) \oplus_M \overline{\text{brun}}^M(t)\big) \stackrel{?}{=} \text{pu}_{M'}(r) \oplus_{M'} \phi\big(t \triangleright \text{brun}^M\big) \quad .$$

This equality will be proved if we prove the monadic naturality of $\oplus_M$ defined by Eq. (14.81):

$$\phi(p^{:M^R} \oplus_M q^{:M^R}) = \phi(p) \oplus_{M'} \phi(q) \quad .$$

A proof is given by the following derivation:

$$\text{expect to equal } \phi(p) \oplus_M \phi(q): \quad (p \oplus_M q) \triangleright \phi = p \triangleright \text{flm}_M\big(u \to q \triangleright (v \to u \oplus_R v)^{\uparrow M}\big) ⨾ \phi$$

$$\text{composition law of } \phi: \quad = p \triangleright \phi ⨾ \text{flm}_{M'}(u \to q \triangleright (v \to u \oplus_R v)^{\underline{\uparrow M ⨾ \phi}})$$

$$\text{naturality law of } \phi: \quad = \phi(p) \triangleright \text{flm}_{M'}(u \to \phi(q) \triangleright (v \to u \oplus_R v)^{\uparrow M'})$$

$$\text{definition (14.81) of } \oplus_M: \quad = \phi(p) \oplus_M \phi(q) \quad .$$

□

We conclude this section with derivations of some properties that were used in the proofs above. The next statement shows that M^{L^A} is a monoid for any fixed type A.

Statement 14.7.2.2 Given any monad M, define the functor L recursively by:

$$L^A \stackrel{\text{def}}{=} \mathbb{1} + A \times M^{L^A} \quad .$$

Denote $\text{Nil} \stackrel{\text{def}}{=} 1 + \mathbb{0}^{:A \times M^{L^A}}$. Then the function `comb` defined by Eq. (14.74) satisfies:

$$\begin{aligned} &\text{identity laws}: && \text{comb}\,(\text{Nil} \triangleright \text{pu}_M)(m^{:M^{L^A}}) = m \quad , \qquad \text{comb}\,(m^{:M^{L^A}})(\text{Nil} \triangleright \text{pu}_M) = m \quad , \\ &\text{associativity law}: && \text{comb}\,(x)\big(\text{comb}\,(y)(z)\big) = \text{comb}\,\big(\text{comb}\,(x)(y)\big)(z) \quad . \end{aligned}$$

These laws show that the type M^{L^A} is a monoid with empty value $\text{pu}_M(\text{Nil})$.

Proof To verify the left identity law:

$$\begin{aligned} &\text{expect to equal } m: && \text{comb}\,(\text{Nil} \triangleright \text{pu}_M)(m^{:M^{L^A}}) \\ & && = \text{Nil} \triangleright \underline{\text{pu}_M \triangleright \text{flm}_M}\left(\begin{array}{c|c} & M^{L^A} \\ \hline \mathbb{1} & 1 \to m \\ A \times M^{L^A} & a \times p \to \text{pu}_M\big(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(m)\big) \end{array}\right) \\ &\text{identity law of } M: && = \text{Nil} \triangleright \left|\begin{array}{c|c} & M^{L^A} \\ \hline \mathbb{1} & \underline{1 \to m} \\ A \times M^{L^A} & a \times p \to \text{pu}_M\big(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(m)\big) \end{array}\right| = m \quad . \end{aligned}$$

To verify the right identity law:

$$\begin{aligned} &\text{expect to equal } m: && \text{comb}\,(m^{:M^{L^A}})(\text{Nil} \triangleright \text{pu}_M) \\ & && = m \triangleright \text{flm}_M\left(\begin{array}{c|c} & M^{L^A} \\ \hline \mathbb{1} & 1 \to \text{Nil} \triangleright \text{pu}_M \\ A \times M^{L^A} & a \times p \to \text{pu}_M\big(\mathbb{0} + a \times \underline{\overline{\text{comb}}\,(p)(\text{Nil} \triangleright \text{pu}_M)}\big) \end{array}\right) \\ &\text{inductive assumption}: && = m \triangleright \text{flm}_M\left(\begin{array}{c|c} & M^{L^A} \\ \hline \mathbb{1} & 1 \to \text{Nil} \triangleright \text{pu}_M \\ A \times M^{L^A} & a \times p \to \text{pu}_M\big(\mathbb{0} + a \times p\big) \end{array}\right) \\ & && = m \triangleright \text{flm}_M\left(\left|\begin{array}{c} (1 \to 1 + \mathbb{0}) \fatsemi \text{pu}_M \\ (a \times p \to \mathbb{0} + a \times p) \fatsemi \text{pu}_M \end{array}\right|\right) = m \triangleright \underline{\text{flm}_M\,(\underline{\text{id} \fatsemi \text{pu}_M})} = m \quad . \end{aligned}$$

To verify the associativity law of `comb`, it is convenient to express `comb` through a helper function ξ:

$$\text{comb}\,(m)(n) \stackrel{\text{def}}{=} m \triangleright \text{flm}_M\,(n \triangleright \xi) = m \triangleright (n \triangleright \xi)^{\uparrow M} \fatsemi \text{ftn}_M \quad , \tag{14.83}$$

$$\text{where } \xi \text{ is defined by}: \quad n \triangleright \xi \stackrel{\text{def}}{=} \left|\begin{array}{c|c} & M^{L^A} \\ \hline \mathbb{1} & 1 \to n \\ A \times M^{L^A} & a \times p \to \text{pu}_M\big(\mathbb{0}^{:\mathbb{1}} + a \times \overline{\text{comb}}\,(p)(n)\big) \end{array}\right| \quad .$$

We then write the two sides of the associativity law. The left-hand side:

$$\text{comb}\,(x)\big(\text{comb}\,(y)(z)\big) = x \triangleright \left|\begin{array}{c} 1 \to \text{comb}\,(y)(z) \\ a \times p \to \text{pu}_M\big(\mathbb{0} + a \times \underline{\overline{\text{comb}}\,(p)(\text{comb}\,(y)(z))}\big) \end{array}\right|^{\uparrow M} \fatsemi \text{ftn}_M \quad .$$

Use the inductive assumption to rewrite the last expression as:

$$x \triangleright \left\| \begin{array}{c} 1 \to \text{comb}\,(y)(z) \\ a \times p \to \text{pu}_M\left(\mathbb{0} + a \times \overline{\text{comb}}\,(\text{comb}\,(p)(y))(z)\right) \end{array} \right|^{\uparrow M} ⨾ \text{ftn}_M \quad .$$

The right-hand side:

$$\text{comb}\left(\text{comb}\,(x)(y)\right)(z) = \text{comb}\,(x)(y) \triangleright (z \triangleright \xi)^{\uparrow M} ⨾ \text{ftn}_M = x \triangleright (y \triangleright \xi)^{\uparrow M} ⨾ \underline{\text{ftn}_M ⨾ (z \triangleright \xi)^{\uparrow M}} ⨾ \text{ftn}_M$$
$$= x \triangleright (y \triangleright \xi)^{\uparrow M} ⨾ (z \triangleright \xi)^{\uparrow M \uparrow M} ⨾ \underline{\text{ftn}_M ⨾ \text{ftn}_M} = x \triangleright \left((y \triangleright \xi) ⨾ (z \triangleright \xi)^{\uparrow M} ⨾ \text{ftn}_M\right)^{\uparrow M} ⨾ \text{ftn}_M \quad .$$

Write out the expression $(y \triangleright \xi) ⨾ (z \triangleright \xi)^{\uparrow M} ⨾ \text{ftn}_M$ (having type $\mathbb{1} + A \times M^{L^A} \to M^{L^A}$) separately:

$$(y \triangleright \xi) ⨾ (z \triangleright \xi)^{\uparrow M} ⨾ \text{ftn}_M$$
$$= \left\| \begin{array}{c} 1 \to y \\ a \times p \to \text{pu}_M\left(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(y)\right) \end{array} \right| ⨾ (z \triangleright \xi)^{\uparrow M} ⨾ \text{ftn}_M$$
$$= \left\| \begin{array}{c} 1 \to \underline{y \triangleright (z \triangleright \xi)^{\uparrow M} ⨾ \text{ftn}_M} \\ a \times p \to \left(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(y)\right) \triangleright \underline{\text{pu}_M} ⨾ (z \triangleright \xi)^{\uparrow M} ⨾ \underline{\text{ftn}_M} \end{array} \right|$$
$$= \left\| \begin{array}{c} 1 \to \text{comb}\,(y)(z) \\ a \times p \to \left(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(y)\right) \triangleright (z \triangleright \xi) \end{array} \right| \quad .$$

The difference between the two sides is only in the last rows of the matrices:

$$\text{pu}_M\left(\mathbb{0} + a \times \overline{\text{comb}}\,(\text{comb}\,(p)(y))(z)\right) \stackrel{?}{=} \left(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(y)\right) \triangleright (z \triangleright \xi) \quad .$$

It remains to simplify the right-hand side above and make it equal to the left-hand side:

$$\left(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(y)\right) \triangleright (z \triangleright \xi)$$
$$= \left(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(y)\right) \triangleright \left\| \begin{array}{c} 1 \to z \\ a \times p \to \text{pu}_M\left(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(z)\right) \end{array} \right|$$
$$= \text{pu}_M\left(\mathbb{0} + a \times \overline{\text{comb}}\,(\overline{\text{comb}}\,(p)(y))(z)\right) \quad .$$

This completes the proof.

Statement 14.7.2.3 The function `comb` defined by Eq. (14.74) satisfies the distributive law:

$$\left(\text{comb}\,(p^{:T^{T^A}})(q^{:T^{T^A}})\right) \triangleright \text{ftn}_T = \text{comb}\,(p \triangleright \text{ftn}_T)(q \triangleright \text{ftn}_T) \quad ,$$

where we denoted $T^A \stackrel{\text{def}}{=} M^{L^A}$ for brevity, and ftn_T is defined by Eq. (14.73).

Proof Begin with the left-hand side of the distributive law and use Eq. (14.83):

$$\left(\text{comb}\,(p)(q)\right) \triangleright \text{ftn}_T = p \triangleright \text{flm}_M(q \triangleright \xi) \triangleright \text{ftn}_T = p \triangleright \text{flm}_M(q \triangleright \xi) ⨾ \text{flm}_M(\text{prod})$$
$$\text{associativity of flm}_M: \quad = p \triangleright \text{flm}_M\left((q \triangleright \xi) ⨾ \text{flm}_M(\text{prod})\right) \quad .$$

Rewrite the right-hand side of the distributive law:

$$\text{comb}\,(p \triangleright \text{ftn}_T)(q \triangleright \text{ftn}_T) = p \triangleright \underline{\text{ftn}_T} \triangleright \text{flm}_M(q \triangleright \text{ftn}_T \triangleright \xi)$$
$$= p \triangleright \text{flm}_M(\text{prod}) ⨾ \text{flm}_M(q \triangleright \text{ftn}_T \triangleright \xi)$$
$$\text{associativity of flm}_M: \quad = p \triangleright \text{flm}_M\left(\text{prod} ⨾ \text{flm}_M(q \triangleright \text{ftn}_T \triangleright \xi)\right) \quad .$$

It remains to prove the following equality between functions of type $L^{T^A} \to T^A$:

$$\text{prod} \,⨾\, \text{flm}_M(q \triangleright \text{ftn}_T \triangleright \xi) \stackrel{?}{=} (q \triangleright \xi) \,⨾\, \text{flm}_M(\text{prod}) \quad .$$

Since we have no matrix representation for flm_M, it is inconvenient to use matrices at this step. Instead, we substitute an arbitrary value of type $L^{T^A} \cong \mathbb{1} + T^A \times T^{T^A}$ into both sides of the equation above. A value of type L^{T^A} has two cases, `Nil` and $h^{:T^A} \times t^{:T^{T^A}}$. Substitute `Nil` into the left-hand side:

$$\underline{\text{Nil} \triangleright \text{prod}} \,⨾\, \text{flm}_M(q \triangleright \text{ftn}_T \triangleright \xi) = \text{Nil} \triangleright \underline{\text{pu}_M \,⨾\, \text{flm}_M}(q \triangleright \text{ftn}_T \triangleright \xi)$$

$$\text{identity law of } M: \quad = \text{Nil} \triangleright (q \triangleright \text{ftn}_T \triangleright \xi) \quad .$$

Since:

$$\text{Nil} \triangleright (s \triangleright \xi) = \left| \begin{array}{cc} 1 & \mathbb{0} \end{array} \right| \triangleright \left\| \begin{array}{c} 1 \to s \\ a \times p \to \text{pu}_M(\mathbb{0} + a \times \overline{\text{comb}}(p)(s)) \end{array} \right| = s \quad ,$$

we can simplify:

$$\text{Nil} \triangleright (q \triangleright \text{ftn}_T \triangleright \xi) = q \triangleright \text{ftn}_T \quad .$$

Substituting `Nil` into the right-hand side yields:

$$\underline{\text{Nil} \triangleright (q \triangleright \xi)} \triangleright \text{flm}_M(\text{prod}) = q \triangleright \underline{\text{flm}_M(\text{prod})} = q \triangleright \text{ftn}_T \quad .$$

Both sides are now equal.

It remains to substitute a value of the form $\mathbb{0} + h^{:T^A} \times t^{:T^{T^A}}$. For the left-hand side:

$$(\mathbb{0} + h \times t) \triangleright \text{prod} \,⨾\, \text{flm}_M(q \triangleright \text{ftn}_T \triangleright \xi)$$

$$= (\mathbb{0} + h \times t) \triangleright \left\| \begin{array}{c} 1 \to \text{Nil} \triangleright \text{pu}_M \\ m \times p \to \text{comb}(m)(p \triangleright \text{flm}_M(\overline{\text{prod}})) \end{array} \right| \,⨾\, \text{flm}_M(q \triangleright \text{ftn}_T \triangleright \xi)$$

$$\text{apply matrix}: \quad = \big(\text{comb}(h)(t \triangleright \text{ftn}_T)\big) \triangleright \text{flm}_M(q \triangleright \text{ftn}_T \triangleright \xi) \quad .$$

Substitute $\mathbb{0} + h \times t$ into the right-hand side:

$$(\mathbb{0} + h \times t) \triangleright (q \triangleright \xi) \,⨾\, \text{flm}_M(\text{prod})$$

$$\text{definition of } c: \quad = (\mathbb{0} + h \times t) \triangleright \left\| \begin{array}{c} 1 \to q \triangleright \text{flm}_M(\text{prod}) \\ a \times p \to \big(\mathbb{0} + a \times \overline{\text{comb}}(p)(q)\big) \triangleright \underline{\text{pu}_M \,⨾\, \text{flm}_M}(\text{prod}) \end{array} \right|$$

$$\text{apply matrix function}: \quad = \big(\mathbb{0} + h \times \overline{\text{comb}}(t)(q)\big) \triangleright \text{prod}$$

$$\text{definition of prod}: \quad = \big(\mathbb{0} + h \times \overline{\text{comb}}(t)(q)\big) \triangleright \left\| \begin{array}{c} 1 \to \text{Nil} \triangleright \text{pu}_M \\ m \times p \to \text{comb}(m)(p \triangleright \overline{\text{ftn}_T}) \end{array} \right|$$

$$\text{apply matrix}: \quad = \text{comb}(h)\big(\underline{(\overline{\text{comb}}(t)(q)) \triangleright \overline{\text{ftn}_T}}\big)$$

$$\text{inductive assumption}: \quad = \text{comb}(h)\big(\text{comb}(t \triangleright \text{ftn}_T)(q \triangleright \text{ftn}_T)\big)$$

$$\text{associativity law of comb}: \quad = \underline{\text{comb}}\big(\text{comb}(h)((t \triangleright \text{ftn}_T))\big)(q \triangleright \text{ftn}_T)$$

$$\text{definition (14.74) of comb}: \quad = \big(\text{comb}(h)((t \triangleright \text{ftn}_T))\big) \triangleright \text{flm}_M(q \triangleright \text{ftn}_T \triangleright \xi) \quad .$$

The two sides are now equal. (The associativity law of `comb` was proved in Statement 14.7.2.2.)

Statement 14.7.2.4 The function `comb` defined by Eq. (14.83) satisfies the monadic naturality law with respect to the foreign monad M: For any two monads M and M' with a monad morphism $\phi : M \rightsquigarrow M'$, we define ψ (which depends on ϕ) by Eq. (14.78). The law is then written as:

$$\big(\text{comb}(p^{:T^A})(q^{:T^A})\big) \triangleright \psi^{\uparrow M} \,⨾\, \phi = \text{comb}(p \triangleright \psi^{\uparrow M} \,⨾\, \phi)(q \triangleright \psi^{\uparrow M} \,⨾\, \phi) \quad .$$

Note that `comb` must use the monad M' in the right-hand side of the law.

Proof We write the left-hand side of the law:

$$\begin{aligned}
& \text{comb}\,(p)(q) \triangleright \psi^{\uparrow M} \fatsemi \phi = p \triangleright \text{flm}_M\,(q \triangleright \xi) \fatsemi \underline{\psi^{\uparrow M} \fatsemi \phi} \\
\text{naturality law of } \phi: \quad & = p \triangleright \underline{\text{flm}_M\,(q \triangleright \xi) \fatsemi \phi} \fatsemi \psi^{\uparrow M'} \\
\text{composition law of } \phi: \quad & = p \triangleright \phi \fatsemi \text{flm}_{M'}\big((q \triangleright \xi) \fatsemi \phi\big) \fatsemi \underline{\psi^{\uparrow M'}} \\
\text{naturality law of } \text{flm}_{M'}: \quad & = p \triangleright \phi \fatsemi \text{flm}_{M'}\big((q \triangleright \xi) \fatsemi \phi \fatsemi \psi^{\uparrow M'}\big) \quad .
\end{aligned}$$

The right-hand side is rewritten as:

$$\begin{aligned}
& \text{comb}\,(p \triangleright \psi^{\uparrow M} \fatsemi \phi)(q \triangleright \psi^{\uparrow M} \fatsemi \phi) = p \triangleright \underline{\psi^{\uparrow M} \fatsemi \phi} \fatsemi \text{flm}_{M'}\big(q \triangleright \underline{\psi^{\uparrow M} \fatsemi \phi} \fatsemi \xi\big) \\
\text{naturality law of } \phi: \quad & = p \triangleright \phi \fatsemi \underline{\psi^{\uparrow M'} \fatsemi \text{flm}_{M'}}\big(q \triangleright \phi \fatsemi \psi^{\uparrow M'} \fatsemi \xi\big) \\
\text{naturality of } \text{flm}_{M'}: \quad & = p \triangleright \phi \fatsemi \text{flm}_{M'}\big(\psi \fatsemi (q \triangleright \phi \fatsemi \psi^{\uparrow M'} \fatsemi \xi)\big) \quad .
\end{aligned}$$

The remaining difference between the two sides is the following equality:

$$(q \triangleright \xi) \fatsemi \phi \fatsemi \psi^{\uparrow M'} \stackrel{?}{=} \psi \fatsemi (q \triangleright \phi \fatsemi \psi^{\uparrow M'} \fatsemi \xi) \quad .$$

This should hold for any $q^{:T^A}$. Write the two sides separately:

$$\begin{aligned}
\text{left-hand side}: \quad & (q \triangleright \xi) \fatsemi \phi \fatsemi \psi^{\uparrow M'} = \left\| \begin{array}{c} 1 \to q \triangleright \phi \fatsemi \psi^{\uparrow M'} \\ a \times p \to \big(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(q)\big) \triangleright \underline{\text{pu}_M \fatsemi \phi \fatsemi \psi^{\uparrow M'}} \end{array} \right\| \\
& = \left\| \begin{array}{c} 1 \to q \triangleright \phi \fatsemi \psi^{\uparrow M'} \\ a \times p \to \big(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(q)\big) \triangleright \psi \fatsemi \text{pu}_{M'} \end{array} \right\| \quad , \\
\text{right-hand side}: \quad & \psi \fatsemi (q \triangleright \phi \triangleright \psi^{\uparrow M'} \triangleright \xi) \\
& = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \boxtimes (\phi \fatsemi \overline{\psi}^{\uparrow M'}) \end{array} \right\| \fatsemi \left\| \begin{array}{c} 1 \to q \triangleright \phi \triangleright \psi^{\uparrow M'} \\ a \times p \to \big(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(q \triangleright \phi \triangleright \psi^{\uparrow M'})\big) \triangleright \text{pu}_{M'} \end{array} \right\| \\
& = \left\| \begin{array}{c} 1 \to q \triangleright \phi \fatsemi \psi^{\uparrow M'} \\ a \times p \to \big(\mathbb{0} + a \times \overline{\text{comb}}\,(p \triangleright \phi \fatsemi \overline{\psi}^{\uparrow M'})(q \triangleright \phi \fatsemi \psi^{\uparrow M'})\big) \triangleright \text{pu}_{M'} \end{array} \right\| \quad .
\end{aligned}$$

By the inductive assumption:

$$\overline{\text{comb}}\,(p \triangleright \phi \fatsemi \overline{\psi}^{\uparrow M'})(q \triangleright \phi \fatsemi \psi^{\uparrow M'}) = \big(\text{comb}\,(p)(q)\big) \triangleright \phi \fatsemi \psi^{\uparrow M'} \quad .$$

So, the remaining difference between the two sides is:

$$\big(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(q)\big) \triangleright \psi \stackrel{?}{=} \mathbb{0} + a \times \big(\text{comb}\,(p)(q)\big) \triangleright \phi \fatsemi \psi^{\uparrow M'} \quad .$$

Using the definition (14.78) of ψ, we find:

$$\begin{aligned}
& \big(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(q)\big) \triangleright \psi = \big(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(q)\big) \triangleright \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \boxtimes (\phi \fatsemi \overline{\psi}^{\uparrow M'}) \end{array} \right\| \\
& = a \times \big(\overline{\text{comb}}\,(p)(q)\big) \triangleright \phi \fatsemi \overline{\psi}^{\uparrow M'} \quad ,
\end{aligned}$$

which equals the right-hand side above. This completes the proof.

Statement 14.7.2.5 The function `prod` defined by Eq. (14.75) satisfies Eqs. (14.76)–(14.79).

Proof To verify the left identity law of `prod`, note that $\text{Nil} \triangleright \text{prod} = \text{pu}_M(\text{Nil})$ and write:

$$\begin{aligned}
\text{expect to equal id}: \quad & \text{pu}_L \,;\, \text{prod} \\
&= \left\| \begin{array}{cc} \mathbb{0} & a \to a \times (\text{Nil} \triangleright \text{pu}_M) \end{array} \right| \,;\, \left\| \begin{array}{c} 1 \to \text{Nil} \triangleright \text{pu}_M \\ m \times p \to \text{comb}\,(m)(p \triangleright \overline{\text{prod}}^{\uparrow M} \,;\, \text{ftn}_M) \end{array} \right| \\
&= a^{:M^{L^A}} \to \text{comb}\,(a)(\text{Nil} \triangleright \underline{\text{pu}_M \,;\, \overline{\text{prod}}^{\uparrow M}} \,;\, \text{ftn}_M) \\
&= a \to \text{comb}\,(a)(\text{Nil} \triangleright \text{prod} \,;\, \underline{\text{pu}_M \,;\, \text{ftn}_M}) = a \to \text{comb}\,(a)(\underline{\text{Nil} \triangleright \text{prod}}) \\
&= a \to \underline{\text{comb}\,(a)(\text{Nil} \triangleright \text{pu}_M)} = a \to a = \text{id} \quad .
\end{aligned}$$

In the last line, we used an identity law of `comb` from Statement 14.7.2.2.

To verify the right identity law of `prod`, show that its left-hand side equals pu_M:

$$\begin{aligned}
&\text{pu}_T^{\uparrow L} \,;\, \text{prod} = (\text{pu}_L \,;\, \text{pu}_M)^{\uparrow L} \,;\, \text{prod} \\
&= \left| \begin{array}{c|cc} & \mathbb{1} & M^{L^A} \times M^{L^{M^{L^A}}} \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A \times M^{L^A} & \mathbb{0} & \text{pu}_T \boxtimes \overline{\text{pu}}_T^{\uparrow L \uparrow M} \end{array} \right\| \,;\, \left| \begin{array}{c|c} & M^{L^A} \\ \hline \mathbb{1} & 1 \to \text{Nil} \triangleright \text{pu}_M \\ M^{L^A} \times M^{L^{M^{L^A}}} & m \times p \to \text{comb}\,(m)(p \triangleright \overline{\text{prod}}^{\uparrow M} \,;\, \text{ftn}_M) \end{array} \right| \\
&= \left| \begin{array}{c|c} & M^{L^A} \\ \hline \mathbb{1} & 1 \to \text{Nil} \triangleright \text{pu}_M \\ A \times M^{L^A} & a \times m \to \text{comb}\,(a \triangleright \text{pu}_T)(m \triangleright \overline{\text{pu}}_T^{\uparrow L \uparrow M} \,;\, \overline{\text{prod}}^{\uparrow M} \,;\, \text{ftn}_M) \end{array} \right| \quad .
\end{aligned}$$

Simplify the second row of the matrix separately:

$$\begin{aligned}
& \text{comb}\,(a \triangleright \text{pu}_T)(m \triangleright \underline{\overline{\text{pu}}_T^{\uparrow L \uparrow M} \,;\, \overline{\text{prod}}^{\uparrow M}} \,;\, \text{ftn}_M) \\
\text{inductive assumption}: \quad & = \text{comb}\,(a \triangleright \text{pu}_T)(m \triangleright \underline{\text{pu}_M \,;\, \text{ftn}_M}) = \text{comb}\,(a \triangleright \text{pu}_T)(m) \\
\text{definition of comb}: \quad & = a \triangleright \text{pu}_L \,;\, \underline{\text{pu}_M \,;\, \text{flm}_M}\left(\left\| \begin{array}{c} 1 \to m \\ a \times p \to \text{pu}_M(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(m)) \end{array} \right| \right) \\
\text{identity law of } M: \quad & = \underline{a \triangleright \text{pu}_L} \triangleright \left\| \begin{array}{c} 1 \to m \\ a \times p \to \text{pu}_M(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(m)) \end{array} \right| \\
\text{definition of } \text{pu}_L: \quad & = \left\| \begin{array}{cc} \mathbb{0} & a \times (\text{Nil} \triangleright \text{pu}_M) \end{array} \right| \triangleright \left\| \begin{array}{c} 1 \to m \\ a \times p \to \text{pu}_M(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(m)) \end{array} \right| \\
\text{apply matrix}: \quad & = \text{pu}_M(\mathbb{0} + a \times \overline{\text{comb}}\,(p)(\text{Nil} \triangleright \text{pu}_M)) = \text{pu}_M(\mathbb{0} + a \times m) \quad .
\end{aligned}$$

So, we find that the code matrix for $\text{pu}_T^{\uparrow L} \,;\, \text{prod}$ is:

$$\begin{aligned}
\text{pu}_T^{\uparrow L} \,;\, \text{prod} &= \left| \begin{array}{c|c} & M^{L^A} \\ \hline \mathbb{1} & 1 \to \text{Nil} \triangleright \text{pu}_M \\ A \times M^{L^A} & a \times m \to \text{pu}_M(\mathbb{0} + a \times m) \end{array} \right| = \left| \begin{array}{c|c} & M^{L^A} \\ \hline \mathbb{1} & 1 \to \text{pu}_M(1 + \mathbb{0}) \\ A \times M^{L^A} & a \times m \to \text{pu}_M(\mathbb{0} + a \times m) \end{array} \right| \\
&= \left| \begin{array}{c|c} & \mathbb{1} + A \times M^{L^A} \\ \hline \mathbb{1} & 1 \to (1 + \mathbb{0}) \\ A \times M^{L^A} & a \times m \to (\mathbb{0} + a \times m) \end{array} \right| \,;\, \text{pu}_M = \text{id} \,;\, \text{pu}_M = \text{pu}_M \quad .
\end{aligned}$$

This proves the right identity law.

To verify the associativity law, write its two sides separately. We will use the formula $\text{ftn}_T = \text{prod}^{\uparrow M} ⨾ \text{ftn}_M$ to make the derivation shorter, but we will not use any laws of ftn_T in the derivation (since those laws will be established only after this proof is complete).

Some useful properties are:

$$\text{Nil} \triangleright \text{prod} = \text{Nil} \triangleright \text{pu}_M \quad ,$$
$$\text{pu}_M ⨾ \text{ftn}_T = \underline{\text{pu}_M ⨾ \text{prod}^{\uparrow M}} ⨾ \text{ftn}_M = \text{prod} ⨾ \underline{\text{pu}_M ⨾ \text{ftn}_M} = \text{prod} \quad .$$

Begin with the left-hand side of the associativity law:

$$\text{prod} ⨾ \underline{\text{prod}^{\uparrow M} ⨾ \text{ftn}_M} = \text{prod} ⨾ \text{ftn}_T = \left\| \begin{array}{c} 1 \to \text{Nil} \triangleright \underline{\text{pu}_M ⨾ \text{ftn}_T} \\ m \times p \to \text{comb}\,(m)(p \triangleright \overline{\text{prod}}^{\uparrow M} ⨾ \text{ftn}_M) \triangleright \underline{\text{ftn}_T} \end{array} \right|$$
$$= \left\| \begin{array}{c} 1 \to \underline{\text{Nil} \triangleright \text{prod}} \\ m \times p \to \text{comb}\,(m \triangleright \text{ftn}_T)(p \triangleright \underline{\overline{\text{prod}}^{\uparrow M} ⨾ \text{ftn}_M} ⨾ \text{ftn}_T) \end{array} \right|$$
$$= \left\| \begin{array}{c} 1 \to \text{Nil} \triangleright \text{pu}_M \\ m \times p \to \text{comb}\,(m \triangleright \text{ftn}_T)(p \triangleright \overline{\text{ftn}}_T ⨾ \text{ftn}_T) \end{array} \right| \quad .$$

To transform the right-hand side of the associativity law, we need the lifting of ftn_T to L:

$$\text{prod}^{\uparrow M \uparrow L} ⨾ \text{ftn}_M^{\uparrow L} = \text{ftn}_T^{\uparrow L} = \left| \begin{array}{c|cc} & \mathbb{1} & T^A \times T^{T^A} \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ T^{T^A} \times T^{T^{T^A}} & \mathbb{0} & h \times t \to (h \triangleright \text{ftn}_T) \times (t \triangleright \text{ftn}_T^{\overline{\uparrow L} \uparrow M}) \end{array} \right| \quad .$$

Then the right-hand side is written as:

$$\text{prod}^{\uparrow M \uparrow L} ⨾ \text{ftn}_M^{\uparrow L} ⨾ \text{prod} = \text{ftn}_T^{\uparrow L} ⨾ \text{prod}$$
$$= \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h \times t \to (h \triangleright \text{ftn}_T) \times (t \triangleright \text{ftn}_T^{\overline{\uparrow L} \uparrow M}) \end{array} \right| ⨾ \left\| \begin{array}{c} 1 \to \text{Nil} \triangleright \text{pu}_M \\ m \times p \to \text{comb}\,(m)(p \triangleright \overline{\text{ftn}}_T) \end{array} \right|$$
$$= \left\| \begin{array}{c} 1 \to \text{Nil} \triangleright \text{pu}_M \\ h \times t \to \text{comb}\,(h \triangleright \text{ftn}_T)(t \triangleright \text{ftn}_T^{\overline{\uparrow L} \uparrow M} \triangleright \overline{\text{ftn}}_T) \end{array} \right| \quad .$$

The two matrices differ only in the last rows. We rename m, p to h, t and get:

$$\text{comb}\,(h \triangleright \text{ftn}_T)(t \triangleright \overline{\text{ftn}}_T ⨾ \text{ftn}_T) \stackrel{?}{=} \text{comb}\,(h \triangleright \text{ftn}_T)(t \triangleright \text{ftn}_T^{\overline{\uparrow L} \uparrow M} \triangleright \overline{\text{ftn}}_T) \quad .$$

The remaining difference is:

$$\overline{\text{ftn}}_T ⨾ \text{ftn}_T \stackrel{?}{=} \text{ftn}_T^{\overline{\uparrow L} \uparrow M} ⨾ \overline{\text{ftn}}_T \quad .$$

This looks like the associativity law for ftn_T applied to some recursive calls to ftn_T (note that $f^{\uparrow L \uparrow M} = f^{\uparrow T}$ by definition of $T \stackrel{\text{def}}{=} M \circ L$). However, here we may *not* assume that the associativity law holds for ftn_T. The only allowed inductive hypothesis is that the law (14.77) holds when it involves any recursive calls to `prod`. So, we need to rewrite the above equation through $\overline{\text{prod}}$:

$$\text{left-hand side}: \quad \overline{\text{ftn}}_T ⨾ \text{ftn}_T = \text{flm}_M(\overline{\text{prod}}) ⨾ \text{flm}_M(\text{prod})$$
$$\text{associativity law of } \text{flm}_M: \quad = \text{flm}_M(\overline{\text{prod}} ⨾ \text{flm}_M(\text{prod}))$$
$$\text{right-hand side}: \quad \stackrel{?}{=} \text{ftn}_T^{\overline{\uparrow L} \uparrow M} ⨾ \overline{\text{ftn}}_T = \text{ftn}_T^{\overline{\uparrow L} \uparrow M} ⨾ \text{flm}_M(\overline{\text{prod}})$$
$$\text{naturality law of } \text{flm}_M: \quad = \text{flm}_M(\text{ftn}_T^{\overline{\uparrow L}} ⨾ \overline{\text{prod}}) = \text{flm}_M(\text{ftn}_T^{\overline{\uparrow L}} ⨾ \overline{\text{prod}}) \quad .$$

It remains to show that:

$$\overline{\text{prod}} \fatsemi \text{flm}_M(\text{prod}) = \overline{\text{prod}} \fatsemi \text{prod}^{\uparrow M} \fatsemi \text{ftn}_M \stackrel{?}{=} \text{ftn}_T^{\overline{\uparrow L}} \fatsemi \overline{\text{prod}} = (\text{prod}^{\uparrow M} \fatsemi \text{ftn}_M)^{\uparrow L} \fatsemi \overline{\text{prod}} \quad .$$

This is equivalent to:

$$\overline{\text{prod}} \fatsemi \text{prod}^{\uparrow M} \fatsemi \text{ftn}_M \stackrel{?}{=} \text{prod}^{\uparrow M \uparrow L} \fatsemi \text{ftn}_M^{\uparrow L} \fatsemi \overline{\text{prod}} \quad ,$$

which holds by the inductive assumption. So, the associativity law of `prod` holds.

To verify the monadic naturality law of `prod`, begin with its left-hand side:

$$\text{prod}_{M,L} \fatsemi \psi^{\uparrow M} \fatsemi \phi = \left\| \begin{array}{c} 1 \to \text{Nil} \triangleright \underline{\text{pu}_M \fatsemi \psi^{\uparrow M}} \fatsemi \phi \\ h \times t \to \text{comb}\,(h)(t \triangleright \overline{\text{ftn}_T}) \triangleright \psi^{\uparrow M} \fatsemi \phi \end{array} \right|$$

$$= \left\| \begin{array}{c} 1 \to \underline{\text{Nil} \triangleright \psi} \fatsemi \underline{\text{pu}_M \fatsemi \phi} \\ h \times t \to \text{comb}\,(h)(t \triangleright \overline{\text{ftn}_T}) \triangleright \psi^{\uparrow M} \fatsemi \phi \end{array} \right| = \left\| \begin{array}{c} 1 \to \text{Nil} \triangleright \text{pu}_{M'} \\ h \times t \to \text{comb}\,(h)(t \triangleright \overline{\text{ftn}_T}) \triangleright \psi^{\uparrow M} \fatsemi \phi \end{array} \right| \quad .$$

For the right-hand side of the monadic naturality law, we simplify the expression $\psi \fatsemi \text{prod}$ separately:

$$\psi \fatsemi \text{prod}_{M',L'} = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h \times t \to h \times (t \triangleright \psi^{\uparrow M} \fatsemi \phi) \end{array} \right| \fatsemi \left| \begin{array}{c|c} & M'^{L'^A} \\ \hline \mathbb{1} & 1 \to \text{Nil} \triangleright \text{pu}_{M'} \\ M'^{L'^A} \times M'^{L'^{M'^{L'^A}}} & m \times p \to \text{comb}\,(m)(p \triangleright \overline{\text{ftn}_{T'}}) \end{array} \right|$$

$$= \left\| \begin{array}{c} 1 \to \text{Nil} \triangleright \text{pu}_{M'} \\ h \times t \to \text{comb}\,(h)(t \triangleright \psi^{\uparrow M} \fatsemi \phi \fatsemi \overline{\text{ftn}_{T'}}) \end{array} \right| \quad .$$

Now the right-hand side of the law can be written as:

$$(\psi^{\uparrow M} \fatsemi \phi)^{\uparrow L} \fatsemi \psi \fatsemi \text{prod}_{M',L'}$$

$$= \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & h \times t \to (h \triangleright \psi^{\uparrow M} \fatsemi \phi) \times \left(t \triangleright (\psi^{\uparrow M} \fatsemi \phi)^{\overline{\uparrow L} \uparrow M}\right) \end{array} \right| \fatsemi \left\| \begin{array}{c} 1 \to \text{Nil} \triangleright \text{pu}_{M'} \\ h \times t \to \text{comb}\,(h)(t \triangleright \psi^{\uparrow M} \fatsemi \phi \fatsemi \overline{\text{ftn}_{T'}}) \end{array} \right|$$

$$= \left\| \begin{array}{c} 1 \to \text{Nil} \triangleright \text{pu}_{M'} \\ h \times t \to \text{comb}\,(h \triangleright \psi^{\uparrow M} \fatsemi \phi)\left(t \triangleright (\psi^{\uparrow M} \fatsemi \phi)^{\overline{\uparrow L} \uparrow M} \fatsemi \psi^{\uparrow M} \fatsemi \phi \fatsemi \overline{\text{ftn}_{T'}}\right) \end{array} \right| \quad .$$

The difference between the two sides is now only in the last rows of the matrices:

$$\text{comb}\,(h)(t \triangleright \overline{\text{ftn}_T}) \triangleright \psi^{\uparrow M} \fatsemi \phi \stackrel{?}{=} \text{comb}\,(h \triangleright \psi^{\uparrow M} \fatsemi \phi)\left(t \triangleright (\psi^{\uparrow M} \fatsemi \phi)^{\overline{\uparrow L} \uparrow M} \fatsemi \psi^{\uparrow M} \fatsemi \phi \fatsemi \overline{\text{ftn}_{T'}}\right) \quad .$$

By Statement 14.7.2.4, we have:

$$\text{comb}\,(h)(t \triangleright \overline{\text{ftn}_T}) \triangleright \psi^{\uparrow M} \fatsemi \phi = \text{comb}\,(h \triangleright \psi^{\uparrow M} \fatsemi \phi)(t \triangleright \overline{\text{ftn}_T} \fatsemi \psi^{\uparrow M} \fatsemi \phi) \quad .$$

The remaining difference between the two sides is:

$$(\psi^{\uparrow M} \fatsemi \phi)^{\overline{\uparrow L} \uparrow M} \fatsemi \psi^{\uparrow M} \fatsemi \phi \fatsemi \overline{\text{ftn}_{T'}} \stackrel{?}{=} \overline{\text{ftn}_T} \fatsemi \psi^{\uparrow M} \fatsemi \phi \quad .$$

This is the monad morphism composition law for $\text{frun}\,(\phi) \stackrel{\text{def}}{=} \psi^{\uparrow M} \fatsemi \phi$. However, the present proof must be completed without assuming that `frun` is a monad morphism (of type $T \rightsquigarrow T'$).

So, we need to replace ftn_T with `prod` in both sides above:

$$\begin{aligned}
\text{left-hand side}: \quad & (\psi^{\uparrow M} \fatsemi \phi)^{\overline{\uparrow L}\uparrow M} \fatsemi \psi^{\uparrow M} \fatsemi \phi \fatsemi \overline{\text{ftn}}_{T'} = \underline{(\psi^{\uparrow M} \fatsemi \phi)^{\overline{\uparrow L}\uparrow M} \fatsemi \psi^{\uparrow M} \fatsemi \phi} \fatsemi \text{flm}_{M'}(\overline{\text{prod}}) \\
\text{naturality law of } \phi: \quad & = \phi \fatsemi \left((\psi^{\uparrow M} \fatsemi \phi)^{\overline{\uparrow L}} \fatsemi \psi\right)^{\uparrow M'} \fatsemi \text{flm}_{M'}(\overline{\text{prod}}) \\
\text{naturality law of } \text{flm}_M: \quad & = \phi \fatsemi \text{flm}_{M'}((\psi^{\uparrow M} \fatsemi \phi)^{\overline{\uparrow L}} \fatsemi \psi \fatsemi \overline{\text{prod}}) \quad , \\
\text{right-hand side}: \quad & \overline{\text{ftn}}_T \fatsemi \psi^{\uparrow M} \fatsemi \phi = \text{flm}_M(\overline{\text{prod}}) \fatsemi \psi^{\uparrow M} \fatsemi \phi \\
\text{naturality law of } \text{flm}_M: \quad & = \text{flm}_M(\overline{\text{prod}} \fatsemi \psi^{\uparrow M}) \fatsemi \phi \\
\text{composition law of } \phi: \quad & = \phi \fatsemi \text{flm}_{M'}(\overline{\text{prod}} \fatsemi \psi^{\uparrow M} \fatsemi \phi) \quad .
\end{aligned}$$

It is clear that we need to use the inductive hypothesis to make progress. We adapt the inductive hypothesis to be used in the above equation:

$$\begin{aligned}
\text{inductive hypothesis}: \quad & \overline{\text{prod}}_{M,L} \fatsemi \psi^{\uparrow M} \fatsemi \phi = (\psi^{\uparrow M} \fatsemi \phi)^{\uparrow L} \fatsemi \psi \fatsemi \overline{\text{prod}}_{M',L'} \quad , \\
\text{apply } \text{flm}_{M'} \text{ to both sides}: \quad & \text{flm}_{M'}(\overline{\text{prod}}_{M,L} \fatsemi \psi^{\uparrow M} \fatsemi \phi) = \text{flm}_{M'}((\psi^{\uparrow M} \fatsemi \phi)^{\uparrow L} \fatsemi \psi \fatsemi \overline{\text{prod}}_{M',L'}) \quad .
\end{aligned}$$

Now we can rewrite both sides to obtain the following expressions:

$$\begin{aligned}
\text{left-hand side}: \quad & (\psi^{\uparrow M} \fatsemi \phi)^{\overline{\uparrow L}\uparrow M} \fatsemi \psi^{\uparrow M} \fatsemi \phi \fatsemi \overline{\text{ftn}}_{T'} = \phi \fatsemi \text{flm}_{M'}((\psi^{\uparrow M} \fatsemi \phi)^{\overline{\uparrow L}} \fatsemi \psi \fatsemi \overline{\text{prod}}) \\
\text{right-hand side}: \quad & \stackrel{?}{=} \overline{\text{ftn}}_T \fatsemi \psi^{\uparrow M} \fatsemi \phi = \phi \fatsemi \text{flm}_{M'}((\psi^{\uparrow M} \fatsemi \phi)^{\uparrow L} \fatsemi \psi \fatsemi \overline{\text{prod}}_{M',L'}) \quad .
\end{aligned}$$

Both sides are now equal.

This completes the proof of the laws for the `List` monad's transformer.

14.8 Incomplete transformers

A monad transformer that obeys the laws of a pointed and co-pointed functor in the category of monads (see Section 14.2.3) will give the programmer full flexibility of combining monadic effects. We may call such transformers **complete**. The monad transformers for linear monads, rigid monads, product monads, free pointed monads, and recursive (tree-like and list-like) monads are "complete", as we have proved in the previous sections. We now consider certain monads (state, continuation, choice, and codensity) whose transformers are "incomplete". Those transformers are deficient in one or another way, which leads to certain limitations in the practical use of the affected transformers.

14.8.1 Transformer for the state monad. Proofs

As shown in Examples 14.1.1.3–14.1.1.4, the `State` monad does not compose with several other monads. The `State` monad's transformer requires custom code (see Section 14.1.3).

Statement 14.8.1.1 The monad transformer (`StateT`) for the `State` monad is:

$$T^{M,A}_{\text{State}} \stackrel{\text{def}}{=} S \to M^{A \times S} \quad .$$

It satisfies all monad transformer laws except for the monad morphism laws for base runners.

Proof We need to verify all the monad transformer laws (Section 14.2.4) separately.

As we will see below, the `StateT` transformer has one problem: its base runner violates the monad morphism composition law. The standard runner for the `State` monad:

$$\text{run}_{\text{State}}(s_0) : \text{State}^{S,A} \to A \quad , \qquad \text{run}_{\text{State}}(s_0) \stackrel{\text{def}}{=} p^{:S \to A \times S} \to p(s_0) \triangleright \pi_1 \quad ,$$

also violates that law because it discards the updated state, while the composition law requires to pass the updated state to the next monad value. For the `State` monad, this problem can be solved by

starting the runner from an updated intermediate state instead of the fixed initial state s_0. However, that trick cannot be used with the `StateT` transformer whose updated state is wrapped as $M^{A\times S}$ in an arbitrary foreign monad M. The monad M's effect could wrap several values, or no values at all. The "updated intermediate state" is not well-defined in that case.

For this reason, a monadic program involving `StateT` cannot be refactored to run in several steps. To get correct results, the runner must be applied to the entire monadic program.

In the language of category theory, the `StateT` transformer fails to be a co-pointed functor in the category of monads. However, it is a pointed functor in that category: its lifts and the foreign runner satisfy all laws, as we will prove shortly.

Monad laws We need to show that T^M_{State} is a lawful monad, assuming that M is one. The monad methods for T^M_{State} are defined by Eqs. (14.1)–(14.2). To verify the monad laws, it is faster to use the uncurried Kleisli formulation, as we did for the `State` monad (Statement 10.2.7.2).

The type signature for a T_{State}-Kleisli function is:

$$A \to T^{M,B}_{\text{State}} = A \to S \to M^{B\times S} \quad .$$

When we uncurry this type signature, we get $A \times S \to M^{B\times S}$. It is easy to define the Kleisli composition ($\tilde{\diamond}_T$) for uncurried Kleisli functions because $\tilde{\diamond}_T$ is the same as the M-Kleisli composition ($\diamond_M$) except applied to types $A \times S$, $B \times S$, and $C \times S$:

$$f^{:A\times S\to M^{B\times S}} \tilde{\diamond}_T g^{:B\times S\to M^{C\times S}} \stackrel{\text{def}}{=} f \diamond_M g \quad .$$

The proper T_{State}-Kleisli composition ($\diamond_T$) is then defined by uncurrying:

$$f^{:A\to S\to M^{B\times S}} \diamond_T g^{:B\to S\to M^{C\times S}} \stackrel{\text{def}}{=} \text{uncu}\,(f) \diamond_M \text{uncu}\,(g) \quad .$$

The **uncurrying** transformation (`uncu`) defined as:

$$\text{uncu} : (A \to B \to C) \to A \times B \to C \quad , \qquad \text{uncu}\,(f) \stackrel{\text{def}}{=} a^{:A} \times b^{:B} \to f(a)(b) \quad ,$$

is an isomorphism with the inverse **currying** transformation (`cu`) defined as:

$$\text{cu} : (A \times B \to C) \to A \to B \to C \quad , \qquad \text{cu}\,(f) \stackrel{\text{def}}{=} a^{:A} \to b^{:B} \to f(a \times b) \quad .$$

So, it is sufficient to verify the monad laws in the uncurried Kleisli formulation.

To verify the identity laws, we need to define the uncurried `pure` method for T_{State}:

$$\tilde{\text{pu}}_T \stackrel{\text{def}}{=} \text{uncu}\,(\text{pu}_T) = a^{:A} \times s^{:S} \to \text{pu}_M(a \times s) = \text{pu}^{A\times S}_M \quad .$$

It is now quick to show that all three monad laws for T_{State} hold due to M's monad laws:

$$\tilde{\text{pu}}_T \tilde{\diamond}_T f = \text{pu}_M \diamond_M f = f \quad , \qquad f \tilde{\diamond}_T \tilde{\text{pu}}_T = f \diamond_M \text{pu}_M = f \quad ,$$
$$(f \tilde{\diamond}_T g) \tilde{\diamond}_T h = (f \diamond_M g) \diamond_M h = f \diamond_M (g \diamond_M h) = f \tilde{\diamond}_T (g \tilde{\diamond}_T h) \quad .$$

Identity laws When $M = \text{Id}$, the type of $T^{M,A}_{\text{State}}$ becomes equal to State^A, and the monad methods become equal to those of the `State` monad.

Lifting laws The `flift` function is defined by:

$$m^{:M^A} \triangleright \text{flift} \stackrel{\text{def}}{=} s^{:S} \to m \triangleright (a^{:A} \to a \times s)^{\uparrow M} \quad . \tag{14.84}$$

We need to show that `flift` is a monad morphism $M \rightsquigarrow T^M_{\text{State}}$. The identity law:

$$a^{:A} \triangleright \text{pu}_M \triangleright \text{flift} = s \to a \triangleright \underline{\text{pu}_M \fatsemi (a^{:A} \to a \times s)^{\uparrow M}}$$

naturality law of pu_M : $\quad = s \to a \triangleright (a^{:A} \to a \times s) \fatsemi \text{pu}_M = s \to \text{pu}_M(a \times s) = \text{pu}_T \quad .$

The composition law in terms of the Kleisli composition ($\diamond_T$) is:

$$(f^{:A\to M^B} \fatsemi \text{flift}) \diamond_T (g^{:B\to M^C} \fatsemi \text{flift}) \stackrel{?}{=} (f \diamond_M g) \fatsemi \text{flift} \quad .$$

In terms of $\tilde{\diamond}_T$, this equation is reformulated as:

$$\text{uncu}\,(f^{:A\to M^B} \fatsemi \text{flift}) \,\tilde{\diamond}_T\, \text{uncu}\,(g^{:B\to M^C} \fatsemi \text{flift}) \stackrel{?}{=} \text{uncu}\,\big((f \diamond_M g) \fatsemi \text{flift}\big) \quad .$$

To show that this equation holds, begin by uncurrying the arguments of $\diamond_T$:

$$\text{uncu}\,(f^{:A\to M^B} \fatsemi \text{flift}) = \text{uncu}\,(a^{:A} \to f(a) \triangleright \text{flift}) = \text{uncu}\,\big(a^{:A} \to s^{:S} \to a \triangleright f \triangleright (b^{:B} \to b \times s)^{\uparrow M}\big)$$
$$= a \times s \to a \triangleright f \fatsemi (b \to b \times s)^{\uparrow M} \quad .$$

Using this formula, we get:

$$\text{uncu}\,(f^{:A\to M^B} \fatsemi \text{flift}) \,\tilde{\diamond}_T\, \text{uncu}\,(g^{:B\to M^C} \fatsemi \text{flift}) = \text{uncu}\,(f^{:A\to M^B} \fatsemi \text{flift}) \diamond_M \text{uncu}\,(g^{:B\to M^C} \fatsemi \text{flift})$$
$$= \big(a \times s \to a \triangleright f \fatsemi (b \to b \times s)^{\uparrow M}\big) \diamond_M \big(b \times s \to b \triangleright g \fatsemi (c \to c \times s)^{\uparrow M}\big) \quad . \tag{14.85}$$

We need somehow to move $(b \to b \times s)^{\uparrow M}$ to the right of $\diamond_M$. However, the naturality law of $\diamond_M$ cannot be used because $(b \to b \times s)^{\uparrow M}$ contains the variable s, which is bound in the scope of the left argument of $\diamond_M$. To proceed, we apply the function in Eq. (14.85) to an arbitrary value $a \times s$. This will enable us to move the rest of the expression ($\diamond_M$...) into the scope of the left argument of $\diamond_M$:

$$(a \times s) \triangleright \big(a \times s \to a \triangleright f \fatsemi (b \to b \times s)^{\uparrow M}\big) \diamond_M \ldots$$
$$= (a \times s) \triangleright \big(a \times s \to a \triangleright f \fatsemi (b \to b \times s)^{\uparrow M}\big) \fatsemi \text{flm}_M(\ldots)$$
$$= a \triangleright f \fatsemi (b \to b \times s)^{\uparrow M} \fatsemi \text{flm}_M(\ldots) = a \triangleright f \fatsemi (b \to b \times s)^{\uparrow M} \diamond_M \ldots$$

This trick allows us to complete the derivation:

$$a \triangleright f \fatsemi (b \to b \times s)^{\uparrow M} \diamond_M \big(b \times s \to b \triangleright g \fatsemi (c \to c \times s)^{\uparrow M}\big)$$
$$\text{naturality law of } \diamond_M: \quad = a \triangleright f \diamond_M \big(b \to \underline{b \times s}\big) \fatsemi \big(\underline{b \times s} \to b \triangleright g \fatsemi (c \to c \times s)^{\uparrow M}\big)$$
$$\text{compute composition}: \quad = a \triangleright f \diamond_M \big(\underline{b \to b \triangleright} g \fatsemi (c \to c \times s)^{\uparrow M}\big)$$
$$\triangleright\text{-notation}: \quad = a \triangleright f \diamond_M \big(g \fatsemi \underline{(c \to c \times s)^{\uparrow M}}\big)$$
$$\text{naturality law of } \diamond_M: \quad = a \triangleright (f \diamond_M g) \fatsemi (c \to c \times s)^{\uparrow M} = (a \times s) \triangleright \text{uncu}\,\big((f \diamond_M g) \fatsemi \text{flift}\big) \quad .$$

Foreign runner laws The foreign runner (`frun`) is defined by:

$$\text{frun}\,(\phi^{:M\rightsquigarrow N}) \stackrel{\text{def}}{=} p^{:S\to M^{A\times S}} \to \underline{s^{:S} \to s \triangleright} p \triangleright \phi = p^{:S\to M^{A\times S}} \to p \fatsemi \phi \quad .$$

First, we need to prove that frun (ϕ) is a monad morphism of type $T^M_{\text{State}} \rightsquigarrow T^N_{\text{State}}$. For convenience, denote $P \stackrel{\text{def}}{=} T^M_{\text{State}}$ and $Q \stackrel{\text{def}}{=} T^N_{\text{State}}$. To verify the identity law:

$$\text{pu}_P \fatsemi \text{frun}\,(\phi^{:M\rightsquigarrow N}) = (a^{:A} \to s^{:S} \to (a \times s) \triangleright \text{pu}_M) \fatsemi (p \to p \fatsemi \phi)$$
$$\text{compute composition}: \quad = a \to s \to (a \times s) \triangleright \underline{\text{pu}_M \fatsemi \phi}$$
$$\text{identity law of } \phi: \quad = a \to s \to (a \times s) \triangleright \text{pu}_N = \text{pu}_Q \quad .$$

The composition law is easier to verify in the uncurried Kleisli formulation. For brevity, denote $r \stackrel{\text{def}}{=} \text{frun}\,(\phi)$; this is a monad morphism of type $P \rightsquigarrow Q$. We need to show that:

$$\big(f^{:A\to P^B} \fatsemi r\big) \diamond_Q \big(g^{:B\to P^C} \fatsemi r\big) \stackrel{?}{=} (f \diamond_P g) \fatsemi r \quad .$$

Express this through uncurried functions and the operations $\diamond_P$ and $\tilde{\diamond}_Q$:

$$\text{uncu}\left(f^{:A\to P^B} \fatsemi r\right) \tilde{\diamond}_Q \text{uncu}\left(g^{:B\to P^C} \fatsemi r\right) \stackrel{?}{=} \text{uncu}\left((f \diamond_P g) \fatsemi r\right) \quad .$$

Uncurrying a function composition such as $f \fatsemi r$ works like this:

$$\begin{aligned}
&\text{uncu}\left(f^{:A\to S\to M^{B\times S}} \fatsemi r^{:P^B\rightsquigarrow Q^B}\right) = a^{:A}\times s^{:S} \to s \triangleright (\underline{a \triangleright f \fatsemi r}) = a^{:A}\times s^{:S} \to s \triangleright (f(a) \triangleright \underline{r}) \\
\text{definition of } r: \quad &= a\times s \to s \triangleright (\underline{f(a) \triangleright (p \to p \fatsemi \phi)}) = a\times s \to s \triangleright f(a) \fatsemi \phi \\
&= \left(a\times s \to s \triangleright f(a)\right) \fatsemi \phi = \text{uncu}\,(f) \fatsemi \phi \quad . \qquad (14.86)
\end{aligned}$$

The right-hand side of the composition law then becomes:

$$\text{uncu}\left((f \diamond_P g) \fatsemi r\right) = \text{uncu}\,(f \diamond_P g) \fatsemi \phi = \left(\text{uncu}\,(f) \diamond_M \text{uncu}\,(g)\right) \fatsemi \phi \quad .$$

We can now proceed to simplify the left-hand side of the composition law:

$$\begin{aligned}
\text{expect to equal uncu}\left((f \diamond_P g) \fatsemi r\right): \quad & \underline{\text{uncu}\left(f^{:A\to P^B} \fatsemi r\right)} \tilde{\diamond}_Q \underline{\text{uncu}\left(g^{:B\to P^C} \fatsemi r\right)} \\
\text{use Eq. (14.86)}: \quad & = \left(\text{uncu}\,(f) \fatsemi \phi\right) \underline{\tilde{\diamond}_Q} \left(\text{uncu}\,(g) \fatsemi \phi\right) \\
\text{definition of } \tilde{\diamond}_Q: \quad & = \left(\text{uncu}\,(f) \fatsemi \phi\right) \diamond_N \left(\text{uncu}\,(g) \fatsemi \phi\right) \\
\text{composition law of } \phi: \quad & = \left(\text{uncu}\,(f) \diamond_M \text{uncu}\,(g)\right) \fatsemi \phi = \text{uncu}\left((f \diamond_P g) \fatsemi r\right) \quad .
\end{aligned}$$

It remains to verify that $\text{frun}\,(\phi^{:M\rightsquigarrow N})$ satisfies the functor laws with respect to the monad parameters M, N. The functor identity law:

$$\text{frun}\,(\text{id}^{:M\rightsquigarrow M}) = p \to p \fatsemi \text{id} = p \to p = \text{id} \quad .$$

The functor composition law:

$$\text{frun}\,(\phi^{:L\rightsquigarrow M}) \fatsemi \text{frun}\,(\chi^{:M\rightsquigarrow N}) = (p \to p \fatsemi \phi) \fatsemi (q \to q \fatsemi \chi) = p \to p \fatsemi \phi \fatsemi \chi = \text{frun}\,(\phi \fatsemi \chi) \quad .$$

Base runner laws The base runner is parameterized by an initial state value s_0:

$$\text{brun}\,(s_0) \stackrel{\text{def}}{=} p^{:S\to M^{A\times S}} \to p(s_0) \triangleright \pi_1^{\uparrow M} \quad .$$

We will check the laws for a fixed s_0. The nondegeneracy law holds:

$$\begin{aligned}
\text{expect to equal } m: \quad & m \triangleright \text{flift} \triangleright \text{brun}\,(s_0) = m \triangleright \text{flift} \triangleright (p \to p(s_0) \triangleright \pi_1^{\uparrow M}) = s_0 \triangleright (\underline{m \triangleright \text{flift}}) \triangleright \pi_1^{\uparrow M} \\
\text{use Eq. (14.84)}: \quad & = m \triangleright (a \to a\times s_0)^{\uparrow M} \triangleright \pi_1^{\uparrow M} = m \triangleright (a \to a) = m \quad .
\end{aligned}$$

We now turn to the monadic morphism laws. The identity law holds:

$$\begin{aligned}
\text{expect to equal pu}_M: \quad & \text{pu}_P \fatsemi \text{brun}\,(s_0) = \left(a \to s \to (a\times s) \triangleright \text{pu}_M\right) \fatsemi (p \to p(s_0) \triangleright \pi_1^{\uparrow M}) \\
\text{compute composition}: \quad & = a \to (a\times s_0) \triangleright \text{pu}_M \triangleright \pi_1^{\uparrow M} = a \to (a\times s_0) \triangleright \underline{\text{pu}_M \fatsemi \pi_1^{\uparrow M}} \\
\text{naturality law of pu}_M: \quad & = a \to (a\times s_0) \triangleright \pi_1 \fatsemi \text{pu}_M = a \to a \triangleright \text{pu}_M = \text{pu}_M \quad .
\end{aligned}$$

The composition law is quicker to check in the Kleisli formulation:

$$\left(f^{:A\to S\to M^{B\times S}} \fatsemi \text{brun}\,(s_0)\right) \diamond_M \left(g^{:B\to S\to M^{C\times S}} \fatsemi \text{brun}\,(s_0)\right) \stackrel{?}{=} (f \diamond_P g) \fatsemi \text{brun}\,(s_0) \quad .$$

Since it is easier to define $\diamond_P$ in terms of the uncurried Kleisli functions, let us express the composition $f \fatsemi \text{brun}\,(s_0)$ in terms of the uncurried version of f:

$$\begin{aligned}
&f \fatsemi \text{brun}\,(s_0) = a \to a \triangleright f \triangleright (p \to p(s_0) \triangleright \pi_1^{\uparrow M}) = a \to f(a)(s_0) \triangleright \pi_1^{\uparrow M} \\
&= a \to (a\times s_0) \triangleright \text{uncu}\,(f) \fatsemi \pi_1^{\uparrow M} = (a \to a\times s_0) \fatsemi \text{uncu}\,(f) \fatsemi \pi_1^{\uparrow M} \quad . \qquad (14.87)
\end{aligned}$$

We can now write the left-hand side of the composition law as:

$$\begin{aligned}
& \left(f \,⨾\, \text{brun}\,(s_0)\right) \diamond_M \left(g \,⨾\, \text{brun}\,(s_0)\right) = \left((a \to a \times s_0) \,⨾\, \text{uncu}\,(f) \,⨾\, \underline{\pi_1^{\uparrow M}}\right) \underline{\diamond_M} \left(g \,⨾\, \text{brun}\,(s_0)\right) \\
\text{naturality of } \diamond_M : \quad & = (a \to a \times s_0) \,⨾\, \text{uncu}\,(f) \diamond_M \left(\pi_1 \,⨾\, g \,⨾\, \text{brun}\,(s_0)\right) \\
\text{use Eq. (14.87)} : \quad & = (a \to a \times s_0) \,⨾\, \text{uncu}\,(f) \diamond_M \left(\pi_1 \,⨾\, (b \to b \times s_0) \,⨾\, \text{uncu}\,(g) \,⨾\, \pi_1^{\uparrow M}\right) \quad .
\end{aligned}$$

The right-hand side of the law is:

$$\begin{aligned}
& (f \diamond_P g) \,⨾\, \text{brun}\,(s_0) = (a \to a \times s_0) \,⨾\, \text{uncu}\,(f \diamond_P g) \,⨾\, \pi_1^{\uparrow M} \\
\text{definition of } \diamond_P : \quad & = (a \to a \times s_0) \,⨾\, \left(\text{uncu}\,(f) \diamond_M \text{uncu}\,(g)\right) \underline{\,⨾\, \pi_1^{\uparrow M}} \\
\text{right naturality law of } \diamond_M : \quad & = (a \to a \times s_0) \,⨾\, \text{uncu}\,(f) \diamond_M \left(\text{uncu}\,(g) \,⨾\, \pi_1^{\uparrow M}\right) \quad .
\end{aligned}$$

We find that the two sides of the law are *not* equal. The difference is the presence of a function $\pi_1 \,⨾\, (b \to b \times s_0)$ of type $B \times S \to B \times S$. That function replaces the updated state by the fixed initial state s_0 before applying the uncurried function g. This replacement erases the updated state, which leads to an incorrect composition of effects in the `StateT` monad.

If we set $M = \text{Id}$, we will obtain the same violation for the `State` monad's runner.

Monadic naturality laws To verify the monadic naturality of `flift`, apply to an arbitrary $m^{:M^A}$:

$$\begin{aligned}
\text{expect to equal } m \triangleright \phi \triangleright \text{flift}^N : \quad & m \triangleright \text{flift}^M \triangleright \text{frun}\,(\phi) = m \triangleright \text{flift}^M \triangleright (p \to p \,⨾\, \phi) = (m \triangleright \text{flift}^M) \,⨾\, \phi \\
\text{use Eq. (14.84)} : \quad & = \left(s \to m \triangleright (a \to a \times s)^{\uparrow M}\right) \,⨾\, \phi = s \to m \triangleright \underline{(a \to a \times s)^{\uparrow M} \,⨾\, \phi} \\
\text{naturality law of } \phi : \quad & = s \to m \triangleright \phi \,⨾\, (a \to a \times s)^{\uparrow N} = m \triangleright \phi \triangleright \text{flift}^N \quad .
\end{aligned}$$

To verify the monadic naturality of `brun`:

$$\begin{aligned}
\text{expect to equal brun}\,(s_0) \,⨾\, \phi : \quad & \text{frun}\,(\phi) \,⨾\, \text{brun}\,(s_0) = (p \to p \,⨾\, \phi) \,⨾\, (p \to s_0 \triangleright p \,⨾\, \pi_1^{\uparrow N}) \\
\text{compute composition} : \quad & = p \to s_0 \triangleright p \,⨾\, \underline{\phi \,⨾\, \pi_1^{\uparrow N}} = \underline{p \to s_0 \triangleright p \,⨾\, \pi_1^{\uparrow M}} \,⨾\, \phi = \text{brun}\,(s_0) \,⨾\, \phi \quad .
\end{aligned}$$

14.8.2 Transformer for the continuation monad. Proofs

The continuation monad transformer's type constructor `ContT[R, M[_], *]` is defined by:

$$T_{\text{Cont}}^{M,A} \stackrel{\text{def}}{=} (A \to M^R) \to M^R \quad .$$

Unlike most other monad transformers, T_{Cont} is not covariant in the foreign monad M: the type $T_{\text{Cont}}^{M,A}$ contains M both in covariant and contravariant positions. (In the language of categories, T_{Cont} is not a functor in the category of monads.) As a result, we cannot define the foreign runner or the base lift functions. This limits the usability of the continuation monad transformer.

Another limitation comes from the fact that the `Cont` monad does not have a fully parametric runner of type $\forall A. \text{Cont}^{R,A} \to A$. The derivation techniques developed in this book apply only to fully parametric code. The runners constructed in Section 10.1.9 are not fully parametric and will terminate only for values of type $\text{Cont}^{R,A}$ that eventually call their callbacks (of type $A \to R$). Accordingly, the transformer `ContT` does not have fully parametric base runners, and we will not verify their laws. Nevertheless, we will prove that the remaining laws of `ContT` hold.

Monad laws The transformed monad T_{Cont}^M differs from the continuation monad `Cont` only in the type of the result (R), replacing $(A \to R) \to R$ by $(A \to M^R) \to M^R$. The monad's methods work the same way for all result types, so T_{Cont}^M is a monad for all monads M. (This would remain true even if the type constructor M were not a functor!)

Identity laws If we set $M = \text{Id}$, the type of $T_{\text{Cont}}^{M,A}$ is equivalent to the base monad, $(A \to R) \to R$.

Lifting laws The foreign lift function (`flift`) is defined by:

$$\text{flift} : M^A \to (A \to M^R) \to M^R \quad , \qquad \text{flift} \stackrel{\text{def}}{=} \text{flatMap}_M = m^{:M^A} \to f^{:A\to M^R} \to m \triangleright \text{flm}_M(f) \quad .$$

To verify that `flift` is a monad morphism, we use the flipped Kleisli formulation for T_{Cont}. In that formulation, Kleisli functions of type $A \to T^{M,B}_{\text{Cont}} = A \to (B \to M^R) \to M^R$ are flipped into the type signature $(B \to M^R) \to A \to M^R$. For the `Cont` monad, the flipped Kleisli composition is equal to the ordinary *backward* function composition (see Statement 10.2.7.1):

$$f^{:(B\to M^R)\to A\to M^R} \tilde{\diamond}_T \, g^{:(C\to M^R)\to B\to M^R} = (g \,⨾\, f)^{:(C\to M^R)\to A\to M^R} \quad . \tag{14.88}$$

Let us also compute the result of flipping a lifted M-Kleisli function:

$$\text{flip}\,(f^{:A\to M^B} ⨾ \text{flift}) = p^{:B\to M^R} \to a^{:A} \to f(a) \triangleright \text{flm}_M(p) = p \to f ⨾ \text{flm}_M(p) = p \to f \diamond_M p \quad . \tag{14.89}$$

The identity law of `flift` is now easy to verify. The flipped `pure` method for T_{Cont} is:

$$\tilde{\text{pu}}_T : (A \to M^R) \to A \to M^R \quad , \qquad \tilde{\text{pu}}_T = \text{id}^{A\to M^R} \quad .$$

To show that the law holds, we write:

$$\text{expect to equal } \tilde{\text{pu}}_T : \quad \text{flip}\,(\text{pu}_M ⨾ \text{flift}) = p \to \underline{\text{pu}_M \diamond_M p} = p \to p = \text{id} = \tilde{\text{pu}}_T \quad .$$

The composition law in the ordinary Kleisli formulation is:

$$\left(f^{:A\to M^B} ⨾ \text{flift}\right) \diamond_T \left(g^{:B\to M^C} ⨾ \text{flift}\right) \stackrel{?}{=} (f \diamond_M g) ⨾ \text{flift} \quad .$$

To transform the composition law to the flipped Kleisli formulation, we write:

$$\text{flip}\,(f ⨾ \text{flift}) \tilde{\diamond}_T \text{flip}\,(g ⨾ \text{flift}) \stackrel{?}{=} \text{flip}\left((f \diamond_M g) ⨾ \text{flift}\right) \quad .$$

We complete the proof by using Eqs. (14.86)–(14.87) and the associativity of $\diamond_M$:

$$\begin{aligned}&\text{flip}\,(f ⨾ \text{flift}) \tilde{\diamond}_T \text{flip}\,(g ⨾ \text{flift}) = (q \to g \diamond_M q) ⨾ (p \to f \diamond_M p) = q \to \underline{f \diamond_M (g \diamond_M q)} \\ &= q \to (f \diamond_M g) \diamond_M q = \text{flip}\left((f \diamond_M g) ⨾ \text{flift}\right) \quad .\end{aligned}$$

14.8.3 Transformer for the generalized `Search` monad. Proofs

The generalized `Search` monad is defined, for a fixed monad L and a fixed type Q, by:

$$\text{Search}^{L,Q,A} \stackrel{\text{def}}{=} (A \to L^Q) \to L^A \quad .$$

The ordinary `Search` monad is obtained with $L \stackrel{\text{def}}{=} \text{Opt}$ and $Q \stackrel{\text{def}}{=} \mathbb{1}$.

Statement 14.8.3.1 The transformer type for the generalized `Search` monad is given by:

$$T^{M,A}_{\text{Search}} \stackrel{\text{def}}{=} (A \to T^{M,Q}_L) \to T^{M,A}_L \quad ,$$

where L is any fixed monad and T_L is the monad L's transformer. Like the continuation monad, the `Search` monad's transformer has no base lift, foreign runner, or base runner functions.

Proof We assume that T_L already satisfies all the necessary transformer laws.

Monad laws The monad laws hold for $\text{Search}^{L,Q,A}$ due to Statement 10.2.8.9, where we use the L-filterable contrafunctor $A \to L^Q$ as H^A (see Statement 10.2.8.8). Similarly, we find that the monad laws hold for $T^{M,A}_{\text{Search}}$ if we use the T^M_L-filterable contrafunctor $H^A \stackrel{\text{def}}{=} A \to T^{M,Q}_L$, where we assume that T^M_L is a lawful monad. We will use this definition of H^A in the proof below. For brevity, we will write simply T instead of T^M_{Search} (the foreign monad M is fixed throughout the derivations).

The monad methods of $T^M_{\text{Search}} \stackrel{\text{def}}{=} T$ are defined according to Statement 10.2.8.9:

$$\text{pu}_T \stackrel{\text{def}}{=} a^{:A} \to (_^{:H^A}) \to \text{pu}_{T_L}(a) \quad ,$$
$$f^{:H^B \to A \to T^{M,B}_L} \tilde{\diamond}_T g^{:H^C \to A \to T^{M,C}_L} \stackrel{\text{def}}{=} h^{:H^C} \to f\left(h \triangleright \text{lift}_{T^M_L,H}(g(h))\right) \diamond_{T_L} g(h) \quad ,$$

where we defined : $\text{lift}_{T^M_L,H} \stackrel{\text{def}}{=} f^{:A \to T^{M,B}_L} \to h^{:B \to T^{M,Q}_L} \to f \diamond_{T^M_L} h \quad .$

Derivations are shorter if we use the flipped Kleisli composition operation $\tilde{\diamond}_T$ and the (ordinary) Kleisli composition $\diamond_{T_L}$ of the monad T^M_L.

Identity laws We need to establish monad isomorphisms between the monads $R^A \stackrel{\text{def}}{=} (A \to L^Q) \to L^A$ and $S^A \stackrel{\text{def}}{=} (A \to T^{\text{Id},Q}_L) \to T^{\text{Id},A}_L$. Denote for brevity $N^A \stackrel{\text{def}}{=} T^{\text{Id},A}_L$. Our goal is to implement monad morphisms p and q:

$$p : \left((A \to L^Q) \to L^A\right) \to (A \to N^Q) \to N^A \quad , \qquad q : \left((A \to N^Q) \to N^A\right) \to (A \to L^Q) \to L^A \quad ,$$

such that $p \,⨾\, q = \text{id}$ and $q \,⨾\, p = \text{id}$. We may assume that the identity laws already hold for the transformer T_L. These laws imply that there exist two monad morphisms,

$$r^A : L^A \to N^A \quad , \qquad s^A : N^A \to L^A \quad ,$$

satisfying the laws $r \,⨾\, s = \text{id}$ and $s \,⨾\, r = \text{id}$. We may use r and s as given functions.

The listed requirements use L and N in the same way. So, the implementations of p and q will be the same up to exchanging r with s, and it will be sufficient to prove one of the laws (say, $p \,⨾\, q = \text{id}$).

To find the code for p, we use typed holes:

$$p \stackrel{\text{def}}{=} u^{:(A \to L^Q) \to L^A} \to v^{:A \to N^Q} \to ???^{:N^A} = u \to v \to (???^{:A \to L^Q}) \triangleright u \triangleright r^A$$
$$= u \to v \to (???^{:A \to N^Q} \,⨾\, s^Q) \triangleright u \triangleright r = u \to v \to (v \,⨾\, s) \triangleright u \,⨾\, r \quad .$$

The code for q is found by exchanging r and s in the code of p:

$$q \stackrel{\text{def}}{=} t^{:(A \to N^Q) \to N^A} \to w^{:A \to L^Q} \to (w \,⨾\, r) \triangleright t \,⨾\, s \quad .$$

To verify that $p \,⨾\, q = \text{id}$, we use the assumption $r \,⨾\, s = \text{id}$:

$$p \,⨾\, q = \left(u \to \underline{v \to (v \,⨾\, s) \triangleright u \,⨾\, r}\right) ⨾ \left(\underline{t} \to w \to (w \,⨾\, r) \triangleright \underline{t \,⨾\, s}\right)$$

compute composition : $= u \to w \to \underline{(w \,⨾\, r) \triangleright \left(v \to (v \,⨾\, s) \triangleright u \,⨾\, r\right)} ⨾ s$

apply function : $= u \to w \to (w \,⨾\, \underline{r \,⨾\, s}) \triangleright u \,⨾\, \underline{r \,⨾\, s} = u \to (\underline{w \to w \triangleright u}) = u \to u = \text{id} \quad .$

A separate proof for $q \,⨾\, p = \text{id}$ is not needed.

It remains to prove that p is a monad morphism (the same will then hold for q).

To verify the identity law:

expect to equal $a \triangleright \text{pu}_S$: $a \triangleright \text{pu}_R \,⨾\, p = \left((_^{:A \to L^Q}) \to a \triangleright \text{pu}_L\right) \triangleright p = \underline{v} \to a \triangleright \underline{\text{pu}_L \,⨾\, r}$

v is not used : $= _ \to a \triangleright \text{pu}_N = a \triangleright \text{pu}_S \quad .$

To verify the composition law, we use the flipped Kleisli formulation. The flipped Kleisli composition for the monad R is defined by:

$$f^{:(B\to L^Q)\to A\to L^B}\tilde{\diamond}_T\, g^{:(C\to L^Q)\to B\to L^C} \stackrel{\text{def}}{=} h^{:H^C} \to f\big(h \triangleright \text{lift}_{L,H}(g(h))\big) \diamond_L g(h) \quad ,$$

and similarly for the monad S (except for using N instead of L and a corresponding change in the definition of H). The next step is to modify p into a more convenient function, $\tilde{p}$, that directly transforms flipped Kleisli functions in the monad R into those in S:

$$\begin{aligned}
&\tilde{p} : \big((B \to L^Q) \to A \to L^B\big) \to (B \to N^Q) \to A \to N^B \quad ,\\
&\tilde{p} \stackrel{\text{def}}{=} k^{:(B\to L^Q)\to A\to L^B} \to h^{:B\to N^Q} \to k(h \fatsemi s) \fatsemi r \quad .
\end{aligned}$$

We may now rewrite the composition law using $\tilde{p}$:

$$\tilde{p}\big(f^{:(B\to L^Q)\to A\to L^B}\tilde{\diamond}_R\, g^{:(C\to L^Q)\to B\to L^C}\big) \stackrel{?}{=} \tilde{p}(f)\tilde{\diamond}_S\tilde{p}(g) \quad .$$

Rewrite the left-hand side:

$$\begin{aligned}
&\tilde{p}\big(f\tilde{\diamond}_R g\big) = \underline{\tilde{p}}\big(h \to f\big(h \triangleright \text{lift}_{L,H}(g(h))\big) \diamond_L g(h)\big)\\
\text{definition of } \tilde{p}: \quad &= h \to \big(f\big((h \fatsemi s) \triangleright \text{lift}_{L,H}(g(h \fatsemi s))\big)\underline{\diamond_L} g(h \fatsemi s)\big) \underline{\fatsemi r}\\
\text{composition law of } r: \quad &= h \to \big(f\big((h \fatsemi s) \triangleright \text{lift}_{L,H}(g(h \fatsemi s))\big) \fatsemi r\big) \diamond_N \big(g(h \fatsemi s) \fatsemi r\big) \quad .
\end{aligned}$$

Rewrite the right-hand side:

$$\begin{aligned}
&\tilde{p}\,(f)\,\tilde{\diamond}_S\tilde{p}\,(g) = \big(h \to f(h \fatsemi s) \fatsemi r\big)\tilde{\diamond}_S\big(h \to g(h \fatsemi s) \fatsemi r\big)\\
&= h \to \big(h \to f(h \fatsemi s) \fatsemi r\big)(h \triangleright \text{lift}_{N,H}(\big(h \to g(h \fatsemi s) \fatsemi r\big)(h))) \diamond_N \big(h \to g(h \fatsemi s) \fatsemi r\big)(h)\\
&= h \to \big(f((h \triangleright \text{lift}_{N,H}(g(h \fatsemi s) \fatsemi r)) \fatsemi s) \fatsemi r\big) \diamond_N \big(g(h \fatsemi s) \fatsemi r\big) \quad .
\end{aligned}$$

The remaining difference between the two sides is:

$$(h \fatsemi s) \triangleright \text{lift}_{L,H}(g(h \fatsemi s)) \stackrel{?}{=} \big(h \triangleright \text{lift}_{N,H}(g(h \fatsemi s) \fatsemi r)\big) \fatsemi s \quad . \tag{14.90}$$

To proceed, we need to use the code of the lifting functions $\text{lift}_{L,H}$ and $\text{lift}_{N,H}$:

$$\begin{aligned}
&\text{lift}_{L,H} : (A \to L^B) \to (B \to L^Q) \to A \to L^Q \quad , \qquad g^{:B\to L^Q} \triangleright \text{lift}_{L,H}(f^{:A\to L^B}) = f \diamond_L g \quad ,\\
&\text{lift}_{N,H} : (A \to N^B) \to (B \to N^Q) \to A \to N^Q \quad , \qquad g^{:B\to N^Q} \triangleright \text{lift}_{N,H}(f^{:A\to N^B}) = f \diamond_N g \quad .
\end{aligned}$$

Substituting the code of those functions into both sides of Eq. (14.90), we get:

$$\begin{aligned}
\text{left-hand side}: \quad &(h \fatsemi s) \triangleright \text{lift}_{L,H}(g(h \fatsemi s)) = g(h \fatsemi s) \diamond_L (h \fatsemi s)\\
&\stackrel{?}{=} \big(h \triangleright \text{lift}_{N,H}(g(h \fatsemi s) \fatsemi r)\big) \fatsemi s = \big((g(h \fatsemi s) \fatsemi r) \underline{\diamond_N} h\big) \underline{\fatsemi s}\\
\text{composition law of } s: \quad &= \big(g(h \fatsemi s) \fatsemi \underline{r \fatsemi s}\big) \diamond_L (h \fatsemi s) = g(h \fatsemi s) \diamond_L (h \fatsemi s) \quad .
\end{aligned}$$

Both sides are now equal.

Lifting laws The `flift` function of the `Search` monad transformer is defined by:

$$\text{flift}: \quad M^A \to (A \to T_L^{M,Q}) \to T_L^{M,A} \quad , \qquad \text{flift} \stackrel{\text{def}}{=} m^{:M^A} \to \big(_^{:H^A}\big) \to m \triangleright \text{flift}_L \quad ,$$

where $\text{flift}_L : M^A \to T_L^{M,A}$ is the transformer T_L's foreign lift function.

To show that `flift` is a monad morphism, begin with the identity law:

$$\begin{aligned}
\text{expect to equal } a \triangleright \text{pu}_T : \quad & a^{:A} \triangleright \text{pu}_M \,⨾\, \text{flift} = \left(_^{:A \to T_L^{M,Q}}\right) \to a \triangleright \underline{\text{pu}_M \,⨾\, \text{flift}_L} \\
\text{identity law of flift}_L : \quad & = \left(_^{:A \to T_L^{M,Q}}\right) \to a \triangleright \text{pu}_{T_L} \\
\text{definition of pu}_T : \quad & = a \triangleright \text{pu}_T \quad .
\end{aligned}$$

To verify the composition law, we note that `flift` always produces functions that ignore their argument of type H^A. For such values, the Kleisli composition $\tilde{\diamond}_T$ may be simplified: If we know that the functions $f^{:H^B \to A \to T_L^{M,B}}$ and $g^{:H^C \to A \to T_L^{M,C}}$ ignore their first arguments, we may write

$$f \,\tilde{\diamond}_T\, g = k^{:H^C} \to f(...) \diamond_{T_L} g(...) = \left(_^{:H^C}\right) \to f(...) \diamond_{T_L} g(...) \quad .$$

So, the Kleisli composition also always produces functions that ignore their first arguments. We can then simplify the formulas, since $\diamond_T$ reduces to the Kleisli composition $\diamond_{T_L}$ of the monad T_L^M. To make use of this simplification, we need to flip the arguments of Kleisli functions. First, we find, for any function $p^{:A \to M^B}$:

$$\text{flip}\,(p^{:A \to M^B} \,⨾\, \text{flift}) = \left(_^{:H^B}\right) \to \underline{a^{:A} \to a} \triangleright p \triangleright \text{flift}_L = \left(_^{:H^B}\right) \to p \,⨾\, \text{flift}_L \quad .$$

Now we can show that `flift`'s composition law holds:

$$\begin{aligned}
\text{expect } (p \diamond_M q) \,⨾\, \text{flift} : \quad & \left(p^{:A \to M^B} \,⨾\, \text{flift}\right) \diamond_T \left(q^{:B \to M^C} \,⨾\, \text{flift}\right) = \text{flip}\left(\underline{\text{flip}\,(p \,⨾\, \text{flift})} \,\tilde{\diamond}_T\, \underline{\text{flip}\,(q \,⨾\, \text{flift})}\right) \\
& = \text{flip}\left((_ \to p \,⨾\, \text{flift}_L) \,\tilde{\diamond}_T\, (_ \to q \,⨾\, \text{flift}_L)\right) = \text{flip}\left(_ \to \underline{(p \,⨾\, \text{flift}_L) \diamond_{T_L} (q \,⨾\, \text{flift}_L)}\right) \\
\text{flift}_L\text{'s composition law} : \quad & = \text{flip}\left(\underline{_ \to (p \diamond_M q) \,⨾\, \text{flift}_L}\right) = \underline{\text{flip}}\left(\text{flip}\left((p \diamond_M q) \,⨾\, \text{flift}\right)\right) \\
\text{use flip} \,⨾\, \text{flip} = \text{id} : \quad & = (p \diamond_M q) \,⨾\, \text{flift} \quad .
\end{aligned}$$

14.8.4 The codensity monad and its transformer. Proofs

For any type constructor F, the **codensity monad**[20] **on** F is defined by:

$$\text{Cod}^{F,A} \stackrel{\text{def}}{=} \forall R.\, (A \to F^R) \to F^R \quad .$$

Before considering its transformer, we need to establish some properties of this monad.

Note the similarity of $\text{Cod}^{F,A}$ with the continuation monad $\text{Cont}^{F^R,A} = (A \to F^R) \to F^R$ whose "return type" is F^R. The difference is that $\text{Cod}^{F,A}$ imposes a universal quantifier ($\forall R$) on the type parameter R:

$$\text{Cod}^{F,A} = \forall R.\, \text{Cont}^{F^R,A} \quad .$$

Many of the properties of the continuation monad continue to hold after adding $\forall R$. In general, adding a type quantifier preserves monad laws:

Statement 14.8.4.1 Assume a monad $K^{R,\bullet}$ with an extra type parameter R. In other words, $K^{R,A}$ is a lawful monad with respect to the type parameter A and with a fixed R. Then **(a)** $L^A \stackrel{\text{def}}{=} \forall R.\, K^{R,A}$ is also a lawful monad, and **(b)** for any fixed R, there is a monad morphism of type $L^\bullet \rightsquigarrow K^{R,\bullet}$.

[20] See, e.g., the talk `https://begriffs.com/posts/2016-02-04-difference-lists-and-codennsity.html` for applications of the codensity monad in functional programming and for more references.

Proof **(a)** We need to define the monad L's methods and verify the laws. The idea is to add the quantifier ($\forall R$) to monad K's methods. (For brevity, we will sometimes write K to mean $K^{R,\bullet}$.) Begin with the method `pure`:

$$\text{pu}_L \stackrel{\text{def}}{=} \forall R.\, \text{pu}_{K^{R,\bullet}} = \forall R.\, a^{:A} \to \text{pu}_{K^{R,\bullet}}(a) = a^{:A} \to \forall R.\, \text{pu}_{K^{R,\bullet}}(a) \quad .$$

Here, we may move the quantifier as $\forall R.\, a^{:A} \to (...)^R = a^{:A} \to \forall R.\, (...)^R$ because the type parameter A does not depend on R.

To define the `flatMap` method for L, we add quantifiers to the `flatMap` method of K:

$$\text{flm}_K(f^{:A\to K^{R,B}})(d^{:K^{R,A}}) : K^{R,B} \quad ,$$

and to its two curried arguments (f and d). To write the resulting code, we need to use distinct type parameters R, S, T for the arguments f, d and for the return value respectively:

$$\text{flm}_L(f^{:\forall R.\, A\to K^{R,B}}) \stackrel{\text{def}}{=} d^{:\forall S.\, K^{S,A}} \to \forall T.\, \text{flm}_K(f^T)(d^T) \quad .$$

The arguments f and d of flm_L have their own type parameters (R and S), and we *assign* those type parameters to the type T in the expression $\text{flm}_K(f^T)(d^T)$.

To verify the monad laws for L, we use the Kleisli formulation. To define the Kleisli composition ($\diamond_L$) for L, we use two different universally quantified types (R, S) to avoid confusion:

$$f^{:A\to L^B} \diamond_L g^{:B\to L^C} = f^{:\forall R.\, A\to K^{R,B}} \diamond_L g^{:\forall S.\, B\to K^{S,C}} \stackrel{\text{def}}{=} \forall T.\, f^T \diamond_K g^T \quad .$$

As in the definition of `flatMap`, we need to assign the type parameters R and S to T.

To verify the identity laws, we assume that the same laws hold for the monad K:

$$\text{expect to equal } g: \quad \text{pu}_L \diamond_L g = \forall T.\, \text{pu}_{K^{T,\bullet}} \diamond_K g^T = \forall T.\, g^T = g \quad ,$$
$$\text{expect to equal } f: \quad f \diamond_L \text{pu}_L = \forall T.\, f^T \diamond_K \text{pu}_{K^{T,\bullet}} = \forall T.\, f^T = f \quad .$$

To verify the associativity law:

$$\begin{aligned}
\text{left-hand side}: \quad & (f \diamond_L g) \diamond_L h = \left(\forall T.\, f^T \diamond_K g^T\right) \diamond_L h \\
& = \forall S.\, \left(\forall T.\, f^T \diamond_K g^T\right)^S \diamond_K h^S = \forall S.\, \left(f^S \diamond_K g^S\right) \diamond_K h^S \quad , \\
\text{right-hand side}: \quad & f \diamond_L (g \diamond_L h) = \forall S.\, f^S \diamond_K (g \diamond_L h)^S \\
& = \forall S.\, f^S \diamond_K \left(\forall T.\, g^T \diamond_K h^T\right)^S = \forall S.\, f^S \diamond_K (g^S \diamond_K h^S) \quad .
\end{aligned}$$

The two sides are equal due to the associativity of the Kleisli composition ($\diamond_K$) of the monad K.

(b) The monad morphism $\phi : L^A \to K^{R,A}$ is defined by setting the quantified type parameter T in $L^A \stackrel{\text{def}}{=} \forall T.\, K^{T,A}$ as $T = R$. The action of ϕ on a Kleisli function $f^{:\forall T.\, A\to K^{T,B}}$ can be written as $f \fatsemi \phi = f^R$. To verify that ϕ is monad morphism, begin with the identity law:

$$\text{expect to equal } \text{pu}_K: \quad \text{pu}_L \fatsemi \phi = (\forall T.\, \text{pu}_{K^{T,\bullet}}) \fatsemi \phi = (\forall T.\, \text{pu}_{K^{T,\bullet}})^R = \text{pu}_{K^{R,\bullet}} = \text{pu}_K \quad .$$

To verify the composition law, we show that its two sides are equal:

$$\begin{aligned}
\text{left-hand side}: \quad & \left(f^{:\forall R.\, A\to K^{R,B}} \diamond_L g^{:\forall S.\, B\to K^{S,C}}\right) \fatsemi \phi = (\forall T.\, f^T \diamond_K g^T)^R = f^R \diamond_K g^R \quad , \\
\text{right-hand side}: \quad & (f \fatsemi \phi) \diamond_K (g \fatsemi \phi) = f^R \diamond_K g^R \quad .
\end{aligned}$$

Statement 14.8.4.2 $\text{Cod}^{F,\bullet}$ is a monad for any type constructor F (not necessarily a functor).

Proof By Statement 14.8.4.1, adding a quantifier ($\forall R$) to the continuation monad $\text{Cont}^{F^R,A}$ gives again a lawful monad. □

The next statement shows that $\text{Cod}^{F,\bullet}$ includes the effects of F when F is a monad.

Statement 14.8.4.3 **(a)** If F is itself a monad then there exist functions $\text{inC} : F^\bullet \rightsquigarrow \text{Cod}^{F,\bullet}$ and $\text{outC} : \text{Cod}^{F,\bullet} \rightsquigarrow F^\bullet$, such that $\text{inC} \,{}^{\circ}\!{;}\, \text{outC} = \text{id}$ (so, `inC` is an injection and `outC` a surjection).

(b) The function `inC` is a monad morphism $F^\bullet \rightsquigarrow \text{Cod}^{F,\bullet}$ (while `outC` is not a monad morphism).

(c) The functions `inC` and `outC` are *not* isomorphisms between the types F^A and $\text{Cod}^{F,A}$.

Proof **(a)** Begin by implementing the functions `inC` and `outC`:

$$\text{inC} : F^A \to \forall R.\,(A \to F^R) \to F^R \quad , \qquad \text{inC}^{R,A} \stackrel{\text{def}}{=} f^{:F^A} \to k^{:A\to F^R} \to f \triangleright \text{flm}_F(k) \quad ,$$
$$\text{outC} : \left(\forall R.\,(A \to F^R) \to F^R\right) \to F^A \quad , \qquad \text{outC}^A \stackrel{\text{def}}{=} d^{:\forall R.\,(A\to F^R)\to F^R} \to d^A(\text{pu}_F^{:A\to F^A}) \quad .$$

Note that `inC` is the same as F's `flatMap`, or flm_F with flipped arguments.

To verify the nondegeneracy law, compute:

$$\begin{aligned}
\text{expect to equal } f : \quad & f^{:F^A} \triangleright \text{inC} \,{}^{\circ}\!{;}\, \text{outC} = (k \to f \triangleright \text{flm}_F(k)) \triangleright \text{outC} \\
& = (k \to f \triangleright \text{flm}_F(k))(\text{pu}_F) = f \triangleright \underline{\text{flm}_F(\text{pu}_F)} \\
\text{right identity law of } F : \quad & = f \triangleright \text{id} = f \quad .
\end{aligned}$$

(b) To show that `inC` is a monad morphism, we use the Kleisli composition:

$$f^{:\forall R.\,A\to(B\to F^R)\to F^R} \diamond_{\text{Cod}} g^{:\forall S.\,B\to(C\to F^S)\to F^S} \stackrel{\text{def}}{=} \forall T.\,a^{:A} \to k^{:C\to F^T} \to f^T(a)\left(b^{:B} \to g^T(b)(k)\right) \quad .$$

To verify the identity law of `inC`:

$$\begin{aligned}
\text{expect to equal } a \triangleright \text{pu}_{\text{Cod}} : \quad & a^{:A} \triangleright \text{pu}_F \,{}^{\circ}\!{;}\, \text{inC} = \forall R.\,g^{:A\to F^R} \to a \triangleright \underline{\text{pu}_F \triangleright \text{flm}_F}(g) \\
\text{left identity law of } F : \quad & = \forall R.\,g^{:A\to F^R} \to a \triangleright g = \text{pu}_{\text{Cod}}(a) \quad .
\end{aligned}$$

The composition law of `inC` is written as:

$$(f^{:A\to F^B} \diamond_F g^{:B\to F^C}) \,{}^{\circ}\!{;}\, \text{inC} \stackrel{?}{=} (f \,{}^{\circ}\!{;}\, \text{inC}) \diamond_{\text{Cod}} (g \,{}^{\circ}\!{;}\, \text{inC}) \quad .$$

Apply both sides to an arbitrary value $a^{:A}$ and simplify the results:

$$\begin{aligned}
\text{left-hand side} : \quad & a^{:A} \triangleright (f \diamond_F g) \,{}^{\circ}\!{;}\, \text{inC} = k \to a \triangleright (f \diamond_F g) \,{}^{\circ}\!{;}\, \text{flm}_F(k) \\
\text{definition of } \diamond_F : \quad & = k \to a \triangleright f \,{}^{\circ}\!{;}\, \text{flm}_F(g) \,{}^{\circ}\!{;}\, \text{flm}_F(k) \quad , \\
\text{right-hand side} : \quad & a^{:A} \triangleright \left((f \,{}^{\circ}\!{;}\, \text{inC}) \diamond_{\text{Cod}} (g \,{}^{\circ}\!{;}\, \text{inC})\right) = k \to (f \,{}^{\circ}\!{;}\, \text{inC})^T(a)\left(b \to (g \,{}^{\circ}\!{;}\, \text{inC})^T(b)(k)\right) \quad .
\end{aligned}$$

For clarity, let us simplify the expression $(f \,{}^{\circ}\!{;}\, \text{inC})^T(a)(x)$ separately:

$$(f \,{}^{\circ}\!{;}\, \text{inC})^T(a)(x) = x \triangleright \left(a \triangleright f \triangleright \text{inC}\right)^T = x \triangleright \left(k \to a \triangleright f \triangleright \text{flm}_F(k)\right) = a \triangleright f \triangleright \text{flm}_F(x) \quad .$$

The same simplification applies to the expression $(g \,{}^{\circ}\!{;}\, \text{inC})^T(b)(k)$. So, the right-hand side becomes:

$$\begin{aligned}
\text{right-hand side} : \quad & k \to (f \,{}^{\circ}\!{;}\, \text{inC})^T(a)\left(\underline{b \to b \triangleright} g \,{}^{\circ}\!{;}\, \text{flm}_F(k)\right) = k \to \underline{(f \,{}^{\circ}\!{;}\, \text{inC})(a)}(g \,{}^{\circ}\!{;}\, \text{flm}_F(k)) \\
& = k \to a \triangleright f \triangleright \underline{\text{flm}_F(g \,{}^{\circ}\!{;}\, \text{flm}_F(k))} = k \to a \triangleright f \triangleright \text{flm}_F(g) \,{}^{\circ}\!{;}\, \text{flm}_F(k) \quad .
\end{aligned}$$

The two sides of the law are now equal.

As for the function `outC`, it does satisfy the identity law of monad morphisms:

$$\begin{aligned}
\text{expect to equal } a \triangleright \text{pu}_F : \quad & a^{:A} \triangleright \text{pu}_{\text{Cod}} \,{}^{\circ}\!{;}\, \text{outC} = \left(\forall R.\,g^{:A\to F^R} \to a \triangleright g\right) \triangleright \text{outC} \\
\text{definition of outC} : \quad & = \left(\forall R.\,g^{:A\to F^R} \to a \triangleright g\right)^A(\text{pu}_F) = a \triangleright \text{pu}_F \quad .
\end{aligned}$$

But the composition law does not hold in general:

$$\left(f^{:\forall R.\,A\to(B\to F^R)\to F^R} \diamond_{\text{Cod}} g^{:\forall S.\,B\to(C\to F^S)\to F^S}\right) ⨾ \text{outC} \overset{?}{=} (f ⨾ \text{outC}) \diamond_F (g ⨾ \text{outC}) \quad .$$

Apply both sides to an arbitrary value $a^{:A}$ and simplify the results:

$$\begin{aligned}
\text{left-hand side}: \quad & a \triangleright (f \diamond_{\text{Cod}} g) ⨾ \text{outC} = \left(\forall T.\,k \to f^T(a)\left(b \to g^T(b)(k)\right)\right) ⨾ (d \to d(\text{pu}_F)) \\
& = \left(\forall T.\,k \to f^T(a)\left(b \to g^T(b)(k)\right)\right)(\text{pu}_F) = f(a)(b \to g(b)(\text{pu}_F)) \quad . \\
\text{right-hand side}: \quad & a \triangleright \left((f ⨾ \text{outC}) \diamond_F (g ⨾ \text{outC})\right) = a \triangleright (f ⨾ \text{outC}) ⨾ \text{flm}_F(g ⨾ \text{outC}) \\
& = a \triangleright f \triangleright \text{outC} \triangleright \text{flm}_F(g ⨾ \text{outC}) = f(a)(\text{pu}_F) \triangleright \text{flm}_F\left(b \to g(b)(\text{pu}_F)\right) \quad .
\end{aligned}$$

We find two different expressions that cannot be transformed into each other:

$$f(a)(b \to g(b)(\text{pu}_F)) \neq f(a)(\text{pu}_F) \triangleright \text{flm}_F\left(b \to g(b)(\text{pu}_F)\right) \quad .$$

There are no laws that would allow us to replace pu_F in the *argument* of f because f is an arbitrary function. In part **(c)** of this proof, we will see an example that violates the composition law of `outC`.

(c) The functions `inC` and `outC` are not isomorphisms because $\text{outC} ⨾ \text{inC} \neq \text{id}$. To show this, choose $F^A \overset{\text{def}}{=} R \to A$ (the `Reader` monad) and simplify the type $\text{Cod}^{F,A}$ using the Yoneda identity:

$$\text{Cod}^{F,A} = \forall B.\,(\underline{A \to R} \to B) \to \underline{R \to} B = R \to \underline{\forall B.\,(A \times R \to B) \to B} = R \to A \times R \quad .$$

The resulting type constructor is the `State` monad ($\text{State}^{R,A}$), which includes the effect of `Reader` and additionally the effect of changing a value of type R and passing it to the next computation (which `Reader` cannot do).

For this choice of F, the functions `inC` and `outC` are defined as:

$$\begin{aligned}
& \text{inC} : (R \to A) \to R \to A \times R \quad , \qquad \text{inC} \overset{\text{def}}{=} g^{:R\to A} \to r^{:R} \to g(r) \times r \quad , \\
& \text{outC} : (R \to A \times R) \to R \to A \quad , \qquad \text{outC} \overset{\text{def}}{=} s^{:R\to A\times R} \to r^{:R} \to s(r) \triangleright \pi_1 \quad .
\end{aligned}$$

(These type signatures admit no other fully parametric implementations.) The function `outC` is a flipped version of the `State` monad's runner. That runner returns the result value of type A but discards the possibly changed state (of type R). This makes the `State` monad's runner fail the composition law, as we already noted in the proof of Statement 14.8.1.1.

To obtain a specific example showing that $\text{outC} ⨾ \text{inC} \neq \text{id}$, choose the type $R \overset{\text{def}}{=} \mathbb{1} + P$ with some fixed type P and define a monadic value $s : \text{State}^{R,R}$ by:

$$s : \mathbb{1} + P \to (\mathbb{1} + P) \times (\mathbb{1} + P) \quad , \qquad s \overset{\text{def}}{=} r^{:\mathbb{1}+P} \to r \times (\mathbb{1} + \mathbb{0}) \quad .$$

Let us now compute $s \triangleright \text{outC} ⨾ \text{inC}$ and check that the result is not equal to s:

$$\begin{aligned}
& s \triangleright \text{outC} = r^{:\mathbb{1}+P} \to s(r) \triangleright \pi_1 = r \to r = \text{id} \quad , \\
& s \triangleright \text{outC} \triangleright \text{inC} = \text{id} \triangleright \text{inC} = r \to r \times r \neq s = r \to r \times (\mathbb{1} + \mathbb{0}) \quad .
\end{aligned}$$

We conclude that $\text{outC} ⨾ \text{inC} \neq \text{id}$ for some monads F. □

This proof shows that $\text{Cod}^{F,\bullet}$ may support some additional effects not described by the monad F. So, $\text{Cod}^{F,\bullet}$ and F cannot be equivalent in general. An example when the equivalence $\text{Cod}^{F,A} \cong F^A$ nevertheless holds is found by setting $F = \text{Id}$:

$$\text{Cod}^{\text{Id},\bullet} = \forall R.\,(A \to R) \to R \cong A = \text{Id}^A \quad .$$

Statement 14.8.4.4 A monad transformer for $\text{Cod}^{F,\bullet}$ can be defined by:

$$T^{M,A}_{\text{Cod}} \overset{\text{def}}{=} \forall R.\,(A \to M^{F^R}) \to M^{F^R} \quad .$$

This assumes a lawful foreign monad M, while F may be any type constructor. The transformer T_{Cod} does not have the base lift, foreign runner, or base runner methods.

Proof Similarly to the continuation monad's transformer, we only prove three groups of laws.

The monad laws hold for $T^{M,A}_{\text{Cod}}$ due to Statement 14.8.4.2, where we need to replace the type constructor F by the composition $M^{F^\bullet}$.

The identity laws When $M = \text{Id}$, the type $T^{\text{Id},A}_{\text{Cod}}$ is the same as $\text{Cod}^{F,A}$, so the isomorphism is given by identity functions. Laws always hold for identity functions.

The lifting laws require implementing a monad morphism `flift` with type $M^A \to T^{M,A}_{\text{Cod}}$:

$$\text{flift} : M^A \to \forall R.\,(A \to M^{F^R}) \to M^{F^R} \quad , \qquad \text{flift} \stackrel{\text{def}}{=} \forall R.\, m^{:M^A} \to k^{:A\to M^{F^R}} \to m \triangleright \text{flm}_M(k) \quad .$$

We note that the code of `flift` is the same as the code of the function `inC` from Statement 14.8.4.3:

$$\text{inC} : M^A \to \forall S.\,(A \to M^S) \to M^S \quad , \qquad \text{inC} \stackrel{\text{def}}{=} \forall S.\, m^{:M^A} \to k^{:A\to M^S} \to m \triangleright \text{flm}_M(k) \quad ,$$

where we renamed F to M and R to S to make the comparison easier. The difference between `flift` and `inC` is only in the universally quantified types. We obtain `flift` from `inC` by restricting the type S to types of the form F^R (where R is arbitrary). By Statement 14.8.4.3, the monad morphism laws hold for `inC`, where the type S is unrestricted. So, the same laws will hold for `flift` where S is limited to a subset of possible types. We do not need to repeat the proofs of those laws.

14.8.5 Examples of monads with two different transformers

As a rule, a monad has only one monad transformer. This section shows two exceptions to that rule: monads with two inequivalent transformers. As we will see, in both cases one transformer is complete (has all required methods) but the other is incomplete.

The first example is the selector monad, $\text{Sel}^{Q,A} = (A \to Q) \to A$. That monad, described in Section 14.5.3, has a composed-outside transformer $T_1^{M,A} \stackrel{\text{def}}{=} (M^A \to Q) \to M^A$.

At the same time, $\text{Sel}^{Q,A}$ is equivalent to the generalized `Search` monad (Section 14.8.3) with L chosen to be the identity monad:

$$\text{Sel}^{Q,A} \cong \text{Search}^{\text{Id},Q,A} \quad .$$

So, the selector monad $\text{Sel}^{Q,A}$ has another transformer with the type $T_2^{M,A} \stackrel{\text{def}}{=} (A \to M^Q) \to M^A$.

The transformers T_1 and T_2 cannot be equivalent because the transformer T_1 is covariant in the type parameter M while T_2 is not. Since the transformer T_2 is incomplete (it lacks runners and a base lift), it is preferable to use the transformer T_1.

The second example is found when considering the codensity monad ($\text{Cod}^{F,\bullet}$). That monad is defined using an arbitrary type constructor F. When F is itself a monad, we see that $\text{Cod}^{F,\bullet}$ is a new monad that includes the effects of F (Statement 14.8.4.3). So, we could view $\text{Cod}^{F,\bullet}$ as a monad transformer applied to a foreign monad F. What is the base monad for that monad transformer? We expect the base monad to be obtained as $\text{Cod}^{F,\bullet}$ with $F = \text{Id}$; this gives $\text{Cod}^{\text{Id},\bullet} = \text{Id}^\bullet$ (see proof of Statement 14.8.4.3). So, the codensity type constructor is an alternative transformer *for the identity monad*. The identity monad also has a standard transformer, $T^{M,A}_{\text{Id}} \stackrel{\text{def}}{=} M^A$.

Statement 14.8.5.1 The type constructor $\text{Cod}^{M,\bullet}$ is a transformer for the identity monad (Id) applied to the foreign monad M. The transformer is incomplete as it lacks a foreign runner, and its base runner fails the composition law of monad morphisms.

Proof The monad laws hold because $\text{Cod}^{M,\bullet}$ is a monad (Statement 14.8.4.2).

To verify the identity laws: when $M = \text{Id}$, we have (by the Yoneda identity):

$$\text{Cod}^{\text{Id},A} = \forall R.\,(A \to R) \to R \cong A = \text{Id}^A \quad .$$

The foreign lift method is defined as the monad morphism `inC` (Statement 14.8.4.3).

A foreign runner is not possible since $\text{Cod}^{M,\bullet}$ is not covariant in M.

A base runner should have type signature $\text{Cod}^{M,A} \to M^A$. This is implemented by the function `outC` from Statement 14.8.4.3. However, the proof of that statement shows that `outC` does not satisfy the monad morphism composition law.

14.9 Further developments

14.9.1 Some properties of monad morphisms

In this section we prove some properties of monad morphisms used elsewhere in this chapter.

Statement 14.9.1.1 For any monad M, the function $\Delta : M^A \to M^A \times M^A$ is a monad morphism between monads M and $M \times M$.

Proof We use the definition of the product monad. The method $\mathrm{pu}_{M\times M}$ is defined by:

$$x \triangleright \mathrm{pu}_{M\times M} = \mathrm{pu}_M(x) \times \mathrm{pu}_M(x) = x \triangleright \mathrm{pu}_M \fatsemi \Delta \quad ,$$

which proves the identity law for Δ (namely, $\mathrm{pu}_M \fatsemi \Delta = \mathrm{pu}_{M\times M}$). To verify Δ's composition law,

$$\mathrm{ftn}_M \fatsemi \Delta = \Delta^{\uparrow M} \fatsemi \Delta \fatsemi \mathrm{ftn}_{M\times M} \quad ,$$

we use the definition (10.17) of $\mathrm{ftn}_{M\times M}$:

$$\mathrm{ftn}_{M\times M} : M^{M^\bullet \times M^\bullet} \times M^{M^\bullet \times M^\bullet} \to M^\bullet \times M^\bullet = \left(\pi_1^{\uparrow M} \fatsemi \mathrm{ftn}_M\right) \boxtimes \left(\pi_2^{\uparrow M} \fatsemi \mathrm{ftn}_M\right) \quad ,$$

and compute:

$$\begin{aligned}
\text{expect to equal } \mathrm{ftn}_M \fatsemi \Delta : \quad & \underline{\Delta^{\uparrow M} \fatsemi \Delta} \fatsemi \mathrm{ftn}_{M\times M} \\
\text{naturality law of } \Delta : \quad & = \Delta \fatsemi \left(\Delta^{\uparrow M} \boxtimes \Delta^{\uparrow M}\right) \fatsemi \underline{\mathrm{ftn}_{M\times M}} \\
\text{definition of } \mathrm{ftn}_{M\times M} : \quad & = \Delta \fatsemi \left(\Delta^{\uparrow M} \boxtimes \Delta^{\uparrow M}\right) \fatsemi \left(\pi_1^{\uparrow M} \fatsemi \mathrm{ftn}_M\right) \boxtimes \left(\pi_2^{\uparrow M} \fatsemi \mathrm{ftn}_M\right) \\
\text{composition law of } \boxtimes : \quad & = \Delta \fatsemi \left(\underline{\Delta^{\uparrow M} \fatsemi \pi_1^{\uparrow M}} \fatsemi \mathrm{ftn}_M\right) \boxtimes \left(\underline{\Delta^{\uparrow M} \fatsemi \pi_2^{\uparrow M}} \fatsemi \mathrm{ftn}_M\right) \\
\text{simplify } \Delta \fatsemi \pi_i = \mathrm{id} : \quad & = \Delta \fatsemi \mathrm{ftn}_M \boxtimes \mathrm{ftn}_M \\
\text{duplication law of } \Delta : \quad & = \mathrm{ftn}_M \fatsemi \Delta \quad .
\end{aligned}$$

Statement 14.9.1.2 For any monads K, L, M, N and monad morphisms $\phi : K \rightsquigarrow M$ and $\chi : L \rightsquigarrow N$, the pair product $\phi \boxtimes \chi : K \times L \rightsquigarrow M \times N$ is a monad morphism between the monads $K \times L$ and $M \times N$.

Proof The definitions of $\mathrm{pu}_{K\times L}$ and $\mathrm{pu}_{M\times N}$ are:

$$\mathrm{pu}_{K\times L} = \mathrm{pu}_K \boxtimes \mathrm{pu}_L \quad , \qquad \mathrm{pu}_{M\times N} = \mathrm{pu}_M \boxtimes \mathrm{pu}_N \quad .$$

To verify the identity law for $\phi \boxtimes \chi$:

$$\begin{aligned}
\text{expect to equal } \mathrm{pu}_{M\times N} : \quad & \underline{\mathrm{pu}_{K\times L}} \fatsemi \phi \boxtimes \chi \\
\text{definition of } \mathrm{pu}_{K\times L} : \quad & = \mathrm{pu}_K \boxtimes \mathrm{pu}_L \fatsemi \phi \boxtimes \chi \\
\text{composition law of } \boxtimes : \quad & = \left(\underline{\mathrm{pu}_K \fatsemi \phi}\right) \boxtimes \left(\underline{\mathrm{pu}_L \fatsemi \chi}\right) \\
\text{identity laws of } \phi, \chi : \quad & = \mathrm{pu}_M \boxtimes \mathrm{pu}_N = \mathrm{pu}_{M\times N} \quad .
\end{aligned}$$

To verify the composition law for $\phi \boxtimes \chi$, we use the definitions:

$$\begin{aligned}
\mathrm{ftn}_{K\times L} &= \left(\pi_1^{\uparrow K} \fatsemi \mathrm{ftn}_K\right)^{:K^{K^\bullet \times L^\bullet} \to K^\bullet} \boxtimes \left(\pi_2^{\uparrow L} \fatsemi \mathrm{ftn}_L\right)^{:L^{K^\bullet \times L^\bullet} \to L^\bullet} \quad , \\
\mathrm{ftn}_{M\times N} &= \left(\pi_1^{\uparrow M} \fatsemi \mathrm{ftn}_M\right) \boxtimes \left(\pi_2^{\uparrow N} \fatsemi \mathrm{ftn}_N\right) \quad .
\end{aligned}$$

Denote $\psi \stackrel{\mathrm{def}}{=} \phi \boxtimes \chi$ for brevity. The required law is:

$$\mathrm{ftn}_{K\times L} \fatsemi \psi = \psi^{\uparrow (K\times L)} \fatsemi \psi \fatsemi \mathrm{ftn}_{M\times N} \quad .$$

The left-hand side of this law is:

$$\begin{aligned}
& \text{ftn}_{K\times L} ⨾ \psi \\
\text{definition of ftn}_{M\times N}: \quad & = (\pi_1^{\uparrow K} ⨾ \text{ftn}_K) \boxtimes (\pi_2^{\uparrow L} ⨾ \text{ftn}_L) ⨾ \phi \boxtimes \chi \\
\text{composition law of } \boxtimes: \quad & = (\pi_1^{\uparrow K} ⨾ \text{ftn}_K ⨾ \phi) \boxtimes (\pi_2^{\uparrow L} ⨾ \text{ftn}_L ⨾ \chi)
\end{aligned}$$

The right-hand side is:

$$\begin{aligned}
& \underline{\psi^{\uparrow(K\times L)}} ⨾ \psi ⨾ \text{ftn}_{M\times N} \\
\text{lifting to } K\times L: \quad & = (\psi^{\uparrow K} \boxtimes \psi^{\uparrow L}) ⨾ (\phi \boxtimes \chi) ⨾ \underline{\text{ftn}_{M\times N}} \\
\text{definition of ftn}_{M\times N}: \quad & = \psi^{\uparrow K} \boxtimes \psi^{\uparrow L} ⨾ \phi \boxtimes \chi ⨾ (\pi_1^{\uparrow M} ⨾ \text{ftn}_M) \boxtimes (\pi_2^{\uparrow N} ⨾ \text{ftn}_N) \\
\text{composition law of } \boxtimes: \quad & = (\psi^{\uparrow K} ⨾ \phi ⨾ \pi_1^{\uparrow M} ⨾ \text{ftn}_M) \boxtimes (\psi^{\uparrow L} ⨾ \chi ⨾ \pi_2^{\uparrow N} ⨾ \text{ftn}_N) \quad .
\end{aligned}$$

Now rewrite the first part of the pair product in the last line:

$$\begin{aligned}
& \underline{\psi^{\uparrow K} ⨾ \phi} ⨾ \pi_1^{\uparrow M} ⨾ \text{ftn}_M \\
\text{naturality law of } \phi: \quad & = \phi ⨾ \underline{\psi^{\uparrow M} ⨾ \pi_1^{\uparrow M}} ⨾ \text{ftn}_M \\
\text{projection law of } \pi_1: \quad & = \underline{\phi ⨾ \pi_1^{\uparrow M}} ⨾ \phi^{\uparrow M} ⨾ \text{ftn}_M \\
\text{naturality law of } \phi: \quad & = \pi_1^{\uparrow K} ⨾ \underline{\phi ⨾ \phi^{\uparrow M} ⨾ \text{ftn}_M} \\
\text{composition law of } \phi: \quad & = \pi_1^{\uparrow K} ⨾ \text{ftn}_K ⨾ \phi \quad .
\end{aligned}$$

In the same way, we find that the second part of the pair product is $\pi_2^{\uparrow L} ⨾ \text{ftn}_L ⨾ \chi$, and so the composition law holds.

Statement 14.9.1.3 For any monads M and N, the projection function $\pi_1 : M \times N \rightsquigarrow M$ is a monad morphism. Same for $\pi_2 : M \times N \rightsquigarrow N$.

Proof It is sufficient to verify the laws for π_1; the proof for π_2 will be similar. The identity law:

$$\begin{aligned}
\text{expect to equal pu}_M: \quad & \text{pu}_{M\times N} ⨾ \pi_1 \\
\text{definition of pu}_{M\times N}: \quad & = (\text{pu}_M \boxtimes \text{pu}_N) ⨾ \pi_1 \\
\text{projection law of } \pi_1: \quad & = \text{pu}_M \quad .
\end{aligned}$$

The composition law:

$$\begin{aligned}
\text{expect to equal } \pi_1 ⨾ \pi_1^{\uparrow M} ⨾ \text{ftn}_M: \quad & \underline{\text{ftn}_{M\times N}} ⨾ \pi_1 \\
\text{definition of ftn}_{M\times N}: \quad & = \underline{(\pi_1^{\uparrow M} ⨾ \text{ftn}_M)} \boxtimes (\pi_2^{\uparrow N} ⨾ \text{ftn}_N) ⨾ \underline{\pi_1} \\
\text{projection law of } \pi_1: \quad & = \pi_1 ⨾ \pi_1^{\uparrow M} ⨾ \text{ftn}_M \quad .
\end{aligned}$$

Statement 14.9.1.4 For any monads M and N, the swapping function $\sigma : M \times N \rightsquigarrow N \times M$ is a monad morphism.

Proof The code for σ can be written as a combination of other functions: $\sigma = \Delta ⨾ (\pi_2 \boxtimes \pi_1)$. The functions Δ, π_1, and π_2 are monad morphisms by Statements 14.9.1.1 and 14.9.1.3. The function product $\pi_1 \boxtimes \pi_2$ is a monad morphism by Statement 14.9.1.2. So σ is a composition of monad morphisms; by Statement 10.3.4.5, σ is a monad morphism.

14.9.2 Does a composition of rigid monads have a `swap` function?

In Section 14.5.3, we proved the monad transformer laws for $T_R^M \stackrel{\text{def}}{=} R \circ M$ by using a `swap` function with type signature:

$$\text{sw}_{R,M} : M \circ R \rightsquigarrow R \circ M \quad ,$$

and proving its special laws. If R and S are both rigid monads then the composed monad $R \circ S$ is also rigid. Does $T \stackrel{\text{def}}{=} R \circ S$ have a suitable `swap` function:

$$\text{sw}_{T,M} : M \circ R \circ S \rightsquigarrow R \circ S \circ M \quad ,$$

satisfying all the required special laws of `swap`? If so, we may be able to find a simpler definition of `flatten` for the monad stack $R \circ S \circ M$. Let us briefly investigate this question. However, keep in mind that the absence of a suitable `swap` function will not invalidate the composition properties of rigid monad stacks: those properties were established without assuming the existence of `swap` for the composed monad.

It turns out that we need yet another law for `swap` (the "3-swap" law) if we wish to prove that the composed monad also has a lawful `swap`.

Statement 14.9.2.1 Assume that two monads R and S are rigid and both have `swap` functions ($\text{sw}_{R,M}$ and $\text{sw}_{S,M}$) satisfying the 8 laws listed in Statements 14.3.2.1 and 14.3.7.1. Additionally, assume that the functions $\text{sw}_{R,M}$ and $\text{sw}_{S,M}$ obey the **3-swap law** with respect to an arbitrary monad M:

$$\begin{tikzcd}
M^{S^{R^A}} \arrow[r, "\text{sw}^{\uparrow M}_{R,S}"] \arrow[d, "\text{sw}_{S,M}"'] & M^{R^{S^A}} \arrow[r, "\text{sw}_{R,M}"] & R^{M^{S^A}} \arrow[d, "\text{sw}^{\uparrow R}_{S,M}"] \\
S^{M^{R^A}} \arrow[r, "\text{sw}^{\uparrow S}_{R,M}"] & S^{R^{M^A}} \arrow[r, "\text{sw}_{R,S}"] & R^{S^{M^A}}
\end{tikzcd}$$

$$\begin{tikzcd}
M \circ R \circ S \arrow[r, "\text{sw}_{R,M}"] \arrow[dr, "\text{sw}_{T,M} \stackrel{\text{def}}{=}"'] & R \circ M \circ S \arrow[d, "\text{sw}^{\uparrow R}_{S,M}"] \\
& R \circ S \circ M
\end{tikzcd}$$

$$\text{sw}^{\uparrow M}_{R,S} \fatsemi \text{sw}_{R,M} \fatsemi \text{sw}^{\uparrow R}_{S,M} = \text{sw}_{S,M} \fatsemi \text{sw}^{\uparrow S}_{R,M} \fatsemi \text{sw}_{R,S} \quad . \tag{14.91}$$

Then a `swap` function ($\text{sw}_{T,M}$) also exists for the composed monad $T \stackrel{\text{def}}{=} R \circ S$, is defined by :

$$\text{sw}_{T,M} \stackrel{\text{def}}{=} \text{sw}_{R,M} \fatsemi \text{sw}^{\uparrow R}_{S,M} \quad , \tag{14.92}$$

and satisfies the same 8 laws.

Proof We need to verify the 8 laws for $\text{sw}_{T,M}$ (one naturality law, two identity laws, two interchange laws, and three monadic naturality laws), assuming that these 8 laws hold for $\text{sw}_{R,M}$ and $\text{sw}_{S,M}$. In addition, we assume that Eq. (14.91) holds, where M is an arbitrary monad. The monad methods of T are defined by Eqs. (14.19)–(14.20) after renaming $L \stackrel{\text{def}}{=} R$ and $M \stackrel{\text{def}}{=} S$:

$$\text{pu}_T = \text{pu}_S \fatsemi \text{pu}_R \quad , \tag{14.93}$$

$$\text{ftn}_T = \text{sw}^{\uparrow R}_{R,S} \fatsemi \text{ftn}_R \fatsemi \text{ftn}^{\uparrow R}_S \quad . \tag{14.94}$$

As usual, we do not need to verify the naturality law for $\text{sw}_{T,M}$.

Identity laws To verify the outer identity law:

$$\begin{aligned}
\text{expect to equal } \text{pu}_T : \quad & \text{pu}^{\uparrow M}_T \fatsemi \text{sw}_{T,M} \\
\text{use Eqs. (14.92)-(14.93)} : \quad & = \text{pu}^{\uparrow M}_S \fatsemi \underline{\text{pu}^{\uparrow M}_R \fatsemi \text{sw}_{R,M}} \fatsemi \text{sw}^{\uparrow R}_{S,M} \\
\text{outer identity law of } \text{sw}_{R,M} : \quad & = \text{pu}^{\uparrow M}_S \fatsemi \underline{\text{pu}_R \fatsemi \text{sw}^{\uparrow R}_{S,M}} \\
\text{naturality law of } \text{pu}_R : \quad & = \underline{\text{pu}^{\uparrow M}_S \fatsemi \text{sw}_{S,M}} \fatsemi \text{pu}_R \\
\text{outer identity law of } \text{sw}_{S,M} : \quad & = \text{pu}_S \fatsemi \text{pu}_R \\
\text{use Eq. (14.93)} : \quad & = \text{pu}_T \quad .
\end{aligned}$$

To verify the inner identity law:

$$\begin{aligned}
\text{expect to equal } \text{pu}^{\uparrow T}_M : \quad & \text{pu}_M \fatsemi \underline{\text{sw}_{T,M}} \\
\text{use Eq. (14.92)} : \quad & = \underline{\text{pu}_M \fatsemi \text{sw}_{R,M}} \fatsemi \text{sw}^{\uparrow R}_{S,M} \\
\text{inner identity law of } \text{sw}_{R,M} : \quad & = \text{pu}^{\uparrow R}_M \fatsemi \text{sw}^{\uparrow R}_{S,M} = (\underline{\text{pu}_M \fatsemi \text{sw}_{S,M}})^{\uparrow R} \\
\text{inner identity law of } \text{sw}_{S,M} : \quad & = \text{pu}^{\uparrow S \uparrow R}_M = \text{pu}^{\uparrow T}_M \quad .
\end{aligned}$$

Interchange laws The outer interchange law is:

$$\text{ftn}_T^{\uparrow M} \fatsemi \text{sw}_{T,M} = \text{sw}_{T,M} \fatsemi \text{sw}_{T,M}^{\uparrow T} \fatsemi \text{ftn}_T \quad . \tag{14.95}$$

We will use the outer interchange laws for $\text{sw}_{R,M}$ and $\text{sw}_{S,M}$:

$$\text{ftn}_R^{\uparrow M} \fatsemi \text{sw}_{R,M} = \text{sw}_{R,M} \fatsemi \text{sw}_{R,M}^{\uparrow R} \fatsemi \text{ftn}_R \quad , \tag{14.96}$$

$$\text{ftn}_S^{\uparrow M} \fatsemi \text{sw}_{S,M} = \text{sw}_{S,M} \fatsemi \text{sw}_{S,M}^{\uparrow S} \fatsemi \text{ftn}_S \quad . \tag{14.97}$$

Begin with the right-hand side of Eq. (14.95) since it is more complicated:

$$\begin{aligned}
& \text{sw}_{T,M} \fatsemi \text{sw}_{T,M}^{\uparrow T} \fatsemi \text{ftn}_T \\
\text{definitions}: \quad & = \text{sw}_{R,M} \fatsemi \underline{\text{sw}_{S,M}^{\uparrow R} \fatsemi (\text{sw}_{R,M} \fatsemi \text{sw}_{S,M}^{\uparrow R})^{\uparrow S \uparrow R} \fatsemi \text{sw}_{R,S}^{\uparrow R}} \fatsemi \text{ftn}_R \fatsemi \text{ftn}_S^{\uparrow R} \\
\text{composition under } {}^{\uparrow R}: \quad & = \text{sw}_{R,M} \fatsemi (\text{sw}_{S,M} \fatsemi \text{sw}_{R,M}^{\uparrow S} \fatsemi \underline{\text{sw}_{S,M}^{\uparrow R \uparrow S} \fatsemi \text{sw}_{R,S}})^{\uparrow R} \fatsemi \text{ftn}_R \fatsemi \text{ftn}_S^{\uparrow R} \\
\text{naturality law of } \text{sw}_{R,S}: \quad & = \text{sw}_{R,M} \fatsemi (\underline{\text{sw}_{S,M} \fatsemi \text{sw}_{R,M}^{\uparrow S} \fatsemi \text{sw}_{R,S}} \fatsemi \text{sw}_{S,M}^{\uparrow S \uparrow R})^{\uparrow R} \fatsemi \text{ftn}_R \fatsemi \text{ftn}_S^{\uparrow R} \\
\text{use Eq. (14.91)}: \quad & = \text{sw}_{R,M} \fatsemi (\underline{\text{sw}_{R,S}^{\uparrow M} \fatsemi \text{sw}_{R,M} \fatsemi \text{sw}_{S,M}^{\uparrow R}} \fatsemi \text{sw}_{S,M}^{\uparrow S \uparrow R})^{\uparrow R} \fatsemi \text{ftn}_R \fatsemi \text{ftn}_S^{\uparrow R} \quad .
\end{aligned}$$

In the last expression, all `flatten`s are to the right of all `swap`s. So we look to rewrite the left-hand side of Eq. (14.95) into the same form:

$$\begin{aligned}
& \text{ftn}_T^{\uparrow M} \fatsemi \text{sw}_{T,M} \\
\text{use Eqs. (14.92), (14.94)}: \quad & = (\text{sw}_{R,S}^{\uparrow R} \fatsemi \text{ftn}_R \fatsemi \underline{\text{ftn}_S^{\uparrow R}})^{\uparrow M} \fatsemi \text{sw}_{R,M} \fatsemi \text{sw}_{S,M}^{\uparrow R} \\
\text{naturality law of } \text{sw}_{R,M}: \quad & = \text{sw}_{R,S}^{\uparrow R \uparrow M} \fatsemi \underline{\text{ftn}_R^{\uparrow M} \fatsemi \text{sw}_{R,M}} \fatsemi \text{ftn}_S^{\uparrow M \uparrow R} \fatsemi \text{sw}_{S,M}^{\uparrow R} \\
\text{use Eq. (14.96)}: \quad & = \text{sw}_{R,S}^{\uparrow R \uparrow M} \fatsemi \text{sw}_{R,M} \fatsemi \text{sw}_{R,M}^{\uparrow R} \fatsemi \text{ftn}_R \fatsemi (\underline{\text{ftn}_S^{\uparrow M} \fatsemi \text{sw}_{S,M}})^{\uparrow R} \\
\text{use Eq. (14.97)}: \quad & = \text{sw}_{R,M} \fatsemi \text{sw}_{R,S}^{\uparrow M \uparrow R} \fatsemi \text{sw}_{R,M}^{\uparrow R} \fatsemi \underline{\text{ftn}_R \fatsemi (\text{sw}_{S,M} \fatsemi \text{sw}_{S,M}^{\uparrow S} \fatsemi \text{ftn}_S)^{\uparrow R}} \\
\text{naturality law of } \text{ftn}_R: \quad & = \text{sw}_{R,M} \fatsemi \underline{\text{sw}_{R,S}^{\uparrow M \uparrow R} \fatsemi \text{sw}_{R,M}^{\uparrow R} \fatsemi (\text{sw}_{S,M} \fatsemi \text{sw}_{S,M}^{\uparrow S})^{\uparrow R \uparrow R}} \fatsemi \text{ftn}_R \fatsemi \text{ftn}_S^{\uparrow R} \\
\text{composition under } {}^{\uparrow R}: \quad & = \text{sw}_{R,M} \fatsemi (\underline{\text{sw}_{R,S}^{\uparrow M} \fatsemi \text{sw}_{R,M} \fatsemi \text{sw}_{S,M}^{\uparrow R} \fatsemi \text{sw}_{S,M}^{\uparrow S \uparrow R}})^{\uparrow R} \fatsemi \text{ftn}_R \fatsemi \text{ftn}_S^{\uparrow R} \quad .
\end{aligned}$$

Both sides of the outer interchange law (14.95) are now equal.

The proof of the inner interchange law is simpler. The law says:

$$\text{ftn}_M \fatsemi \text{sw}_{T,M} = \text{sw}_{T,M}^{\uparrow M} \fatsemi \text{sw}_{T,M} \fatsemi \text{ftn}_M^{\uparrow T} \quad . \tag{14.98}$$

We will use the inner interchange laws for $\text{sw}_{R,M}$ and $\text{sw}_{S,M}$:

$$\text{ftn}_M \fatsemi \text{sw}_{R,M} = \text{sw}_{R,M}^{\uparrow M} \fatsemi \text{sw}_{R,M} \fatsemi \text{ftn}_M^{\uparrow R} \quad , \tag{14.99}$$

$$\text{ftn}_M \fatsemi \text{sw}_{S,M} = \text{sw}_{S,M}^{\uparrow M} \fatsemi \text{sw}_{S,M} \fatsemi \text{ftn}_M^{\uparrow S} \quad . \tag{14.100}$$

Begin with the left-hand side of Eq. (14.98):

$$\begin{aligned}
& \text{ftn}_M \fatsemi \text{sw}_{T,M} = \underline{\text{ftn}_M \fatsemi \text{sw}_{R,M}} \fatsemi \text{sw}_{S,M}^{\uparrow R} \\
\text{use Eq. (14.99)}: \quad & = \text{sw}_{R,M}^{\uparrow M} \fatsemi \text{sw}_{R,M} \fatsemi \underline{\text{ftn}_M^{\uparrow R} \fatsemi \text{sw}_{S,M}^{\uparrow R}} \\
\text{use Eq. (14.100) under } {}^{\uparrow R}: \quad & = \text{sw}_{R,M}^{\uparrow M} \fatsemi \text{sw}_{R,M} \fatsemi \underline{(\text{sw}_{S,M}^{\uparrow M} \fatsemi \text{sw}_{S,M} \fatsemi \text{ftn}_M^{\uparrow S})^{\uparrow R}} \\
\text{composition under } {}^{\uparrow R}: \quad & = \text{sw}_{R,M}^{\uparrow M} \fatsemi \text{sw}_{R,M} \fatsemi \underline{\text{sw}_{S,M}^{\uparrow M \uparrow R} \fatsemi \text{sw}_{S,M}^{\uparrow R} \fatsemi \text{ftn}_M^{\uparrow S \uparrow R}}
\end{aligned}$$

The right-hand side of Eq. (14.98) is:

$$\begin{aligned}
& \mathrm{sw}^{\uparrow M}_{T,M} \fatsemi \mathrm{sw}_{T,M} \fatsemi \mathrm{ftn}^{\uparrow T}_{M} \\
\text{use Eq. (14.92)}: \quad & = \big(\mathrm{sw}_{R,M} \fatsemi \underline{\mathrm{sw}^{\uparrow R}_{S,M}}\big)^{\uparrow M} \fatsemi \underline{\mathrm{sw}_{R,M}} \fatsemi \mathrm{sw}^{\uparrow R}_{S,M} \fatsemi \mathrm{ftn}^{\uparrow S \uparrow R}_{M} \\
\text{naturality law of } \mathrm{sw}_{R,M}: \quad & = \mathrm{sw}^{\uparrow M}_{R,M} \fatsemi \underline{\mathrm{sw}_{R,M} \fatsemi \mathrm{sw}^{\uparrow M \uparrow R}_{S,M}} \fatsemi \mathrm{sw}^{\uparrow R}_{S,M} \fatsemi \mathrm{ftn}^{\uparrow S \uparrow R}_{M} \quad .
\end{aligned}$$

The right-hand side is now equal to the left-hand side.

Monadic naturality laws We need to verify that the following three laws:

$$\mathrm{sw}_{T,\mathrm{Id}} = \mathrm{id} \quad , \qquad \mathrm{sw}_{T,M} \fatsemi \phi^{\uparrow T} = \phi \fatsemi \mathrm{sw}_{T,N} \quad , \qquad \mathrm{sw}_{T,M} \fatsemi \theta = \theta^{\uparrow M} \quad ,$$

hold for any monad morphisms $\phi : M \rightsquigarrow N$ and $\theta : T \rightsquigarrow \mathrm{Id}$. We may assume that these laws already hold for $\mathrm{sw}_{R,M}$ and $\mathrm{sw}_{S,M}$.

To verify the first law, write:

$$\begin{aligned}
\text{expect to equal id}: \quad & \mathrm{sw}_{T,\mathrm{Id}} \\
\text{use Eq. (14.92)}: \quad & = \mathrm{sw}_{R,\mathrm{Id}} \fatsemi \mathrm{sw}^{\uparrow R}_{S,\mathrm{Id}} \\
\text{first law of } \mathrm{sw}_{R,\mathrm{Id}} \text{ and } \mathrm{sw}_{S,\mathrm{Id}}: \quad & = \mathrm{id} \fatsemi \mathrm{id}^{\uparrow R} = \mathrm{id} \quad .
\end{aligned}$$

To verify the second law, write:

$$\begin{aligned}
\text{expect to equal } \phi \fatsemi \mathrm{sw}_{T,N}: \quad & \mathrm{sw}_{T,M} \fatsemi \phi^{\uparrow T} \\
\text{use Eq. (14.92)}: \quad & = \mathrm{sw}_{R,M} \fatsemi \underline{\mathrm{sw}^{\uparrow R}_{S,M} \fatsemi \phi^{\uparrow S \uparrow R}} \\
\text{second law of } \mathrm{sw}_{S,M} \text{ under } {}^{\uparrow R}: \quad & = \underline{\mathrm{sw}_{R,M} \fatsemi} \big(\phi \fatsemi \mathrm{sw}_{S,N}\big)^{\uparrow R} \\
\text{second law of } \mathrm{sw}_{R,M}: \quad & = \phi \fatsemi \underline{\mathrm{sw}_{R,N} \fatsemi \mathrm{sw}^{\uparrow R}_{S,N}} = \phi \fatsemi \underline{\mathrm{sw}_{T,N}} \quad .
\end{aligned}$$

To verify the third law, we begin with the left-hand side:

$$\mathrm{sw}_{T,M} \fatsemi \theta = \mathrm{sw}_{R,M} \fatsemi \mathrm{sw}^{\uparrow R}_{S,M} \fatsemi \theta \quad .$$

At this point, no relationship or law applies to the intermediate expression $\mathrm{sw}^{\uparrow R}_{S,M} \fatsemi \theta$, so we need additional information to proceed. By Statement 14.9.2.2 (proved below), we can express $\theta = \theta_R \fatsemi \theta_S$ with some monad morphisms θ_R and θ_S. So, we may use θ_R and θ_S with the monad morphism laws for R and S, in particular with the third law:

$$\mathrm{sw}_{R,M} \fatsemi \theta_R = \theta_R^{\uparrow M} \quad , \qquad \mathrm{sw}_{S,M} \fatsemi \theta_S = \theta_S^{\uparrow M} \quad .$$

This allows us to finish the proof of the third monadic naturality law:

$$\begin{aligned}
\text{expect to equal } \theta^{\uparrow M}: \quad & \mathrm{sw}_{T,M} \fatsemi \underline{\theta} \\
\text{use Statement 14.9.2.2}: \quad & = \mathrm{sw}_{R,M} \fatsemi \underline{\mathrm{sw}^{\uparrow R}_{S,M} \fatsemi \theta_R} \fatsemi \theta_S \\
\text{naturality law of } \theta_R: \quad & = \underline{\mathrm{sw}_{R,M} \fatsemi \theta_R} \fatsemi \underline{\mathrm{sw}_{S,M} \fatsemi \theta_S} \\
\text{third law of } R \text{ and } S: \quad & = \theta_R^{\uparrow M} \fatsemi \theta_S^{\uparrow M} = \theta^{\uparrow M} \quad .
\end{aligned}$$

Statement 14.9.2.2 Let R be a rigid monad with a lawful `swap` method $\mathrm{sw}_{R,M}$. Let S be any monad. Then any monad morphism $\theta : R \circ S \rightsquigarrow \mathrm{Id}$ can be expressed as $\theta = \theta_R \fatsemi \theta_S$ by defining the functions:

$$\theta_R \stackrel{\text{def}}{=} \mathrm{pu}^{\uparrow R}_{S} \fatsemi \theta \quad , \qquad \theta_S \stackrel{\text{def}}{=} \mathrm{pu}_R \fatsemi \theta \quad ,$$

which are monad morphisms. So, any runner θ for the composed monad $R \circ S$ can be written as a function composition of some runners $\theta_R : R \rightsquigarrow \mathrm{Id}$ and $\theta_S : S \rightsquigarrow \mathrm{Id}$ for the monads R and S.

Proof How can we obtain θ_R and θ_S from the given runner θ? Note that θ runs the operations of both monads R and S, while θ_R runs only the operations of R. To obtain $\theta_R : R \rightsquigarrow \text{Id}$ from $\theta : R \circ S \rightsquigarrow \text{Id}$, we need to prepend a function of type $R \rightsquigarrow R \circ S$. A suitable function of that type is:

$$\text{pu}_S^{\uparrow R} : R^A \to R^{S^A} \quad .$$

So, $\theta_R = \text{pu}_S^{\uparrow R} ⨾ \theta$ has the correct type signature, $R^A \to A$. Similarly, $\theta_S = \text{pu}_R ⨾ \theta$ has the correct type signature, $S^A \to A$. So, we can obtain θ_R and θ_S from the given runner θ as:

$$\theta_R \stackrel{\text{def}}{=} \text{pu}_S^{\uparrow R} ⨾ \theta \quad , \qquad \theta_S \stackrel{\text{def}}{=} \text{pu}_R ⨾ \theta \quad .$$

Since pu_R and $\text{pu}_S^{\uparrow R}$ are the foreign and base lifts of the transformer $T_R^S = R \circ S$, Statement 14.3.7.1 shows that pu_R and $\text{pu}_S^{\uparrow R}$ are monad morphisms. Since the composition of monad morphisms is again a monad morphism (Statement 10.3.4.5), we see that θ_R and θ_S are monad morphisms.

It remains to verify that $\theta = \theta_R ⨾ \theta_S$:

$$\begin{aligned}
&\text{expect to equal } \theta : && \theta_R ⨾ \theta_S = \text{pu}_S^{\uparrow R} ⨾ \underline{\theta ⨾ \text{pu}_R} ⨾ \theta \\
&\text{naturality law of } \theta : && = \text{pu}_S^{\uparrow R} ⨾ \text{pu}_R^{\uparrow T} ⨾ \theta ⨾ \theta = \text{pu}_S^{\uparrow R} ⨾ \text{pu}_R^{\uparrow S \uparrow R} ⨾ \underline{\theta ⨾ \theta} \\
&\text{composition law of } \theta : && = \text{pu}_S^{\uparrow R} ⨾ \text{pu}_R^{\uparrow S \uparrow R} ⨾ \underline{\text{ftn}_T ⨾ \theta} \quad .
\end{aligned}$$

The last line differs from the required result (θ) by the function $\text{pu}_S^{\uparrow R} ⨾ \text{pu}_R^{\uparrow S \uparrow R} ⨾ \text{ftn}_T$. We will finish the proof if we show that this function is equal to an identity function:

$$\begin{aligned}
&\text{expect to equal id} : && \text{pu}_S^{\uparrow R} ⨾ \text{pu}_R^{\uparrow S \uparrow R} ⨾ \underline{\text{ftn}_T} \\
&\text{use Eq. (14.20)} : && = \text{pu}_S^{\uparrow R} ⨾ \underline{\text{pu}_R^{\uparrow S \uparrow R} ⨾ \text{sw}_{R,S}^{\uparrow R}} ⨾ \text{ftn}_R ⨾ \text{ftn}_S^{\uparrow R} \\
&\text{outer-identity law of } \text{sw}_{R,S} : && = \text{pu}_S^{\uparrow R} ⨾ \underline{\text{pu}_R^{\uparrow R} ⨾ \text{ftn}_R} ⨾ \text{ftn}_S^{\uparrow R} \\
&\text{identity laws of } R \text{ and } S : && = \underline{\text{pu}_S^{\uparrow R} ⨾ \text{ftn}_S^{\uparrow R}} = \text{id} \quad .
\end{aligned}$$

14.9.3 Rigid functors and their properties

Some properties of rigid monads can be extended to a (possibly) larger class of rigid functors. We begin with a definition of a rigid functor that, unlike the definition of rigid monads (Section 14.5), does not refer to monad transformers.

Definition A functor R is **rigid** if there exists a natural transformation `fuseIn` (denoted by fi_R) with type signature:

$$\text{fi}_R : (A \to R^B) \to R^{A \to B} \quad ,$$

satisfying the nondegeneracy law (14.102) shown below.

Not all functors admit a natural transformation with the type signature of `fuseIn`. For example, the functor $F^A \stackrel{\text{def}}{=} Z + A$ is not rigid because the required type signature,

$$\text{fi}_F : (A \to Z + B) \to Z + (A \to B) \quad ,$$

cannot be implemented as a natural transformation. However, any functor R admits the opposite natural transformation `fuseOut` (denoted by fo_R):

$$\text{fo}_R : R^{A \to B} \to A \to R^B \quad , \qquad \text{fo}\,(r) \stackrel{\text{def}}{=} a \to r \triangleright \left(f^{:A \to B} \to f\,(a) \right)^{\uparrow R} \quad . \tag{14.101}$$

The method `fuseIn` must additionally satisfy the nondegeneracy law:

$$\text{fi}_R ⨾ \text{fo}_R = \text{id}^{:(A \to R^B) \to (A \to R^B)} \quad . \tag{14.102}$$

The opposite relation does not hold in general: $\text{fo}_R \fatsemi \text{fi}_R \neq \text{id}$ (see Example 14.9.3.4).

The next two statements show some use cases for rigid functors.

The first use case is a generalization of `flatMap` that can handle multiple M-effects at once. As a motivation, consider a given value $m^{:M^A}$ and two Kleisli functions having the same type signature, $f_1 : A \to M^B$ and $f_2 : A \to M^B$. We may apply $\text{flm}_M(f_1)$ and $\text{flm}_M(f_2)$ to m and obtain two different values of type M^B. This computation can be written as:

$$\big(m \triangleright \text{flm}_M(f_1)\big) \times \big(m \triangleright \text{flm}_M(f_2)\big) : M^B \times M^B \quad .$$

To express this computation in a different but equivalent way, we define a new function f as:

$$f : A \to M^B \times M^B \quad , \qquad f \stackrel{\text{def}}{=} \Delta \fatsemi (f_1 \boxtimes f_2) \quad ,$$

and a new "double-valued `flatMap`" as:

$$\text{flm2}_M : (A \to M^B \times M^B) \to M^A \to M^B \times M^B \quad , \quad \text{flm2}_M \stackrel{\text{def}}{=} f \to \Delta \fatsemi \big(\text{flm}_M(f \fatsemi \pi_1) \boxtimes \text{flm}_M(f \fatsemi \pi_2)\big) \quad .$$

It is clear that the same computation can be extended to a product of any number of M's. The general version of the "multi-valued `flatMap`" is a situation when the collection of M-effects is described by a rigid functor R. The example just shown corresponds to $R^A \stackrel{\text{def}}{=} A \times A$.

Statement 14.9.3.1 For any rigid functor R and a monad M, there exists an "R-valued `flatMap`":

$$\text{rflm}_{M,R} : (A \to R^{M^B}) \to M^A \to R^{M^B} \quad .$$

Proof Using R's `fuseIn` method, we can convert a value of type $A \to R^{M^B}$ into a value of type $R^{A \to M^B}$. After that, we can use M's `flatMap`:

$$\text{rflm}_{M,R} \stackrel{\text{def}}{=} k^{:A \to R^{M^B}} \to m^{:M^A} \to k \triangleright \text{fi}_R \triangleright \big(q^{:A \to M^B} \to m \triangleright \text{flm}_M(q)\big)^{\uparrow R} \quad .$$

This code does not assume that R is a monad. □

The second use case is a program transformation that introduces a wrapper into a given program. Assume that $p^{:C}$ is an expression that uses a function $f^{:A \to B}$, and we would like to replace f by a function $g^{:A \to R^B}$. We may imagine that g performs a computation similar to f except it wraps the results in the functor R (for instance, returning several results or using concurrency). Can we obtain a transformed program $q : R^C$ that has the same functionality as p except for also returning a value wrapped in R? This is possible if R is a rigid functor:

Statement 14.9.3.2 Given a rigid functor R, a program transformation can be implemented:

$$\text{refactor} : ((A \to B) \to C) \to (A \to R^B) \to R^C \quad .$$

This function transforms a program $p(f^{:A \to B}) : C$ into a program $q(g^{:A \to R^B}) : R^C$.

Proof We use R's method `fuseIn` to convert a value of type $A \to R^B$ into a value of type $R^{A \to B}$. The rest of the computation is lifted to the functor R:

$$\text{refactor} \stackrel{\text{def}}{=} p^{:(A \to B) \to C} \to k^{:A \to R^B} \to k \triangleright \text{fi}_R \triangleright p^{\uparrow R} \quad .$$

□

For a rigid functor R, the type signature of fi_R is the same as the type signature of `swap` with respect to the `Reader` monad:

$$\text{sw}_{R,M} : M^{R^A} \to R^{M^A} \cong (Z \to R^A) \to R^{Z \to A} \quad \text{if we set} \quad M^A \stackrel{\text{def}}{=} Z \to A \quad .$$

All rigid monads considered in Section 14.5 have a `swap` method. It turns out that all rigid monads (regardless of the existence of `swap`) are also rigid functors. While proving that statement, we will directly use the definition of rigid monads via the composed-outside monad transformer, without assuming a `swap` method.

Statement 14.9.3.3 Any rigid monad R is a rigid functor.

Proof By assumption, the monad R has a composed-outside monad transformer $T_R^M = R \circ M$ and the corresponding method ftn_T. We can define the transformation fi_R in the same way as Eq. (14.21) defined the `swap` method via ftn_T. To use that formula, we need to set the foreign monad M to be the `Reader` monad, $M^B \stackrel{\text{def}}{=} A \to B$ (with a fixed type A):

$$\begin{tikzcd}
(A \to R^B) \arrow[r, "\text{pu}_M^{\uparrow R \uparrow M}"] \arrow[d, "\text{fi}_R \stackrel{\text{def}}{=}"'] & (A \to R^{A \to B}) \arrow[d, "\text{pu}_R"] \\
R^{A \to B} & R^{A \to R^{A \to B}} \arrow[l, "\text{ftn}_T"']
\end{tikzcd}$$

$$\text{fi}_R(f^{:A \to R^B}) = \text{pu}_M^{\uparrow R \uparrow M} \fatsemi \text{pu}_R \fatsemi \text{ftn}_T \quad . \tag{14.103}$$

Since M is the `Reader` monad with the environment type A, we define pu_M and $f^{\uparrow M}$ by:

$$\text{pu}_M(x) = (_^{:A} \to x) \quad , \qquad r^{:A \to X} \triangleright (f^{:X \to Y})^{\uparrow M} = r \fatsemi f \quad .$$

The type signature of the function `fuseOut`,

$$\text{fo}_R : R^{A \to B} \to A \to R^B \quad ,$$

resembles "running" the composed monad R^{M^B} into R^B, consuming a value of type A. Indeed, given a fixed value $a^{:A}$, we can "run" the `Reader` monad into the identity monad. The corresponding runner $\phi_a^{:M \rightsquigarrow \text{Id}}$ is:

$$\phi_a^{:M^X \to X} \stackrel{\text{def}}{=} m^{:A \to X} \to m(a) \quad . \tag{14.104}$$

So we are inspired to use the runner law for the monad transformer T_R^M (which holds since we assumed that T_R^M satisfies all laws). The law says that the lifted runner $\phi_a^{\uparrow R}$ is a monad morphism $T_R^M \rightsquigarrow T_R^{\text{Id}} \cong T_R^M \rightsquigarrow R$. The monad morphism law for $\phi_a^{\uparrow R}$ is then written as:

$$\text{ftn}_T \fatsemi \phi_a^{\uparrow R} = \phi_a^{\uparrow R \uparrow M \uparrow R} \fatsemi \phi_a^{\uparrow R} \fatsemi \text{ftn}_R \quad . \tag{14.105}$$

How could we use this law to obtain Eq. (14.102)? Compare Eqs. (14.101) and (14.104) and derive the connection between the runner ϕ_a and the `fuseOut` function (which exists for any functor R):

$$\text{fo}_R = r \to a \to r \triangleright \phi_a^{\uparrow R} \quad .$$

Then we can complete the derivation by rewriting the law (14.102):

$$\begin{aligned}
\text{expect to equal } m: \quad & m^{:M^{R^B}} \triangleright \text{fi}_R \fatsemi \text{fo}_R = (m \triangleright \text{fi}_R) \triangleright \underline{\text{fo}_R} \\
\text{use Eq. (14.101)}: \quad & = a \to m \triangleright \underline{\text{fi}_R} \triangleright \phi_a^{\uparrow R} \\
\text{use Eq. (14.103)}: \quad & = a \to m \triangleright \text{pu}_M^{\uparrow R \uparrow M} \fatsemi \text{pu}_R \fatsemi \underline{\text{ftn}_T \fatsemi \phi_a^{\uparrow R}} \\
\text{use Eq. (14.105)}: \quad & = a \to m \triangleright \text{pu}_M^{\uparrow R \uparrow M} \fatsemi \underline{\text{pu}_R \fatsemi \phi_a^{\uparrow R \uparrow M \uparrow R} \fatsemi \phi_a^{\uparrow R}} \fatsemi \text{ftn}_R \\
\text{naturality law of } \text{pu}_R: \quad & = a \to m \triangleright \text{pu}_M^{\uparrow R \uparrow M} \fatsemi \phi_a^{\uparrow R \uparrow M} \fatsemi \phi_a \fatsemi \underline{\text{pu}_R \fatsemi \text{ftn}_R} \\
\text{left identity law of } R: \quad & = a \to m \triangleright \underline{(\text{pu}_M \fatsemi \phi_a)}^{\uparrow R \uparrow M} \fatsemi \phi_a \\
\text{monad morphism identity law of } \phi_a: \quad & = a \to m \triangleright \underline{\phi_a} \\
\text{definition (14.104) of } \phi_a: \quad & = a \to m(a) = m \quad .
\end{aligned}$$

Here we used the monad morphism identity law for ϕ_a:

$$\text{pu}_M \fatsemi \phi_a = \big(x \to (_ \to x)\big) \fatsemi (m \to a \triangleright m) = (x \to a \triangleright (_ \to x)) = (x \to x) = \text{id} \quad .$$

Example 14.9.3.4 Show that the functions `fuseIn` and `fuseOut` are not always inverses: for some R,

$$\text{fo}_R \fatsemi \text{fi}_R \neq \text{id} \quad .$$

Solution Consider the rigid monads $P^A \stackrel{\text{def}}{=} Z \to A$ and $R^A \stackrel{\text{def}}{=} (A \to Q) \to A$, where Q and Z are fixed types. Since all rigid monads are rigid functors, it follows that the monads P and R have methods fi_P, fo_P, fi_R, fo_R satisfying the nondegeneracy law (14.102). It turns out that additionally $\text{fo}_P \fatsemi \text{fi}_P = \text{id}$, i.e., the methods fo_P and fi_P *are* inverses of each other, but $\text{fo}_R \fatsemi \text{fi}_R \neq \text{id}$.

To show that $\text{fo}_P \fatsemi \text{fi}_P = \text{id}$, consider the type signatures of fo_P and fi_P:

$$\begin{aligned} \text{fo}_P &: P^{A\to B} \to A \to P^B \cong (Z \to A \to B) \to (A \to Z \to B) \quad , \\ \text{fi}_P &: (A \to P^B) \to P^{A\to B} \cong (A \to Z \to B) \to (Z \to A \to B) \quad . \end{aligned}$$

Fully parametric implementations of these functions are derived uniquely from type signatures. The functions fo_P and fi_P switch the curried arguments of types A and Z of a function that returns values of type B. It is clear that these two functions are inverses of each other. To show this directly, consider the type signature of $\text{fo}_P \fatsemi \text{fi}_P$:

$$\left(\text{fo}_P \fatsemi \text{fi}_P\right) : P^{A\to B} \to P^{A\to B} \cong (Z \to A \to B) \to (Z \to A \to B) \quad .$$

There is only one fully parametric implementation for this type signature: the identity function $\text{id}^{Z\to A\to B}$.

For the monad R, the type signatures of fi_R and fo_R are:

$$\begin{aligned} \text{fi}_R &: (A \to (B \to Q) \to B) \to ((A \to B) \to Q) \to A \to B \quad , \\ \text{fo}_R &: (((A \to B) \to Q) \to A \to B) \to A \to (B \to Q) \to B \quad . \end{aligned}$$

The implementations are again derived uniquely from type signatures:

$$\begin{aligned} \text{fi}_R\left(f^{:A\to(B\to Q)\to B}\right) &= x^{:(A\to B)\to Q} \to a^{:A} \to f(a)(b^{:B} \to x(_ \to b)) \quad , \\ \text{fo}_R\left(g^{:((A\to B)\to Q)\to A\to B}\right) &= a^{:A} \to y^{:B\to Q} \to g(h^{:A\to B} \to y(h(a)))(a) \quad . \end{aligned}$$

We notice that the implementation of fi_R uses a constant function, $(_ \to b)$, which is likely to lose information. Indeed, while $\text{fi}_R \fatsemi \text{fo}_R = \text{id}$ as it must be due to the rigid nondegeneracy law, we find:

$$\begin{aligned} \text{expect } \textit{not} \text{ to equal } g: \quad & g \triangleright \text{fo}_R \fatsemi \text{fi}_R = (g \triangleright \text{fo}_R) \triangleright \underline{\text{fi}_R} \\ \text{definition of } \text{fi}_R: \quad & = x \to a \to \underline{\text{fo}_R}(g)(a)(b \to x(_ \to b)) \\ \text{definition of } \text{fo}_R: \quad & = x \to a \to g\left(h^{:A\to B} \to h(a) \triangleright \underline{(b \to} x(_ \to b))\right)(a) \\ \text{apply to argument } b: \quad & = x \to a \to g\left(h \to x(_ \to h(a))\right)(a) \quad . \end{aligned}$$

We cannot simplify the last line any further: the functions g, h, and x are unknown, and we cannot calculate symbolically, say, the value of $x(_ \to h(a))$. If the last line were equal to g, we would expect it to be $x \to a \to g(x)(a)$. The difference is in the first argument of g: we have $h \to x(_ \to h(a))$ instead of x. The two last expressions are not always equal. They would be equal if we had:

$$(h \to x(h)) = (h \to x(k \to h(k))) \quad ,$$

instead of $h \to x(_ \to h(a))$. Consider again the argument of x in the two last expressions: $k \to h(a)$ instead of $k \to h(k)$. Since h is not always a constant function (h is an arbitrary function of type $A \to B$), the two expressions $k \to h(a)$ and $k \to h(k)$ are generally not equal. So, we must conclude that $\text{fo}_R \fatsemi \text{fi}_R \neq \text{id}$. □

Since all rigid monads are rigid functors, we can reuse all the rigid monad constructions to obtain new rigid functors. The following statement shows a construction of rigid functors that does not assume any monadic properties.

Statement 14.9.3.5 The functor $S^\bullet \stackrel{\text{def}}{=} H^\bullet \to P^\bullet$ is rigid when H is any contrafunctor and P is any rigid functor. (Note that P does not need to be a monad.)

Proof We assume that fi_P and fo_P are known and satisfy the nondegeneracy law (14.102). The function fi_S is then defined by:

$$\text{fi}_S : \left(A \to H^B \to P^B\right) \to H^{A\to B} \to P^{A\to B} \quad ,$$
$$\text{fi}_S \stackrel{\text{def}}{=} f^{:A\to H^B\to P^B} \to h^{:H^{A\to B}} \to \text{fi}_P\left(a \to f(a)\left(h \triangleright (b \to _ \to b)^{\downarrow H}\right)\right) \quad ,$$

or equivalently, using the pipe notation:

$$h \triangleright \text{fi}_S(f) = \left(a \to h \triangleright (b \to _ \to b)^{\downarrow H} \triangleright f(a)\right) \triangleright \text{fi}_P \quad .$$

Let us write the definition of fo_S as well:

$$\text{fo}_S : \left(H^{A\to B} \to P^{A\to B}\right) \to A \to H^B \to P^B \quad ,$$
$$\text{fo}_S \stackrel{\text{def}}{=} g^{:H^{A\to B}\to P^{A\to B}} \to a^{:A} \to h^{:H^B} \to \text{fo}_P\left(g\left(h \triangleright (p^{:A\to B} \to p(a))^{\downarrow H}\right)\right)(a) \quad ,$$

or equivalently, using the pipe notation:

$$\text{fo}_S(g) = a \triangleright \left(h \triangleright (p \to p(a))^{\downarrow H} \fatsemi g \fatsemi \text{fo}_P\right) \quad .$$

To verify the nondegeneracy law for S, apply both sides to some arguments; we expect $f \triangleright (\text{fi}_S \fatsemi \text{fo}_S)$ to equal f for an arbitrary $f : A \to H^B \to P^B$. To compare values, we need to apply both sides further to some arguments $a : A$ and $h : H^B$. So we expect the following expression to equal $f(a)(h)$:

$$\begin{aligned}
& (f \triangleright \text{fi}_S \fatsemi \text{fo}_S)\,(a)(h) = (f \triangleright \text{fi}_S \triangleright \underline{\text{fo}_S})(a)(h) \\
\text{expand } \text{fo}_S : \quad & = a \triangleright \left(h \triangleright (p \to a \triangleright p)^{\downarrow H} \triangleright \underline{\text{fi}_S(f)} \triangleright \text{fo}_P\right) \\
\text{expand } \text{fi}_S : \quad & = a \triangleright \left(\left(a \to h \triangleright \underline{(p \to p(a))^{\downarrow H} \fatsemi (b \to _ \to b)^{\downarrow H}} \fatsemi f(a)\right) \triangleright \text{fi}_P \triangleright \text{fo}_P\right) \\
\text{compose } {}^{\downarrow H} : \quad & = a \triangleright \left(\left(a \to h \triangleright \underline{((b \to _ \to b) \fatsemi (p \to p(a)))^{\downarrow H}} \fatsemi f(a)\right) \triangleright \text{fi}_P \underline{\triangleright} \text{fo}_P\right) \quad .
\end{aligned}$$

Computing the function composition:

$$(b \to _ \to b) \fatsemi (p \to p(a)) = (b \to (_ \to b)(a)) = (b \to b) = \text{id}$$

and using the nondegeneracy law $\text{fi}_P \fatsemi \text{fo}_P = \text{id}$, we can simplify further:

$$\begin{aligned}
& a \triangleright \left(\left(a \to h \triangleright \underline{((b \to _ \to b) \fatsemi (p \to p(a)))^{\downarrow H}} \fatsemi f(a)\right) \triangleright \text{fi}_P \underline{\triangleright} \text{fo}_P\right) \\
\text{identity law for } H : \quad & = a \triangleright \left(\left(a \to h \triangleright f(a)\right) \triangleright \underline{\text{fi}_P \fatsemi \text{fo}_P}\right) \\
\text{non-degeneracy} : \quad & = \underline{a \triangleright \left(a \to} h \triangleright f(a)\right) = h \triangleright f(a) \quad .
\end{aligned}$$

This equals $f(a)(h)$, as required.

Statement 14.9.3.6 Any rigid functor R is pointed. The required `pure` method can be defined by:

$$\text{pu}_R(x^{:A}) \stackrel{\text{def}}{=} \text{id}^{:R^A\to R^A} \triangleright \text{fi}_R \triangleright (_ \to x)^{\uparrow R} \quad .$$

The corresponding "wrapped unit" value wu_R of type $R^{\mathbb{1}}$ is computed as:

$$\text{wu}_R \stackrel{\text{def}}{=} \text{pu}_R(1) = \text{id} \triangleright \text{fi}_R \triangleright (_ \to 1)^{\uparrow R} \quad . \tag{14.106}$$

Proof The method $\text{fi}_R : (X \to R^Y) \to R^{X\to Y}$ with type parameters $X = R^A$ and $Y = A$ is applied to the identity function $\text{id} : R^A \to R^A$, considered as a value of type $X \to R^Y$. The result is:

$$(\text{id} \triangleright \text{fi}_R) : R^{R^A\to A} \quad .$$

This value is transformed via the lifted constant function $(_ \to x)^{\uparrow R}$, which takes a value of type $R^{R^A \to A}$ and returns a value of type R^A. The resulting code can be written as:

$$\text{pu}_R(x^{:A}) \stackrel{\text{def}}{=} \text{id} \triangleright \text{fi}_R \triangleright (_^{:R^A \to A} \to x)^{\uparrow R} \quad .$$

The function pu_R defined in this way is a natural transformation since fi_R is one. Applying pu_R to a unit value, we obtain a selected value wu_R of type $R^{\mathbb{1}}$. $\square$

The next statement shows that wu_R is the *only* distinct value of the type $R^{\mathbb{1}}$. This means, in particular, that a rigid functor cannot be a disjunctive type defined with more than one constructor, such as $\mathbb{1} + A$ or List^A. This is the motivation for the name "rigid": the data structure R^A must have a fixed "shape" and cannot be a disjunction of several variants.

Statement 14.9.3.7 If a functor R is rigid then the type $R^{\mathbb{1}}$ is equivalent to the unit type: $R^{\mathbb{1}} \cong \mathbb{1}$. The value wu_R defined by Eq. (14.106) is the only distinct value of type $R^{\mathbb{1}}$. The isomorphism between the types $\mathbb{1}$ and $R^{\mathbb{1}}$ is the function $(_^{:\mathbb{1}} \to \text{wu}_R)$.

Proof The trick is to apply both sides of the nondegeneracy law ($\text{fi}_R \fatsemi \text{fo}_R = \text{id}$) to the identity function of type $R^{\mathbb{1}} \to R^{\mathbb{1}}$. To adapt the type parameters, consider the type signature of $\text{fi}_R \fatsemi \text{fo}_R$:

$$(\text{fi}_R \fatsemi \text{fo}_R) : (A \to R^B) \to (A \to R^B) \quad ,$$

and set $A = R^{\mathbb{1}}$ and $B = \mathbb{1}$. The left-hand side of the law can be now applied to the identity function $\text{id} : R^{\mathbb{1}} \to R^{\mathbb{1}}$, which yields a value of type $R^{\mathbb{1}} \to R^{\mathbb{1}}$, i.e., a function:

$$f_1 : R^{\mathbb{1}} \to R^{\mathbb{1}} \quad , \qquad f_1 \stackrel{\text{def}}{=} \text{fo}_R(\text{fi}_R(\text{id})) \quad .$$

We will show that f_1 is a constant function, $f_1 = (_ \to \text{wu}_R)$, always returning the same value wu_R defined in Statement 14.9.3.6. However, the right-hand side of the nondegeneracy law applied to `id` is the identity function of type $R^{\mathbb{1}} \to R^{\mathbb{1}}$. So, the nondegeneracy law means that $f_1 = \text{id}$. If the identity function of type $R^{\mathbb{1}} \to R^{\mathbb{1}}$ always returns the same value (wu_R), it will mean that wu_R is the only distinct value of the type $R^{\mathbb{1}}$.

We note that for any fixed type A, the function type $A \to \mathbb{1}$ is equivalent to the unit type ($\mathbb{1}$). This is because there exists only one function of type $A \to \mathbb{1}$, namely $(_ \to 1)$. The code of that function is uniquely determined by its type signature. In other words, there exists only one distinct value of the type $A \to \mathbb{1}$, and that value is $(_^{:A} \to 1)$.

The isomorphism between the types $A \to \mathbb{1}$ and $\mathbb{1}$ is realized by the functions $u : \mathbb{1} \to A \to \mathbb{1}$ and $v : (A \to \mathbb{1}) \to \mathbb{1}$. The code of these functions is also uniquely determined by their type signatures:

$$u = \left(1 \to _^{:A} \to 1\right) \quad , \qquad v = \left(_^{:A \to \mathbb{1}} \to 1\right) \quad .$$

To prove that f_1 is a constant function, we apply fi_R to the identity function of type $R^{\mathbb{1}} \to R^{\mathbb{1}}$ and get a value that we denote by g:

$$g : R^{R^{\mathbb{1}} \to \mathbb{1}} \quad , \qquad g \stackrel{\text{def}}{=} \text{id} \triangleright \text{fi}_R \quad .$$

Since the type $R^{\mathbb{1}} \to \mathbb{1}$ is equivalent to $\mathbb{1}$, the type $R^{R^{\mathbb{1}} \to \mathbb{1}}$ is equivalent to $R^{\mathbb{1}}$. To use this equivalence explicitly, we need to lift the isomorphisms u and v into the functor R. The isomorphism will then map $g : R^{R^{\mathbb{1}} \to \mathbb{1}}$ to some $g_1 : R^{\mathbb{1}}$ by:

$$g_1 \stackrel{\text{def}}{=} g \triangleright v^{\uparrow R} \quad .$$

Substituting the definitions of g, v, and wu_R, we find that actually $g_1 = \text{wu}_R$:

$$g_1 = \text{id} \triangleright \text{fi}_R \triangleright v^{\uparrow R} = \text{id} \triangleright \text{fi}_R \triangleright (_ \to 1)^{\uparrow R} = \text{wu}_R \quad .$$

We can now map g_1 back to g via the lifted isomorphism u:

$$\begin{aligned} g &= g_1 \triangleright u^{\uparrow R} = \text{wu}_R \triangleright u^{\uparrow R} \\ \text{definition of } u: \quad &= \text{wu}_R \triangleright \left(1 \to _ \to 1\right)^{\uparrow R} \quad . \end{aligned} \tag{14.107}$$

Now we compute $f_1 \stackrel{\text{def}}{=} \text{fo}_R(g)$ as:

$$\begin{aligned}
& f_1 = \text{fo}_R(g) = g \triangleright \text{fo}_R \\
\text{use Eq. (14.101)}: \quad & = a^{:R^{\mathbb{1}}} \to g \triangleright \left(f^{:A\to\mathbb{1}} \to f\,(a) \right)^{\uparrow R} \\
\text{use Eq. (14.107)}: \quad & = a \to \text{wu}_R \triangleright (1 \to _^{:A} \to 1)^{\uparrow R} \triangleright \left(f^{:A\to\mathbb{1}} \to f\,(a) \right)^{\uparrow R} \\
\text{composition under } {}^{\uparrow R}: \quad & = a \to \text{wu}_R \triangleright \left(1 \to (_^{:A} \to 1)(a) \right)^{\uparrow R} = a \to \text{wu}_R \triangleright \underline{(1 \to 1)}^{\uparrow R} \\
(1 \to 1) \text{ is identity}: \quad & = (a \to \text{wu}_R \triangleright \text{id}) = (a^{:R^{\mathbb{1}}} \to \text{wu}_R) = (_^{:R^{\mathbb{1}}} \to \text{wu}_R) \quad .
\end{aligned}$$

So, $f_1 \stackrel{\text{def}}{=} \text{fo}_R(\text{fi}_R(\text{id}))$ is a function of type $R^{\mathbb{1}} \to R^{\mathbb{1}}$ that ignores its argument and always returns the same value wu_R.

By virtue of the nondegeneracy law, we get $f_1 = \text{fo}_R(g) = \text{id}$. So, the identity function $\text{id}^{:R^{\mathbb{1}}\to R^{\mathbb{1}}}$ always returns the same value wu_R. Applying this function to an arbitrary value $x : R^{\mathbb{1}}$, we get:

$$x = x \triangleright \text{id} = x \triangleright \text{fo}_R(g) = x \triangleright (_ \to \text{wu}_R) = \text{wu}_R \quad .$$

It means that all values x of type $R^{\mathbb{1}}$ are equal to wu_R. So the function $_^{:\mathbb{1}} \to \text{wu}_R$ is indeed an isomorphism between the types $\mathbb{1}$ and $R^{\mathbb{1}}$.

Statement 14.9.3.8 The product of rigid functors is a rigid functor.

Proof We assume that two rigid functors P and Q are given, with their methods fi_P and fi_Q satisfying the nondegeneracy law. The functor product $R \stackrel{\text{def}}{=} P \times Q$ needs its own fi_R method, with the type signature:

$$\text{fi}_R : (A \to P^B \times Q^B) \to P^{A\to B} \times Q^{A\to B} \quad .$$

To implement fi_R, note that the function type $A \to P \times Q$ is isomorphic to $(A \to P) \times (A \to Q)$. The isomorphism is given by a `split` function, defined as:

$$\text{split} : (A \to B \times C) \to (A \to B) \times (A \to C) \quad , \qquad \text{split} \stackrel{\text{def}}{=} f^{:A\to B\times C} \to (f \,\text{⨾}\, \pi_1) \times (f \,\text{⨾}\, \pi_2) \quad .$$

Then the method fi_R is written more concisely:

$$\text{fi}_R \stackrel{\text{def}}{=} \text{split} \,\text{⨾}\, (\text{fi}_P \boxtimes \text{fi}_Q) \quad .$$

To check the nondegeneracy law ($\text{fi}_R \,\text{⨾}\, \text{fo}_R = \text{id}$), we need the code of fo_R. It is convenient to define fo_R via a `merge` function:

$$\text{merge} : (A \to B) \times (A \to C) \to A \to B \times C \quad , \qquad \text{merge} \stackrel{\text{def}}{=} f^{:A\to B} \times g^{:A\to C} \to a^{:A} \to f(a) \times g(a) \quad .$$

The code of fo_R is then written as:

$$\text{fo}_R : P^{A\to B} \times Q^{A\to B} \to A \to P^B \times Q^B \quad , \qquad \text{fo}_R \stackrel{\text{def}}{=} (\text{fo}_P \boxtimes \text{fo}_Q) \,\text{⨾}\, \text{merge} \quad .$$

The functions `split` and `merge` are mutual inverses. Now we can verify the nondegeneracy law:

$$\begin{aligned}
& \text{fi}_R \,\text{⨾}\, \text{fo}_R = \text{split} \,\text{⨾}\, \underline{(\text{fi}_P \boxtimes \text{fi}_Q) \,\text{⨾}\, (\text{fo}_P \boxtimes \text{fo}_Q)} \,\text{⨾}\, \text{merge} \\
& = \text{split} \,\text{⨾}\, \left(\underline{(\text{fi}_P \,\text{⨾}\, \text{fo}_P)} \boxtimes \underline{(\text{fi}_Q \,\text{⨾}\, \text{fo}_Q)} \right) \,\text{⨾}\, \text{merge} = \text{split} \,\text{⨾}\, \underline{(\text{id} \boxtimes \text{id})} \,\text{⨾}\, \text{merge} = \text{split} \,\text{⨾}\, \text{merge} = \text{id} \quad .
\end{aligned}$$

Statement 14.9.3.9 The composition of rigid functors is a rigid functor.

Proof We will show that the nondegeneracy law ($\text{fi}_R \,\text{⨾}\, \text{fo}_R = \text{id}$) holds for the functor $R^\bullet \stackrel{\text{def}}{=} P^{Q^\bullet}$ as long as P, Q are rigid. Begin by defining the methods fi_R and fo_R, assuming that the corresponding methods of P, Q are known and satisfy the nondegeneracy law. We write:

$$\begin{aligned}
& \text{fi}_R : (A \to P^{Q^B}) \to P^{Q^{A\to B}} \quad , \qquad \text{fi}_R \stackrel{\text{def}}{=} \text{fi}_P \,\text{⨾}\, \text{fi}_Q^{\uparrow P} \quad , \\
& \text{fo}_R : P^{Q^{A\to B}} \to A \to P^{Q^B} \quad , \qquad \text{fo}_R \stackrel{\text{def}}{=} \text{fo}_Q^{\uparrow P} \,\text{⨾}\, \text{fo}_P \quad .
\end{aligned}$$

To verify the nondegeneracy law for R:

$$\text{fi}_R \,\text{⨾}\, \text{fo}_R = \text{fi}_P \,\text{⨾}\, \text{fi}_Q^{\uparrow P} \,\text{⨾}\, \text{fo}_Q^{\uparrow P} \,\text{⨾}\, \text{fo}_P = \text{fi}_P \,\text{⨾}\, \underline{(\text{fi}_Q \,\text{⨾}\, \text{fo}_Q)}^{\uparrow P} \,\text{⨾}\, \text{fo}_P = \underline{\text{fi}_P \,\text{⨾}\, \text{fo}_P} = \text{id} \quad .$$

14.9.4 Monadically natural lifting of MTL operations ("`Monatron`")

Section 14.1.8 showed that MTL-style programming requires custom code for lifting monad operations to all supported monad stacks. M. Jaskelioff's "`Monatron`" library[21] is based on a systematic approach that lifts many known monad operations to most monad stacks, including stacks that contain a single monad with an incomplete or missing transformer (such as `Cont` or `IO`) located at the deep end of the stack. This section, based on the methods and results explained in a paper by M. Jaskelioff,[22] will show how `Monatron`'s liftings work.

Common type signatures for operations The first step is to transform all effectful operations into a standard type signature. Looking at Table 14.4, we find that the type signatures of known operations for various monads M fall into four patterns:

1. Type signature M^Q or $P \to M^Q$, where P and Q are fixed types.
2. Type signature $\forall A.\, M^A$ or $\forall A.\, F^A \to M^A$, where F is a functor that does not depend on M.
3. Type signature $\forall A.\, F^{M^A} \to M^A$, where F is a functor that does not depend on M.
4. A different type signature (the `callcc`, `collect`, `process`, and `filter` operations).

At first glance, the type signatures of different operations appear to be different and incompatible with each other. However, it turns out that almost all operations may be converted into a common type signature of the form $F^{M^A} \to M^A$. The following definition introduces some helpful names:

Definition 14.9.4.1 For a given monad M and a given functor F:

A **"fixed-type" operation** is a function of type $P \to M^Q$, where P and Q are fixed types.

A **"simple" operation** is a natural transformation with type signature $\forall A.\, F^A \to M^A$.

A **"standard" operation** is a natural transformation with type signature $\forall A.\, F^{M^A} \to M^A$.

□

According to this definition, the `State` monad's `get` and `set` operations are "fixed-type", the `Either` monad's `raise` operation is "simple" (with the functor F defined as the constant functor $F^A \stackrel{\text{def}}{=} E$), and the `List` monad's `append` operation is "standard" (with the functor $F^A \stackrel{\text{def}}{=} A \times A$).

The following statement shows that "fixed-type" operations are a subset of "simple" operations, while "simple" operations are a subset of "standard" operations.

Statement 14.9.4.2 **(a)** Any "fixed-type" operation $\text{fop} : P \to M^Q$ is equivalent to a "simple" operation `sop` with functor $F^A \stackrel{\text{def}}{=} P \times (Q \to A)$. The operations `fop` and `sop` are related by:

$$\text{sop} : P \times (Q \to A) \to M^A \quad , \quad \text{sop} = p^{:P} \times q^{:Q\to A} \to p \triangleright \text{fop} \triangleright q^{\uparrow M} \quad , \quad \text{fop} = p^{:P} \to \text{sop}^Q(p \times \text{id}^{Q\to Q}) \quad .$$

(b) Any "standard" operation $\text{op} : F^{M^A} \to M^A$ that satisfies the following compatibility law,

$$\begin{tikzcd} F^{M^{M^A}} \arrow[r, "\text{op}"] \arrow[d, "\text{ftn}_M^{\uparrow F}"'] & M^{M^A} \arrow[d, "\text{ftn}_M"] \\ F^{M^A} \arrow[r, "\text{op}"] & M^A \end{tikzcd} \qquad \text{op} \fatsemi \text{ftn}_M = \text{ftn}_M^{\uparrow F} \fatsemi \text{op} \quad , \tag{14.108}$$

is equivalent to a "simple" operation `sop` defined by:

$$\text{sop} : F^A \to M^A \quad , \quad \text{sop} \stackrel{\text{def}}{=} \text{pu}_M^{\uparrow F} \fatsemi \text{op} \quad , \quad \text{op} = \text{sop} \fatsemi \text{ftn}_M \quad .$$

Proof **(a)** Derive the type equivalence $P \to M^Q \cong \forall A.\, P \times (Q \to A) \to M^A$:

$$\forall A.\, \underline{P\times}(Q \to A) \to M^A \cong P \to \underline{\forall A.\, (Q \to A) \to M^A}$$

$$\text{covariant Yoneda identity :} \quad \cong P \to M^Q \quad .$$

(b) We will prove separately the two directions of the isomorphism.

[21] See https://www.fceia.unr.edu.ar/~mauro/pubs/monatron.pdf

[22] "Modular monad transformers", see https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.219.5365

Direction op → sop: Given that a "standard" operation $\text{op} : F^{M^A} \to M^A$ satisfying Eq. (14.108), we first define $\text{sop} \stackrel{\text{def}}{=} \text{pu}_M^{\uparrow F} ⨾ \text{op}$ and then compute a new "standard" operation $\text{op}' \stackrel{\text{def}}{=} \text{sop} ⨾ \text{ftn}_M$. The new operation `op'` is equal to the initial `op`:

$$\text{op}' = \text{sop} ⨾ \text{ftn}_M = \text{pu}^{\uparrow F} ⨾ \underline{\text{op} ⨾ \text{ftn}_M}$$

$$\text{use Eq. (14.108)}: \quad = \underline{\text{pu}^{\uparrow F} ⨾ \text{ftn}_M^{\uparrow F}} ⨾ \text{op} = (\underline{\text{pu} ⨾ \text{ftn}_M})^{\uparrow F} = \text{op} \quad .$$

Direction sop → op: Given a "simple" operation $\text{sop} : F^A \to M^A$, we first define $\text{op} \stackrel{\text{def}}{=} \text{sop} ⨾ \text{ftn}_M$ and then compute a new "simple" operation $\text{sop}' \stackrel{\text{def}}{=} \text{pu}^{\uparrow F} ⨾ \text{op}$. The new operation `sop'` equals `sop`:

$$\text{sop}' = \text{pu}^{\uparrow F} ⨾ \text{op} = \underline{\text{pu}^{\uparrow F} ⨾ \text{sop}} ⨾ \text{ftn}_M$$

$$\text{naturality law of sop}: \quad = \text{sop} ⨾ \underline{\text{pu}^{\uparrow M} ⨾ \text{ftn}_M} = \text{sop} \quad .$$

□

All operations shown in Table 14.4 (except `callcc`, `collect`, `process`, and `filter`) are either "fixed-type", "simple", or "standard", and so (by Statement 14.9.4.2) may be equivalently represented by some "standard" operations. Many operations with a "standard" type signature turn out to be equivalent to "simple" operations because they satisfy the compatibility law.

Example 14.9.4.3 The `List` monad's `append` operation has the type signature of the "standard" form $F^{\text{List}^A} \to \text{List}^A$ with $F^A \stackrel{\text{def}}{=} A \times A$. As shown in Exercise 10.2.9.11, `append` satisfies the compatibility law (14.108). So, Statement 14.9.4.2 shows that `append` is equivalent to a "simple" operation `sappend`:

$$\text{sappend} : A \times A \to \text{List}^A \quad , \qquad \text{sappend}\,(a_1 \times a_2) \stackrel{\text{def}}{=} [a_1, a_2] = \text{pu}_{\text{List}}(a_1) +\!\!+ \text{pu}_{\text{List}}(a_2) \quad ,$$

$$p^{:\text{List}^A} +\!\!+ q^{:\text{List}^A} = \text{ftn}_{\text{List}}([p, q]) = (p \times q) ▷ \text{sappend} ▷ \text{ftn}_{\text{List}} \quad .$$

□

Certain "standard" operations are *not* equivalent to "simple" ones:

Example 14.9.4.4 The `Writer` monad's `clear` operation (erasing the message log) is defined by:

$$\text{clear} : M^A \to M^A \cong A \times W \to A \times W \quad , \qquad \text{clear} \stackrel{\text{def}}{=} a^{:A} \times _^{:W} \to a \times e_W \quad .$$

The type signature $M^A \to M^A$ is of the "standard" form $F^{M^A} \to M^A$ with $F^A \stackrel{\text{def}}{=} A$. However, `clear` does not satisfy the compatibility law (14.108) and is not equivalent to a "simple" operation. If it were, the corresponding "simple" operation would be a natural transformation of type $F^A \to M^A \cong A \to A \times W$. Such a natural transformation must be implemented as $\text{sop} : a^{:A} \to a \times w_0$ with some fixed value $w_0^{:W}$ that is independent of a. The corresponding "standard" operation is:

$$\text{op} : M^A \to M^A \quad , \qquad \text{op} \stackrel{\text{def}}{=} \text{sop} ⨾ \text{ftn}_M = (a \times w \to a \times w \times w_0) ⨾ \text{ftn}_M = a \times w \to a \times (w \oplus w_0) \quad .$$

This operation appends to the log instead of clearing it. So, `op` cannot equal `clear`. □

To build an intuition for the meaning of "simple" and "standard", consider a "standard" operation $\text{op} : F^{M^A} \to M^A$, where F^A contains a few values of type A (such as $F^A \stackrel{\text{def}}{=} A \times A$ as in the `List` monad's `append` operation). A "simple" operation $\text{sop} : F^A \to M^A$ computes a new M-effect value from a few values of type A without using any previously given M-effect values. A "standard" operation `op` transforms a few effectful values of type M^A into a new effectful value. If `op` were equivalent to a "simple" operation `sop`, we would have $\text{op} = \text{sop} ⨾ \text{ftn}_M$. It follows that `op` computes a new M-effect by flattening the previously given M-effects and possibly using some values of type A.

As a heuristic summary, we may say that a "simple" operation creates new effect values from ordinary, non-effectful values. A "standard" operation can transform effects in a way that is *not* equivalent to flattening (as the example with `clear` shows). "Standard" operations that are equivalent to "simple"operations cannot perform arbitrary transformations of effects: the given effects must be flattened and combined with a new effect value, but may not be transformed in any other way.

Putting the "non-standard" operations `callcc`, `collect`, `process`, and `filter` temporarily aside, we now focus on the methods for lifting "simple" and "standard" operations to monad stacks.

The goal is to lift a "standard" operation $\text{op}_M : F^M \rightsquigarrow M$ from a monad M to any monad stack P containing M. We assume that a lawful monad morphism $p : M \rightsquigarrow P$ is given. We expect that the new operation, $\text{op}_P : F^P \rightsquigarrow P$, should correctly perform the functionality of the operation op_M in the monad M embedded within P.

For example, since the `List` monad's `append` operation appends two lists, we expect that the corresponding lifted operation (append_P) should correctly append lists embedded within the stack P. The `Writer` monad's `clear` operation clears the log, so we expect that the corresponding lifted operation (clear_P) also clears the log embedded within the stack P.

To express this requirement formally, we note that the monad morphism p correctly embeds the monad methods of M into P. So, we may require that the transformation of effects by `op` should commute with the mapping by p. This requirement is written as a monadic naturality law:

$$\text{op}_M \fatsemi p = p^{\uparrow F} \fatsemi \text{op}_P \quad . \tag{14.109}$$

$$\begin{tikzcd} F^{M^A} \arrow[r, "\text{op}_M"'] \arrow[d, "p^{\uparrow F}"'] & M^A \arrow[d, "p"] \\ F^{P^A} \arrow[r, "\text{op}_P"] & P^A \end{tikzcd}$$

The goal of this section is to derive monadically natural liftings of operations.

Lifting "simple" operations Section 14.1.8 showed that the `State` monad's `get` and `set` operations are lifted to any monad stack by composing with a lift function. This method works for any "simple" operation $\text{sop}_M : F^A \to M^A$ and an arbitrary monad stack P that has a lift function $p : M^A \to P^A$. The result of lifting is a "simple" operation (sop_P) defined by:

$$\text{sop}_P : F^A \to P^A \quad , \qquad \text{sop}_P \stackrel{\text{def}}{=} \text{sop}_M \fatsemi p \quad . \tag{14.110}$$

The following statement shows how to express this lifting in terms of "standard" operations. It also proves that the "standard" counterpart of sop_P always satisfies the monadic naturality law.

Statement 14.9.4.5 Given a "simple" M-operation $\text{sop}_M : F^A \to M^A$ and a monad morphism $p : M \rightsquigarrow P$ between two monads M and P, define the operation sop_P by Eq. (14.110). Then the "standard" operation $\text{op}_P \stackrel{\text{def}}{=} \text{sop}_P \fatsemi \text{ftn}_P$ corresponding to sop_P is expressed via op_M by:

$$\text{op}_P = \text{pu}_M^{\uparrow F} \fatsemi \text{op}_M \fatsemi p \fatsemi \text{ftn}_P \tag{14.111}$$

and satisfies the monadic naturality law (14.109). Here, we defined $\text{op}_M \stackrel{\text{def}}{=} \text{sop}_M \fatsemi \text{ftn}_M$.

Proof We substitute the definitions of sop_P and the relationship between sop_M and op_M:

$$\text{op}_P = \underline{\text{sop}_P} \fatsemi \text{ftn}_P = \underline{\text{sop}_M} \fatsemi p \fatsemi \text{ftn}_P = \text{pu}_M^{\uparrow F} \fatsemi \text{op}_M \fatsemi p \fatsemi \text{ftn}_P \quad .$$

To verify the monadic naturality law (14.109):

$$\begin{aligned} \text{expect to equal } p^{\uparrow F} \fatsemi \text{op}_P : \quad & \underline{\text{op}_M} \fatsemi p = \text{sop}_M \fatsemi \underline{\text{ftn}_M \fatsemi p} \\ \text{monad morphism composition law of } p : \quad & = \underline{\text{sop}_M \fatsemi p^{\uparrow M}} \fatsemi p \fatsemi \text{ftn}_P \\ \text{naturality law of } \text{sop}_M : \quad & = p^{\uparrow F} \fatsemi \underline{\text{sop}_M \fatsemi p} \fatsemi \text{ftn}_P = p^{\uparrow F} \fatsemi \underline{\text{sop}_P \fatsemi \text{ftn}_P} = p^{\uparrow F} \fatsemi \text{op}_P \quad . \end{aligned}$$

□

The method of Statement 14.9.4.5 allows us to lift a "simple" operation from a monad M to any monad stack P containing M, as long as we have a lift function $p : M \rightsquigarrow P$. The lift p will exist if the monad M is at the deep end of the stack, or if M's transformer supports a base lift. Accordingly, one must place the continuation monad, the search monad, the codensity monad, and task-like monads (such as `IO`) at the deep end of the stack, since those monads do not support base lifts.

The same method allows us to lift a "standard" operation if it is equivalent to a "simple" operation. The condition for that equivalence is the compatibility law (14.108). If a "standard" operation `op` satisfies that law, the corresponding lifted operation op_P is expressed by Eq. (14.111).

Lifting "standard" operations Some "standard" operations fail the compatibility law (14.108). So, they cannot be reduced to a "simple" operation and require a different method of lifting.

We begin with the task of lifting a "standard" operation $\text{op} : F^{M^A} \to M^A$ to a stack P where the monad M is located at the deep end. So, we assume that $P = L \triangleleft M$ for some base monad (or stack) L. The lifting of `op` to P turns out to be complicated, so we split the procedure into four steps:

1. Convert `op` into an equivalent "simple" operation (`scop`) in the codensity monad on M (see Section 14.8.4). For brevity, we will denote the codensity monad $\text{Cod}^{M,\bullet}$ equivalently by Cod^M.
2. Lift `scop` to the stack $P' \stackrel{\text{def}}{=} L \triangleleft \text{Cod}^M$ that contains the codensity monad instead of M.
3. Transform the stack, replacing the codensity monad Cod^M by the original monad M.
4. Derive the code for the new operation (op_P).

We will now go through these steps in detail.

The first step introduces the codensity monad Cod^M. It turns out that the type of natural transformations $F^{M^A} \to M^A$ (the type of "standard" operations in the monad M) is *equivalent* to the type of natural transformations $F^A \to \text{Cod}^{M,A}$ (the type of "simple" operations in the codensity monad on M). To prove this, we will derive a more general property that does not depend on any monads:

Statement 14.9.4.6 For any functors F, K, L, the type of natural transformations $\forall A.\, F^{K^A} \to L^A$ is equivalent to the following type of natural transformations:[23]

$$\forall (A, R).\, F^A \to (A \to K^R) \to L^R \quad ,$$

where naturality laws are required to hold with respect to both A and R.

Proof Swap the curried arguments in the type signature $F^A \to (A \to K^R) \to L^R$:

$$\forall (A, R).\, F^A \to (A \to K^R) \to L^R \cong \forall (A, R).\, (A \to K^R) \to F^A \to L^R = \forall R.\, \left(\forall A.\, (A \to K^R) \to H^{A,R}\right) \quad ,$$

where we define $H^{A,R} \stackrel{\text{def}}{=} F^A \to L^R$. The type constructor $H^{A,R}$ is contravariant in A. Since we assume the naturality laws to hold, we are dealing with functions of type $\forall A.\, (A \to K^R) \to H^{A,R}$ that are natural transformations between contrafunctors. So, we may use the contravariant Yoneda identity (Statement 13.7.1.2) with respect to A. With that, we obtain the following type equivalence:

$$\forall A.\, (A \to K^R) \to H^{A,R} \cong H^{K^R,R} \quad .$$

It remains to substitute the definition of $H^{\bullet,\bullet}$ and to rename R to A:

$$\forall R.\, H^{K^R,R} = \forall R.\, F^{K^R} \to L^R = \forall A.\, F^{K^A} \to L^A \quad .$$

□

Using Statement 14.9.4.6 with $K = L = M$, we find that a "standard" operation `op` in a monad M is equivalent to a "simple" operation `scop` in the codensity monad Cod^M. We define `scop` via `op` by:

$$\text{scop} : F^A \to (A \to M^R) \to M^R \quad , \qquad \text{scop} \stackrel{\text{def}}{=} f^{:F^A} \to k^{:A \to M^R} \to f \triangleright k^{\uparrow F} \triangleright \text{op}^R \quad . \tag{14.112}$$

We can lift `scop` to the stack $P' \stackrel{\text{def}}{=} L \triangleleft \text{Cod}^M$ that contains the codensity monad Cod^M instead of M. The result is a "simple" operation ($\text{scop}_{P'}$) defined by:

$$\text{scop}_{P'} : F^A \to (L \triangleleft \text{Cod}^{M,\bullet})^A \quad , \qquad \text{scop}_{P'} \stackrel{\text{def}}{=} \text{scop} \,⨾\, \text{flift}_L^{\text{Cod}^M} \quad .$$

It remains somehow to convert the operation $\text{scop}_{P'}$ into a corresponding operation in the stack P.

Since the monad M is injectively embedded within Cod^M via the functions `inC` and `outC` (see Statement 14.8.4.3), we may use L's `frun` to transform between P and P' in both directions:

$$\text{frun}_L(\text{inC}) : L \triangleleft M \rightsquigarrow L \triangleleft \text{Cod}^M \cong P \rightsquigarrow P' \quad , \qquad \text{frun}_L(\text{outC}) : L \triangleleft \text{Cod}^M \rightsquigarrow L \triangleleft M \cong P' \rightsquigarrow P \quad .$$

[23] For some (but not all) functors K and L, the type $\forall R.\, (A \to K^R) \to L^R$ implements the construction known in category theory as the **right Kan extension** of L along K. See https://bartoszmilewski.com/2017/04/17/kan-extensions/

One way of converting $\text{scop}_{P'}$ into a P-operation would be to compose with $\text{flift}_L(\text{outC})$, which creates a "simple" operation sop_P of type $F^A \to P^A$. However, the resulting operation would not correctly extend $\texttt{op}$'s effect transformation to the stack P. To see why, we rewrite sop_P as:

$$\begin{aligned}
&\text{sop}_P \stackrel{\text{def}}{=} \text{scop}_{P'} \fatsemi \text{flift}_L(\text{outC}) = \text{scop} \fatsemi \underline{\text{flift}_L \fatsemi \text{frun}_L}(\text{outC}) \\
\text{monadic naturality law of flift}_L: \quad &= \text{scop} \fatsemi \text{outC} \fatsemi \text{flift}_L = (f \to k \to f \triangleright k^{\uparrow F} \triangleright \text{op}) \fatsemi \text{outC} \fatsemi \text{flift}_L \\
&= (\underline{f \to f \triangleright \text{pu}_M^{\uparrow F}} \fatsemi \text{op}) \fatsemi \text{flift}_L = \text{pu}_M^{\uparrow F} \fatsemi \text{op} \fatsemi \text{flift}_L \quad .
\end{aligned}$$

We find that the operation sop_P is the lifted version of $\text{sop} \stackrel{\text{def}}{=} \text{pu}_M^{\uparrow F} \fatsemi \text{op}$, which is the "simple" counterpart of $\texttt{op}$. However, by assumption, $\texttt{op}$ fails to obey the compatibility law and is not equivalent to $\texttt{sop}$. In other words, the operation $\texttt{sop}$ does not perform the same transformation of M-effects as $\texttt{op}$. So, the operation sop_P fails to be a correct lifting of $\texttt{op}$.

In this derivation, we used the monadic naturality law (14.5) of $\texttt{flift}$ with $\phi \stackrel{\text{def}}{=} \text{outC}$, although $\texttt{outC}$ is *not* a monad morphism. This is justified because (as we have proved in this chapter) all monad transformers that support $\texttt{frun}$ will also satisfy the monadic naturality law (14.5) of $\texttt{flift}$ with ϕ being any natural transformation, not necessarily a monad morphism.

To obtain a correct lifting, the equivalence between $\texttt{op}$ and $\texttt{scop}$ must be expressed differently:

Statement 14.9.4.7 Given a "standard" operation $\texttt{op}$ in a monad M, define $\texttt{scop}$ by Eq. (14.112). Also, define the corresponding "standard" operation $\texttt{cop}$ (in the codensity monad) via Statement 14.9.4.2:

$$\text{cop} : F^{\text{Cod}^{M,A}} \to \text{Cod}^{M,A} \quad , \qquad \text{cop} \stackrel{\text{def}}{=} \text{scop} \fatsemi \text{ftn}_{\text{Cod}} \quad .$$

$$\begin{tikzcd}
F^{M^A} \arrow[r, "\text{inC}^{\uparrow F}"'] \arrow[d, "\text{op}"'] & F^{\text{Cod}^{M,A}} \arrow[d, "\text{cop}"] \\
M^A & \text{Cod}^{M,A} \arrow[l, "\text{outC}"]
\end{tikzcd}$$

Then $\texttt{op}$ may be expressed through $\texttt{cop}$ by the following code:

$$\text{op} = \text{inC}^{\uparrow F} \fatsemi \text{cop} \fatsemi \text{outC} \quad . \tag{14.113}$$

Proof In the derivation, we will need the code of $^{\uparrow \text{Cod}}$ and ftn_{Cod}:

$$\begin{aligned}
&(f^{:A \to B})^{\uparrow \text{Cod}} = c^{:\text{Cod}^{M,A}} \to \forall R.\, k^{:B \to M^R} \to c(f \fatsemi k) \quad , \\
&c^{:\text{Cod}^{M,\text{Cod}^{M,A}}} \triangleright \text{ftn}_{\text{Cod}} = k^{:A \to M^R} \to c\left(d^{:\text{Cod}^{M,A}} \to d(k)\right) \quad .
\end{aligned}$$

We rewrite the right-hand side of Eq. (14.113) and use the definitions of $\texttt{cop}$ and $\texttt{scop}$:

$$\begin{aligned}
\text{expect to equal op}: \quad &\text{inC}^{\uparrow F} \fatsemi \underline{\text{cop}} \fatsemi \text{outC} = \underline{\text{inC}^{\uparrow F} \fatsemi \text{scop}} \fatsemi \text{ftn}_{\text{Cod}} \fatsemi \text{outC} \\
\text{naturality law of scop}: \quad &= \text{scop} \fatsemi \text{inC}^{\uparrow \text{Cod}} \fatsemi \text{ftn}_{\text{Cod}} \fatsemi \text{outC} \\
\text{use Eq. (14.112) and } ^{\uparrow \text{Cod}}: \quad &= (f \to k \to f \triangleright k^{\uparrow F} \triangleright \text{op}) \fatsemi (c \to k \to c(\text{inC} \fatsemi k)) \fatsemi \text{ftn}_{\text{Cod}} \fatsemi \text{outC} \\
\text{compute composition}: \quad &= f \to \left(k \to f \triangleright \underline{(\text{inC} \fatsemi k)^{\uparrow F}} \triangleright \text{op}\right) \triangleright \text{ftn}_{\text{Cod}} \triangleright \text{outC} \\
\text{definition of ftn}_{\text{Cod}}: \quad &= f \to \left(k \to f \triangleright \text{inC}^{\uparrow F} \fatsemi (d \to d(k))^{\uparrow F} \fatsemi \text{op}\right) \triangleright \text{outC} \\
\text{definition of outC}: \quad &= f \to f \triangleright \text{inC}^{\uparrow F} \fatsemi \underline{(d \to d(\text{pu}_M))}^{\uparrow F} \fatsemi \text{op} \\
\text{definition of outC}: \quad &= f \to f \triangleright \text{inC}^{\uparrow F} \fatsemi \text{outC}^{\uparrow F} \fatsemi \text{op} = f \to f \triangleright \text{op} = \text{op} \quad .
\end{aligned}$$

□

We may convert $\text{scop}_{P'}$ into the corresponding "standard" operation ($\text{cop}_{P'}$):

$$\text{cop}_{P'} : F^{P'^A} \to P'^A \quad , \quad \text{cop}_{P'} \stackrel{\text{def}}{=} \text{scop}_{P'} \fatsemi \text{ftn}_{P'} \quad .$$

To convert $\text{cop}_{P'}$ into op_P, we note that Eq. (14.113) expresses $\texttt{op}$ via $\texttt{cop}$ using the functions $\texttt{inC}$ and $\texttt{outC}$, which transform between the monads M and Cod^M. Lifting these functions with frun_L, we can express the operation op_P via $\text{cop}_{P'}$ like this:

$$\text{op}_P = (\text{frun}_L(\text{inC}))^{\uparrow F} ⨾ \text{cop}_{P'} ⨾ \text{frun}_L(\text{outC}) \quad . \tag{14.114}$$

$$\begin{tikzcd} F^{P^A} \arrow[r, "(\text{frun}_L(\text{inC}))^{\uparrow F}"] \arrow[d, "\text{op}_P"'] & F^{P'^A} \arrow[d, "\text{cop}_{P'}"] \\ P^A & P'^A \arrow[l, "\text{frun}_L(\text{outC})"'] \end{tikzcd}$$

The following statement shows that the resulting operation (op_P) is a correct lifting of `op`.

Statement 14.9.4.8 The function op_P defined by Eq. (14.114) is a monadically natural lifting of `op` if the monad transformer T_L has a `frun` method and if the monadic naturality law of `flift` holds for `outC` (even though `outC` is not a monad morphism).[24]

Proof The lifting of M to P is given by flift_L. To verify the monadic naturality law of $\text{op}_{P'}$, we will have to use the monad morphism composition law of flift_L:

$$\text{flift}_L^{\uparrow \text{Cod}} ⨾ \text{flift}_L ⨾ \text{ftn}_{P'} = \text{ftn}_{\text{Cod}} ⨾ \text{flift}_L \quad .$$

To be able to use that law, we need to move the two functions flift_L towards $\text{ftn}_{P'}$. Compute:

$$\begin{aligned}
\text{expect to equal } (\text{op} ⨾ \text{flift}_L): \quad & \text{flift}_L^{\uparrow F} ⨾ \text{op}_P = \text{flift}_L^{\uparrow F} ⨾ (\text{frun}_L(\text{inC}))^{\uparrow F} ⨾ \text{cop}_{P'} ⨾ \text{frun}_L(\text{outC}) \\
& = \underline{(\text{flift}_L ⨾ \text{frun}_L(\text{inC}))}^{\uparrow F} ⨾ \text{scop}_{P'} ⨾ \text{ftn}_{P'} ⨾ \text{frun}_L(\text{outC}) \\
\text{monadic naturality law of } \text{flift}_L: \quad & = (\text{inC} ⨾ \text{flift}_L)^{\uparrow F} ⨾ \underline{\text{scop}} ⨾ \text{flift}_L ⨾ \text{ftn}_{P'} ⨾ \text{frun}_L(\text{outC}) \\
\text{naturality law of scop}: \quad & = \text{scop} ⨾ \text{inC}^{\uparrow \text{Cod}} ⨾ \underline{\text{flift}_L^{\uparrow \text{Cod}} ⨾ \text{flift}_L ⨾ \text{ftn}_{P'}} ⨾ \text{frun}_L(\text{outC}) \\
\text{composition law of } \text{flift}_L: \quad & = \text{scop} ⨾ \text{inC}^{\uparrow \text{Cod}} ⨾ \text{ftn}_{\text{Cod}} ⨾ \underline{\text{flift}_L ⨾ \text{frun}_L(\text{outC})} \\
\text{monadic naturality law of } \text{flift}_L: \quad & = \text{scop} ⨾ \text{inC}^{\uparrow \text{Cod}} ⨾ \text{ftn}_{\text{Cod}} ⨾ \text{outC} ⨾ \text{flift}_L \quad .
\end{aligned}$$

In the last line, we used the monadic naturality law of flift_L applied to `outC`, which holds by assumption. We now reuse the last derivation in the proof of Statement 14.9.4.7, where we found that:

$$\text{scop} ⨾ \text{inC}^{\uparrow \text{Cod}} ⨾ \text{ftn}_{\text{Cod}} ⨾ \text{outC} = \text{op} \quad .$$

This completes the proof of the equation $\text{flift}_L^{\uparrow F} ⨾ \text{op}_P = \text{op} ⨾ \text{flift}_L$. □

What if we use this method to lift a "standard" operation that is actually equivalent to a "simple" one and could be lifted more easily by composing with a monad morphism $p : M \rightsquigarrow P$? It turns out that the lifted operations will be the same:

Statement 14.9.4.9 Assume the conditions of Statement 14.9.4.8 and additionally assume that the "standard" operation `op` is equivalent to a "simple" operation `sop` (so that $\text{op} = \text{sop} ⨾ \text{ftn}_M$). Then the lifted "standard" operation (op_P) is equivalent to the lifted "simple" operation ($\text{sop}_P \stackrel{\text{def}}{=} \text{sop} ⨾ \text{flift}_L$) according to the usual formula $\text{op}_P = \text{sop}_P ⨾ \text{ftn}_P$.

Proof Begin by rewriting Eq. (14.114):

$$\begin{aligned}
\text{expect to equal } (\text{sop}_P ⨾ \text{ftn}_P): \quad & \text{op}_P = (\text{frun}_L(\text{inC}))^{\uparrow F} ⨾ \underline{\text{cop}_{P'}} ⨾ \text{frun}_L(\text{outC}) \\
\text{definition of } \text{cop}_{P'}: \quad & = \underline{(\text{frun}_L(\text{inC}))^{\uparrow F} ⨾ \text{scop}_{P'}} ⨾ \text{ftn}_{P'} ⨾ \text{frun}_L(\text{outC}) \\
\text{naturality law of } \text{scop}_{P'}: \quad & = \underline{\text{scop}_{P'}} ⨾ (\text{frun}_L(\text{inC}))^{\uparrow P'} ⨾ \text{ftn}_{P'} ⨾ \text{frun}_L(\text{outC}) \\
\text{definition of } \text{scop}_{P'}: \quad & = \text{scop} ⨾ \text{flift}_L^{\text{Cod}} ⨾ (\text{frun}_L(\text{inC}))^{\uparrow P'} ⨾ \text{ftn}_{P'} ⨾ \text{frun}_L(\text{outC}) \quad .
\end{aligned}$$

At this point, we substitute $\text{op} = \text{sop} ⨾ \text{ftn}_M$ into Eq. (14.112) and rewrite `scop` in terms of `inC`:

$$\begin{aligned}
& \text{scop} \stackrel{\text{def}}{=} f^{:F^A} \to k^{:A \to M^R} \to f \triangleright k^{\uparrow F} \triangleright \text{op} = f \to k \to f \triangleright \underline{k^{\uparrow F} ⨾ \text{sop}} ⨾ \text{ftn}_M \\
\text{naturality law of sop}: \quad & = f \to k \to f \triangleright \text{sop} ⨾ \underline{k^{\uparrow M} ⨾ \text{ftn}_M} = f \to k \to f \triangleright \text{sop} ⨾ \text{flm}_M(k) \\
\text{move sop outside}: \quad & = (f \to f \triangleright \text{sop}) ⨾ (\underline{f \to k \to f \triangleright \text{flm}_M(k)}) = \text{sop} ⨾ \text{inC} \quad .
\end{aligned}$$

[24] This version of the monadic naturality law holds for all monad transformers that support `frun`, as this chapter proved.

Now we continue the derivation:

$$\begin{aligned}
\text{substitute scop} = \text{sop} \fatsemi \text{inC} : \quad &= \text{sop} \fatsemi \underline{\text{inC} \fatsemi \text{flift}_L^{\text{Cod}}} \fatsemi (\text{frun}_L(\text{inC}))^{\uparrow P'} \fatsemi \text{ftn}_{P'} \fatsemi \text{frun}_L(\text{outC}) \\
\text{monadic naturality law of flift}_L : \quad &= \text{sop} \fatsemi \text{flift}_L \fatsemi \underline{\text{frun}_L(\text{inC}) \fatsemi (\text{frun}_L(\text{inC}))^{\uparrow P'} \fatsemi \text{ftn}_{P'}} \fatsemi \text{frun}_L(\text{outC}) \\
\text{composition law of frun}_L(\text{inC}) : \quad &= \text{sop} \fatsemi \text{flift}_L \fatsemi \text{ftn}_P \fatsemi \underline{\text{frun}_L(\text{inC}) \fatsemi \text{frun}_L(\text{outC})} \\
\text{composition law of frun}_L : \quad &= \text{sop} \fatsemi \text{flift}_L \fatsemi \text{ftn}_P \fatsemi \text{frun}_L(\underline{\text{inC} \fatsemi \text{outC}}) \\
\text{use inC} \fatsemi \text{outC} = \text{id} : \quad &= \underline{\text{sop} \fatsemi \text{flift}_L} \fatsemi \text{ftn}_P = \text{sop}_P \fatsemi \text{ftn}_P \quad .
\end{aligned}$$

□

Lifting to the base monad The method just shown lifts "standard" M-operations to monad stacks $L \lhd M$ where the monad M is located at the deep end. If M is located somewhere else in the stack, there exist monads L and N such that the stack P is expressed as $P = L \lhd M \lhd N$. If we knew how to lift M-operations to the stack $M \lhd N$ (where M is the base monad), we will then be able to lift them also to P since the monad $M \lhd N$ is located at the deep end of P. So, it remains to see how M-operations may be lifted to a stack $M \lhd N$ that has M as its base monad.

All "standard" M-operations that satisfy the compatibility law (14.108) are equivalent to "simple" operations and are lifted to the stack $Q \stackrel{\text{def}}{=} M \lhd N$ using Q's `blift` function. (For monads whose transformers do not support base lifts, operations must be lifted by custom code.) It remains to consider "standard" operations $\text{op} : F \circ M \rightsquigarrow M$ that do *not* satisfy the compatibility law. Our goal is to obtain a lifted operation $\text{op}_Q : F \circ Q \rightsquigarrow Q$ out of a given non-"simple" operation `op`.

If M's transformer is composed-inside (so that $Q^A \stackrel{\text{def}}{=} M^{N^A}$), defining op_Q is straightforward:

$$\text{op}_Q : F^{M^{N^A}} \to M^{N^A} \quad , \qquad \text{op}_Q \stackrel{\text{def}}{=} \text{op}^{N^A} \quad .$$

To show that op_Q is a monadically natural lifting of `op`, compute:

$$\begin{aligned}
\text{expect to equal } (\text{op} \fatsemi \text{blift}_M) : \quad &\text{blift}_M^{\uparrow F} \fatsemi \text{op}_Q = \text{pu}_N^{\uparrow M \uparrow F} \fatsemi \text{op}_Q \\
\text{naturality law of op}_Q : \quad &= \text{op}_Q \fatsemi \text{pu}_N^{\uparrow M} = \text{op} \fatsemi \text{blift}_M \quad .
\end{aligned}$$

This case covers rigid monads (such as `Reader` with its non-"simple" `local` operation) but does not apply, e.g., to the `Writer` monad's `clear` operation or to other non-rigid monads. The `Writer` monad's transformer is composed-outside, so $M \lhd N = N \circ M$. We could implement a function of type $F^{N^{M^A}} \to N^{M^A}$ given a function $\text{op} : F^{M^A} \to M^A$, but that function will not be a monadically natural lifting of `op`:

Statement 14.9.4.10 Assume that M is a monad with a composed-outside transformer, N is another monad, and $\text{op} : F^{M^A} \to M^A$ is a given M-operation that does *not* satisfy the compatibility law (14.108). Then the function op_Q defined by:

$$\text{op}_Q : F^{N^{M^A}} \to N^{M^A} \quad , \qquad \text{op}_Q \stackrel{\text{def}}{=} \text{pu}_M^{\uparrow F} \fatsemi \text{op} \fatsemi \text{pu}_N \fatsemi \text{ftn}_{N \circ M}$$

is *not* a monadically natural lifting of `op` to the stack $Q \stackrel{\text{def}}{=} M \lhd N \cong N \circ M$.

Proof Using the definition $\text{blift}_M \stackrel{\text{def}}{=} \text{pu}_N$, we write the two sides of the monadic naturality law:

$$\begin{aligned}
\text{left-hand side} : \quad &\text{op} \fatsemi \text{blift}_M = \text{op} \fatsemi \text{pu}_N \quad , \\
\text{right-hand side} : \quad &\text{blift}_M^{\uparrow F} \fatsemi \text{op}_Q = \underline{\text{pu}_N^{\uparrow F} \fatsemi \text{pu}_M^{\uparrow F}} \fatsemi \text{op} \fatsemi \text{pu}_N \fatsemi \text{ftn}_{N \circ M} \\
\text{naturality law of pu}_M : \quad &= \text{pu}_M^{\uparrow F} \fatsemi \underline{\text{pu}_N^{\uparrow M \uparrow F} \fatsemi \text{op}} \fatsemi \text{pu}_N \fatsemi \text{ftn}_{N \circ M} \\
\text{naturality law of op} : \quad &= \text{pu}_M^{\uparrow F} \fatsemi \text{op} \fatsemi \underline{\text{pu}_N^{\uparrow M} \fatsemi \text{pu}_N} \fatsemi \text{ftn}_{N \circ M} = \text{pu}_M^{\uparrow F} \fatsemi \text{op} \fatsemi \underline{\text{blift}_M^{\uparrow M} \fatsemi \text{blift}_M} \fatsemi \text{ftn}_{N \circ M} \\
\text{composition law of blift} : \quad &= \text{pu}_M^{\uparrow F} \fatsemi \text{op} \fatsemi \text{ftn}_M \fatsemi \underline{\text{blift}_M} = \text{pu}_M^{\uparrow F} \fatsemi \text{op} \fatsemi \text{ftn}_M \fatsemi \text{pu}_N \quad .
\end{aligned}$$

The two sides would be equal if:

$$\text{op} ⨾ \text{pu}_N \overset{?}{=} \text{pu}_M^{\uparrow F} ⨾ \text{op} ⨾ \text{ftn}_M ⨾ \text{pu}_N \quad .$$

This would be true if `op` satisfied the compatibility law (14.108), because then we would have:

$$\text{pu}_M^{\uparrow F} ⨾ \text{op} ⨾ \text{ftn}_M = \underline{\text{pu}_M^{\uparrow F} ⨾ \text{ftn}_M^{\uparrow F}} ⨾ \text{op} = \text{op} \quad .$$

However, by assumption the law (14.108) does not hold for `op`. To see that this is sufficient to destroy the monadic naturality of op_Q, choose $F \overset{\text{def}}{=} \text{Id}$, $N \overset{\text{def}}{=} \text{Id}$, $M^A \overset{\text{def}}{=} A \times W$ (a `Writer` monad), and `op=clear`:

$$\text{clear} : M^A \to M^A \quad , \qquad \text{clear} \overset{\text{def}}{=} a^{:A} \times _^{:W} \to a \times e_W \quad .$$

Then $\text{pu}_N = \text{id}$, and we compute $\text{pu}_M^{\uparrow F} ⨾ \text{clear} ⨾ \text{ftn}_M ⨾ \text{pu}_N$ as a function of type $A \times W \to A \times W$:

$$\begin{aligned} &\text{pu}_M^{\uparrow F} ⨾ \text{clear} ⨾ \text{ftn}_M ⨾ \text{pu}_N = (a \times w \to a \times w \times e_W) ⨾ (b^{:A\times W} \times _^{:W} \to b \times e_W) ⨾ \text{ftn}_M \\ &= a \times w \to \text{ftn}_M(a \times w \times e_W) = a \times w \to a \times w = \text{id} \quad . \end{aligned}$$

This is not equal to `clear` because `clear` is not an identity function. □

As the case of the `Writer` monad shows, it does not seem possible to implement a monadically natural lifting to the base monad for all monads M at once. The method of Statements 14.9.4.7–14.9.4.8 cannot be used because the codensity monad whose transformer does not support base lifts. So, lifting those operations requires custom code adapted to the particular structure of the functor F and the base monad M. For example, we may lift the `Writer` monad's `clear` operation to any stack $Q \overset{\text{def}}{=} M ◁ N = N \circ M$ like this:

Example 14.9.4.11 For any monad N, define $Q \overset{\text{def}}{=} \text{Writer}^{W,\bullet} ◁ N$, so $Q^A \overset{\text{def}}{=} N^{A\times W}$. A monadically natural lifting of the `Writer` monad's `clear` operation to Q is then defined by:

$$\text{clear}_Q : N^{A\times W} \to N^{A\times W} \quad , \qquad \text{clear}_Q \overset{\text{def}}{=} \text{clear}^{\uparrow N} \quad .$$

To verify the monadic naturality law using $\text{blift}_{\text{Writer}} \overset{\text{def}}{=} \text{pu}_N$ and $F = \text{Id}$:

$$\begin{aligned} \text{expect to equal } (\text{clear} ⨾ \text{pu}_N) : \quad & \text{pu}_N^{\uparrow F} ⨾ \text{clear}_Q = \text{pu}_N ⨾ \text{clear}^{\uparrow N} \\ \text{naturality law of } \text{pu}_N : \quad & = \text{clear} ⨾ \text{pu}_N \quad . \end{aligned}$$

□

A similar case is the `Either` monad's `handle` operation:

$$\text{handle} : (E \to E + A) \times (E + A) \to E + A \quad , \qquad \text{handle} \overset{\text{def}}{=} h^{:E\to E+A} \times m^{:E+A} \to m ▷ \left\| \begin{array}{c|c} & E + A \\ \hline E & h \\ A & a \to \mathbb{0} + a \end{array} \right\| \quad .$$

This operation is of the "standard" form with the functor $F^B \overset{\text{def}}{=} (E \to B) \times B$ but is not equivalent to a "simple" operation. Custom code is needed to lift the handle operation to a stack $Q \overset{\text{def}}{=} \text{Either}^{E,\bullet} ◁ N$:

Example 14.9.4.12 For any monad N, define $Q \overset{\text{def}}{=} \text{Either}^{E,\bullet} ◁ N$, so $Q^A \overset{\text{def}}{=} N^{E+A}$ and $\text{pu}_Q(a) = \text{pu}_N(\mathbb{0} + a)$. Then a monadically natural lifting of the `Either` monad's `handle` operation to Q is:

$$\text{handle}_Q : (E \to Q^A) \times Q^A \to Q^A \quad , \qquad \text{handle}_Q \overset{\text{def}}{=} h^{:E\to N^{E+A}} \times n^{:N^{E+A}} \to n ▷ \text{flm}_N \left(\left\| \begin{array}{c|c} & N^{E+A} \\ \hline E & h \\ A & \text{pu}_Q \end{array} \right\| \right) \quad .$$

To verify the monadic naturality law using $\text{blift}_{\text{Either}} \stackrel{\text{def}}{=} \text{pu}_N$ and $F^B \stackrel{\text{def}}{=} (E \to B) \times B$:

$$\text{expect}\,(\text{handle} \,⨾\, \text{pu}_N): \quad \text{pu}_N^{\uparrow F} \,⨾\, \text{handle}_Q = \left(h^{:E\to E+A} \times m^{:E+A} \to (h \,⨾\, \text{pu}_N) \times (m \triangleright \text{pu}_N)\right) ⨾ \text{handle}_Q$$

$$= h \times m \to m \triangleright \underline{\text{pu}_N \triangleright \text{flm}_N}\left(\begin{array}{c||c} & N^{E+A} \\ \hline E & h \,⨾\, \text{pu}_N \\ A & \text{pu}_Q \end{array}\right) = h \times m \to m \triangleright \begin{array}{c||c} & N^{E+A} \\ \hline E & h \,⨾\, \text{pu}_N \\ A & \text{pu}_Q \end{array}$$

$$= h \times m \to m \triangleright \begin{array}{c||c} & N^{E+A} \\ \hline E & h \\ A & a \to \mathbb{0} + a \end{array} ⨾ \text{pu}_N = \text{handle} ⨾ \text{pu}_N \quad .$$

□

After writing custom code for lifting each "standard" but non-"simple" operation to the base monad of a stack, we will be able to lift those operations to arbitrary monad stacks. The length of the required custom code is linear in the number of such operations.

We could also combine all "standard" operations ($\text{op}_1 : F_1^{M^A} \to M^A$, $\text{op}_2 : F_2^{M^A} \to M^A$, etc.) into a single operation by using the type equivalence:

$$(F_1^{M^A} \to M^A) \times (F_2^{M^A} \to M^A) \cong (F_1 + F_2)^{M^A} \to M^A \quad .$$

The resulting "standard" operation corresponds to the product of all "standard" operations of the monad M. It is easier to reason about operations when all operations are combined into one. Having lifted the product of all operations, we obtain the product of lifted operations. However, this will not reduce the length of the required custom lifting code. Practical programming is easier when individual operations are kept separate.

"Flipped-standard" operations It remains to lift operations that have neither a "fixed-type", nor "simple", nor a "standard" type signature. Four such operations are shown in Table 14.4, namely `filter`, `collect`, `process`, and `callcc`. As we know from Chapter 9, the `filter` function is equivalent to `deflate` with type signature $\text{List}^{\mathbb{1}+A} \to \text{List}^A$. Then the type signatures of `deflate` and `process` are of the "**flipped-standard**" form $M^{F^A} \to M^A$, where F is some functor. The `collect` operation can be also transformed to a "flipped-standard" form:

Example 14.9.4.13 Show that the type of `Writer` monad's `collect` operation,

$$\text{collect} : \forall A.\, A \times W \to (A \times W) \times W \quad ,$$

is equivalent to the type $\forall A.\, (W \to A) \times W \to A \times W$.

Solution We use Statement 14.9.4.6 with the functors F, K, L defined as $F^A \stackrel{\text{def}}{=} A \times W$, $K^A \stackrel{\text{def}}{=} W \to A$, and $L^A \stackrel{\text{def}}{=} A \times W$. The result is the type equivalence between:

$$\forall A.\, F^{K^A} \to L^A = \forall A.\, (W \to A) \times W \to A \times W$$

and:

$$\forall A.\, F^A \to \forall R.\, (A \to K^R) \to L^R = \forall A.\, A \times W \to \forall R.\, (A \to W \to R) \to R \times W \quad .$$

Transforming the last type via the covariant Yoneda identity, we obtain the type of `collect`:

$$\forall A.\, A \times W \to \forall R.\, (\underline{A \to W \to} R) \to R \times W \cong \forall A.\, A \times W \to \underline{\forall R.}\, (A \times W \to \underline{R) \to R} \times W$$

$$\cong \forall A.\, A \times W \to (A \times W) \times W \quad .$$

□

It follows that the `collect` operation may be expressed equivalently through an operation with the type signature $\text{Writer}^{W, W\to A} \to \text{Writer}^{W,A}$. That type is of "flipped-standard" form, $M^{F^A} \to M^A$, where $M^A = \text{Writer}^{W,A}$ and $F^A \stackrel{\text{def}}{=} W \to A$.

If an operation of "flipped-standard" form ($M^{F^A} \to M^A$) satisfies a suitable compatibility law, it is equivalent to a "simple" operation ($F^A \to M^A$):

Statement 14.9.4.14 The type of functions $\text{op} : M^{F^A} \to M^A$ that obey the compatibility law:

$$\begin{tikzcd} M^{M^{F^A}} \arrow[r, "\text{op}^{\uparrow M}"'] \arrow[d, "\text{ftn}_M"'] & M^{M^A} \arrow[d, "\text{ftn}_M"] \\ M^{P^A} \arrow[r, "\text{op}"] & M^A \end{tikzcd}$$

$$\text{op}^{\uparrow M} \fatsemi \text{ftn}_M = \text{ftn}_M \fatsemi \text{op} \quad ,$$

is equivalent to the type of functions $F^A \to M^A$. (Here M is a monad and F is any functor.)

Proof Given a function $\text{op} : M^{F^A} \to M^A$, define the corresponding operation $\text{sop} : F^A \to M^A$ as:

$$\text{sop} \stackrel{\text{def}}{=} \text{pu}_M \fatsemi \text{op} \quad .$$

Given a function $\text{sop} : F^A \to M^A$, define the corresponding operation:

$$\text{op} \stackrel{\text{def}}{=} \text{sop}^{\uparrow M} \fatsemi \text{ftn}_M = \text{flm}_M(\text{sop}) \quad .$$

To verify one direction of the isomorphism, substitute `sop` into `op`:

$$\begin{aligned} \text{expect to equal op}: \quad & \text{sop}^{\uparrow M} \fatsemi \text{ftn}_M = (\text{pu}_M \fatsemi \underline{\text{op})^{\uparrow M} \fatsemi \text{ftn}_M} \\ \text{compatibility law of op}: \quad & = \underline{\text{pu}_M^{\uparrow M} \fatsemi \text{ftn}_M} \fatsemi \text{op} = \text{op} \quad . \end{aligned}$$

To verify the reverse direction of the isomorphism, substitute `op` into `sop`:

$$\begin{aligned} \text{expect to equal sop}: \quad & \text{pu}_M \fatsemi \text{op} = \text{pu}_M \fatsemi \text{sop}^{\uparrow M} \fatsemi \text{ftn}_M \\ \text{naturality law of pu}_M: \quad & = \text{sop} \fatsemi \underline{\text{pu}_M \fatsemi \text{ftn}_M} = \text{sop} \quad . \end{aligned}$$

If an operation `op` is defined via `sop`, the compatibility law of `op` will hold automatically:

$$\begin{aligned} \text{expect to equal } (\text{ftn}_M \fatsemi \text{op}): \quad & \text{op}^{\uparrow M} \fatsemi \text{ftn}_M = (\text{sop}^{\uparrow M} \fatsemi \underline{\text{ftn}_M)^{\uparrow M} \fatsemi \text{ftn}_M} \\ \text{associativity law of ftn}_M: \quad & = \underline{\text{sop}^{\uparrow M \uparrow M} \fatsemi \text{ftn}_M} \fatsemi \text{ftn}_M = \text{ftn}_M \fatsemi \text{sop}^{\uparrow M} \fatsemi \text{ftn}_M = \text{ftn}_M \fatsemi \text{op} \quad . \end{aligned}$$

□

The "flipped-standard" operations that are equivalent to "simple" operations may be lifted to monad stacks similarly to the "standard" operations that are equivalent to "simple" operations. An example of such an operation is `deflate`: if the monad M is at the same time a filterable functor, it is reasonable to require the **compatibility law**:

$$\text{deflate}^{\uparrow M} \fatsemi \text{ftn}_M = \text{ftn}_M \fatsemi \text{deflate} \quad .$$

This law holds for the `List` monad (filtering before flattening or after flattening gives the same results). One can show that `deflate` is equivalent to a "simple" operation that we will call `defl`:

$$\text{defl} : \mathbb{1} + A \to \text{List}^A \quad , \qquad \text{defl} \stackrel{\text{def}}{=} \begin{array}{|c||cc|} & \mathbb{1} & A \times \text{List}^A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & a \to a \times (\mathbb{1} + \mathbb{0}) \end{array} \quad , \qquad \text{deflate} = \text{flm}_{\text{List}}(\text{defl}) \quad .$$

To lift `deflate` to a stack P, we first lift `defl` to P by composing it with a monad morphism $p : M \rightsquigarrow P$ and then pass to the corresponding `deflate` operation, defined by:

$$\text{deflate}_P \stackrel{\text{def}}{=} \text{flm}_P(\text{defl}_P) = \text{flm}_P(\text{defl} \fatsemi p) \quad .$$

However, the other two operations of "flipped-standard" signature (`collect` and `process`) are *not* equivalent to "simple" operations. Let us see how to lift those operations to an arbitrary monad stack containing the monad M.

We need to consider two cases: the stack P contains M at the deep end ($P \stackrel{\text{def}}{=} L \lhd M$) or in the middle ($P \stackrel{\text{def}}{=} L \lhd M \lhd N$).

In the first case, we assume that the monad L has the foreign runner (frun_L). For a given operation $\text{op} : M^{F^A} \to M^A$, we define the lifted operation (op_P) by:

$$\text{op}_P : (L \lhd M)^{F^A} \to (L \lhd M)^A \quad , \qquad \text{op}_P \stackrel{\text{def}}{=} \text{frun}_L(\text{op}) \quad .$$

To verify the monadic naturality law of op_P written as:

$$\text{flift}_L^{F^A} \fatsemi \text{op}_P \stackrel{?}{=} \text{op} \fatsemi \text{flift}_L \quad ,$$

we use the monadic naturality law of flift_L applied to the natural transformation $\texttt{op}$ (which is not a monad morphism):

$$\text{flift}_L^{F^A} \fatsemi \text{op}_P = \text{flift}_L \fatsemi \text{frun}_L(\text{op}) = \text{op} \fatsemi \text{flift}_L \quad .$$

In the second case, we need to lift $\texttt{op}$ to the stack $Q \stackrel{\text{def}}{=} M \lhd N$ that contains M as a base monad. In general (for arbitrary "flipped-standard" operations), this lifting is not possible since we cannot transform $(M \lhd N)^{F^A}$ to M^{F^A}. However, the operations `collect` and `process` are in the `Writer` monad whose transformer is composed-outside ($M \lhd N = N \circ M$). In that case, the lifted operation (op_Q) can be defined by:

$$\text{op}_Q : N^{M^{F^A}} \to N^{M^A} \quad , \qquad \text{op}_Q \stackrel{\text{def}}{=} \text{op}^{\uparrow N} \quad .$$

To verify the monadic naturality law of op_Q, we use the definition $\text{blift}_M \stackrel{\text{def}}{=} \text{pu}_N$:

$$\text{expect to equal } (\text{op} \fatsemi \text{blift}_M) : \quad \text{blift}_M \fatsemi \text{op}_Q = \text{pu}_N \fatsemi \text{op}^{\uparrow N} = \text{op} \fatsemi \text{pu}_N = \text{op} \fatsemi \text{blift}_M \quad .$$

The `callcc` operation has a type signature of neither "standard" nor "flipped-standard" form:

$$\text{callcc} : \forall (A, B).\, ((A \to \text{Cont}^{R,B}) \to \text{Cont}^{R,A}) \to \text{Cont}^{R,A} \quad .$$

It turns out that, using some tricks, one can express `callcc` through two "simple" operations: one denoted by `jump` (with type signature $\text{jump} : \forall A.\, R \to \text{Cont}^{R,A}$) and another denoted by `scc` (with type signature $\text{scc} : \forall A.\, F^A \to \text{Cont}^{R,A}$). The functor F for operation `scc` is the "selector monad": $F^A \stackrel{\text{def}}{=} \text{Sel}^{R,A} = (A \to R) \to A$. The function `scc` is studied in Exercise 14.9.5.13. The two "simple" operations `scc` and `jump` are lifted straightforwardly to any monad stack.

To express `callcc` via `jump` and `scc`, we first convert `scc` to an equivalent "standard" operation (`cc`):

$$\text{cc} : \text{Sel}^{R,\text{Cont}^{R,A}} \to \text{Cont}^{R,A} \quad , \qquad \text{cc} \stackrel{\text{def}}{=} \text{scc} \fatsemi \text{ftn}_{\text{Cont}} \quad .$$

To arrive at a systematic way of transforming functions of `cc`'s type signature into functions of `callcc`'s type signature, let us write out some more detail in the type signatures of `callcc` and `cc`:

$$\begin{aligned} &\text{callcc} : \forall (A, B).\, \big(\big((A \to (B \to R) \to R\big) \to \text{Cont}^{R,A}\big) \to \text{Cont}^{R,A} \quad , \\ &\text{cc} : \forall A.\, \big(\big(((A \to R) \to R) \to R\big) \to \text{Cont}^{R,A}\big) \to \text{Cont}^{R,A} \quad . \end{aligned}$$

The difference is $A \to (B \to R) \to R$ instead of $((A \to R) \to R) \to R$. Note that the type parameter B appears under the universal quantifier ($\forall B$) and is in a covariant position. So, we can set $B = \mathbb{0}$ in the type of `callcc` and obtain an equivalent type:

$$\underline{\forall B.}\, A \to (\underline{B} \to R) \to R \cong A \to (\underline{\mathbb{0} \to R}) \to R \cong A \to \underline{\mathbb{1} \to} R \cong A \to R \quad .$$

The type $A \to R$ can be seen as equivalent to a function of type $A \to (B \to R) \to R$ that ignores its curried argument of type $B \to R$. In this way, the type expression supports an arbitrary type parameter B.

The next trick is to notice that there is a mapping from the type $((A \to R) \to R) \to R$ to $A \to R$:

Statement 14.9.4.15 Assume that A and R are some fixed types. The type $((A \to R) \to R) \to R$ can be surjectively mapped to the type $A \to R$, which, in turn, can be injectively mapped back to $((A \to R) \to R) \to R$. These two types are *not* equivalent (at least for some A and R).

Proof The mappings are implemented by two functions that we will call `inS` and `outS`:

$$\text{inS} : (A \to R) \to ((A \to R) \to R) \to R \quad , \qquad \text{inS} \stackrel{\text{def}}{=} k^{:A\to R} \to c^{:(A\to R)\to R} \to c(k) \quad ,$$
$$\text{outS} : \big(((A \to R) \to R) \to R\big) \to A \to R \quad , \qquad \text{outS} \stackrel{\text{def}}{=} s^{:((A\to R)\to R)\to R} \to a^{:A} \to s(k^{:A\to R} \to k(a)) \quad .$$

To prove the injective and surjective properties, we will show that $\text{inS} ⨾ \text{outS} = \text{id}$:

$$\begin{aligned} \text{expect to equal } k : \quad & k^{:A\to R} ▷ \text{inS} ▷ \text{outS} = (c \to c(k)) ▷ \text{outS} = a \to (c \to c(k))(k \to k(a)) \\ \text{compute composition} : \quad & = a \to k(a) = k \quad . \end{aligned}$$

To show that the types $A \to R$ and $((A \to R) \to R) \to R$ are not equivalent, we consider $A = R = \mathbb{2}$ and find that these types have different cardinality. □

In the type signature of `cc`, the type $((A \to R) \to R) \to R$ appears wrapped under the functor $(\bullet \to \text{Cont}^{R,A}) \to \text{Cont}^{R,A}$(here the types A and R are considered as fixed). Denoting that functor by G, we can write a function transforming the type of `cc` into the type of `callcc` as:

$$G^X \stackrel{\text{def}}{=} (X \to \text{Cont}^{R,A}) \to \text{Cont}^{R,A} \quad , \qquad \text{outS}^{\uparrow G} : G^{((A\to R)\to R)\to R} \to G^{A\to R} \quad .$$

By Statement 6.4.2.1, $\text{outS}^{\uparrow G}$ is a surjective function. So, we will not lose information if we express `callcc` via `cc`. Let us compute and simplify the code of $\text{cc} ▷ \text{outS}^{\uparrow G}$:

$$\begin{aligned} & (f^{:X\to Y})^{\uparrow G} \stackrel{\text{def}}{=} g^{:(X\to \text{Cont}^{R,A})\to \text{Cont}^{R,A}} \to h^{:Y\to \text{Cont}^{R,A}} \to g(f ⨾ h) \quad , \\ & \text{cc} ▷ \text{outS}^{\uparrow G} = h^{:(A\to R)\to \text{Cont}^{R,A}} \to \text{cc}\,(\text{outS} ⨾ h) \\ & \quad = h^{:(A\to R)\to \text{Cont}^{R,A}} \to \text{cc}\,(s^{:((A\to R)\to R)\to R} \to h(a^{:A} \to s(k^{:A\to R} \to k(a)))) \quad . \end{aligned}$$

To obtain `callcc`, we still need to to replace $A \to R$ by $A \to (B \to R) \to R \cong A \to \text{Cont}^{R,B}$ in the type of h. We insert an ignored argument of type $B \to R$ when h is applied:

$$\text{callcc} = h^{:(A\to \text{Cont}^{R,B})\to \text{Cont}^{R,A}} \to \text{cc}\,\big(s^{:\text{Cont}^{R,A}\to R} \to h(a^{:A} \to _^{:B\to R} \to s(k^{:A\to R} \to k(a)))\big) \quad .$$

The last step is to obtain a formula for the operation `cc` after lifting to a monad stack P. We assume that a monad morphism $p : \text{Cont}^{R,\bullet} \rightsquigarrow P$ is given. Since `cc` is equivalent to a "simple" operation `scc`, the lifted operation (cc_P) is defined by:

$$\text{cc}_P \stackrel{\text{def}}{=} \text{scc} ⨾ p ⨾ \text{ftn}_P \quad .$$

It remains to express callcc_P via cc_P and the operations of the monad P. To achieve that, we cannot use the above formula for `callcc` directly because it uses the type of $\text{Cont}^{R,A}$. Instead, we rewrite that formula using the functions:

$$\text{pu}_{\text{Cont}}(a^{:A}) \stackrel{\text{def}}{=} k^{:A\to R} \to k(a) \quad , \qquad \text{jump}(r^{:R}) \stackrel{\text{def}}{=} _^{:X\to R} \to r \quad .$$

The result is:

$$\begin{aligned} & \text{callcc} = h^{:(A\to \text{Cont}^{R,B})\to \text{Cont}^{R,A}} \to \text{cc}\,\big(s^{:\text{Cont}^{R,A}\to R} \to h(a^{:A} \to \underline{_^{:B\to R} \to} s(\text{pu}_{\text{Cont}}(a)))\big) \\ & = h^{:(A\to \text{Cont}^{R,B})\to \text{Cont}^{R,A}} \to \text{cc}\,\big(s^{:\text{Cont}^{R,A}\to R} \to h(\underline{a^{:A} \to \text{jump}(s(\text{pu}_{\text{Cont}}(a))))}\big) \\ & = h^{:(A\to \text{Cont}^{R,B})\to \text{Cont}^{R,A}} \to \text{cc}\,\big(s^{:\text{Cont}^{R,A}\to R} \to h(\text{pu}_{\text{Cont}} ⨾ s ⨾ \text{jump})\big) \quad . \end{aligned}$$

We may lift the "simple" operation `jump` to the stack P as:

$$\text{jump}_P \stackrel{\text{def}}{=} \text{jump} ⨾ p \quad .$$

Finally, we define the lifted operation callcc_P through cc_P, jump_P, and pu_P:

$$\text{callcc}_P \stackrel{\text{def}}{=} h^{:(A\to P^B)\to P^A} \to \text{cc}_P\left(s^{:P^{A\to R}} \to h(\text{pu}_P \fatsemi s \fatsemi \text{jump}_P)\right) \quad .$$

In this way, we correctly reproduce the behavior of the `callcc` operation within the stack P.

Apart from `callcc`, the continuation monad has other operations such as `reset` and `shift`:[25]

$$\text{reset} : \forall S. \text{Cont}^{R,R} \to \text{Cont}^{S,R} \quad , \qquad \text{reset} \stackrel{\text{def}}{=} c^{:(R\to R)\to R} \to k^{:R\to S} \to k(c(\text{id})) \quad ,$$

$$\text{shift} : \forall A. ((A \to R) \to \text{Cont}^{R,R}) \to \text{Cont}^{R,A} \quad , \qquad \text{shift} \stackrel{\text{def}}{=} g^{:(A\to R)\to \text{Cont}^{R,R}} \to k^{:A\to R} \to g(k)(\text{id}) \quad .$$

The type signatures of these operations are neither "standard" nor of "flipped-standard". Lifting these operations in a systematic way to arbitrary monad stacks is the topic of Problem 15.3.2.12.

Characterizing monads by operations Each monad has a particular set of effectful operations. Can we recover a monad's type given its operations? This appears to be impossible in general. It is also unknown how to obtain a "complete" set of operations for a given monad. The following examples show some failing attempts to characterize monads by operations in the general case.

Example 14.9.4.16 Show that any monad M admits a "standard" operation with $F = M$. Is that "standard" operation equivalent to a "simple" operation?

Solution The type signature of a "standard" operation with $F = M$ is $\text{op} : M^{M^A} \to M^A$. For an arbitrary monad M, the only available function with that type signature is $\text{op} \stackrel{\text{def}}{=} \text{ftn}_M$. The compabitility law holds for this function due to the associativity law of ftn_M:

$$\text{expect to equal } (\text{op} \fatsemi \text{ftn}_M) : \quad \text{ftn}_M^{\uparrow M} \fatsemi \text{op} = \text{ftn}_M^{\uparrow M} \fatsemi \text{ftn}_M = \text{ftn}_M \fatsemi \text{ftn}_M = \text{op} \fatsemi \text{ftn}_M \quad .$$

The corresponding "simple" operation (`sop`) is:

$$\text{sop} \stackrel{\text{def}}{=} \text{pu}_M^{\uparrow M} \fatsemi \text{op} = \text{pu}_M^{\uparrow M} \fatsemi \text{ftn}_M = \text{id} \quad .$$

This "simple" operation is just the identity function (of type $M^A \to M^A$). This function carries no information specific to the monad M and cannot be used to characterize M.

Example 14.9.4.17 Show that for any functor F there exists a monad M that admits a "standard" operation $\text{op} : F^M \rightsquigarrow M$. Is that "standard" operation equivalent to a "simple" operation?

Solution The required monad M is the free monad on F, which we denote for brevity by L:

$$L^A \stackrel{\text{def}}{=} A + F^{L^A} \quad .$$

The operation `op` is the injection of F^{L^A} into the disjunctive type $A + F^{L^A}$:

$$\text{op} : F^{L^A} \to L^A \quad , \qquad \text{op} \stackrel{\text{def}}{=} p^{:F^{L^A}} \to \mathbb{0}^{:A} + p = \begin{array}{|c||cc|} & A & F^{L^A} \\ \hline F^{L^A} & \mathbb{0} & \text{id} \end{array} \quad .$$

This operation satisfies the compatibility law because (see Statement 10.2.8.10):

$$\text{ftn}_L^{\uparrow F} \fatsemi \text{op} = \begin{array}{|c||cc|} & A & F^{L^A} \\ \hline F^{L^{L^A}} & \mathbb{0} & \text{ftn}_L^{\uparrow F} \end{array} \quad ,$$

$$\text{op} \fatsemi \text{ftn}_L = \begin{array}{|c||ccc|} & A & F^{L^A} & F^{L^{L^A}} \\ \hline F^{L^{L^A}} & \mathbb{0} & \mathbb{0} & \text{id} \end{array} \fatsemi \begin{array}{|c||cc|} & A & F^{L^A} \\ \hline A & \text{id} & \mathbb{0} \\ F^{L^A} & \mathbb{0} & \text{id} \\ F^{L^{L^A}} & \mathbb{0} & \overline{\text{ftn}_L^{\uparrow F}} \end{array} = \begin{array}{|c||cc|} & A & F^{L^A} \\ \hline F^{L^{L^A}} & \mathbb{0} & \text{ftn}_L^{\uparrow F} \end{array} \quad .$$

[25] See, for example, the Scala-based blog posts `https://blog.bruchez.name/2011/09/continuations-in-scala.html` and `https://dcsobral.blogspot.com/2009/07/delimited-continuations-explained-in.html`, as well as the Haskell-based discussion `https://stackoverflow.com/questions/43695653/cont-monad-shift`

The equivalent "simple" operation is the injection of F into the free monad:

$$\text{sop}: \quad F^A \to L^A \quad , \qquad \text{sop} \stackrel{\text{def}}{=} p^{:F^A} \to \mathbb{0} + p \triangleright (a^{:A} \to a + \mathbb{0})^{\uparrow F} \quad .$$

□

So, if we have some "simple" effectful operations with functors F_1, F_2, etc., we may combine them into a single effect constructor $F = F_1 + F_2 + ...$ and use a free monad on F to obtain a monad that supports all the given operations. But a free monad cannot describe any operations that are not equivalent to "simple" ones.

14.9.5 Exercises

Exercise 14.9.5.1 Define type constructors for these monad stacks and for their transformers:

(a) `Eval ◁ Try`

(b) `List ◁ Eval`

(c) `Try ◁ Writer[W, *]`

(d) `Cont[R, *] ◁ Try ◁ State[S, *]`

Exercise 14.9.5.2 Consider a monad stack $P \stackrel{\text{def}}{=} K_1 \triangleleft ... \triangleleft K_n$ where each of the monads K_i (for $i = 1, ..., n$) has a runner $\theta_i : K_i^A \to A$. Assume that the monads K_i have complete monad transformers, so that all base runners (brun_{K_i}) are known. Show that each runner θ_i can be lifted to the stack P, giving a monad morphism $\bar{\theta}_i$ with type signature:

$$\bar{\theta}_i : K_1 \triangleleft ... \triangleleft K_n \rightsquigarrow K_1 \triangleleft ... \triangleleft K_{i-1} \triangleleft K_{i+1} \triangleleft ... \triangleleft K_n \quad \text{(the stack } P \text{ without } K_i\text{)} \quad .$$

Prove that all runners $\bar{\theta}_i$ commute: $\bar{\theta}_i \fatsemi \bar{\theta}_j = \bar{\theta}_j \fatsemi \bar{\theta}_i$ for all i and j between 1 and n.

Exercise 14.9.5.3 Statement 14.9.1.1 shows that the duplicating function $\Delta : M \rightsquigarrow M \times M$ is a monad morphism for an arbitrary monad M. Prove that the same holds for all similarly defined duplicating functions of types $M \rightsquigarrow M \times M \times M$, $M \rightsquigarrow M \times M \times M \times M$, and so on.

Exercise 14.9.5.4 We know from Statement 10.3.4.6 that pu_M is a monad morphism from the identity monad (Id) to a given monad M. Show that there are no other such monad morphisms ϕ:

$$\phi : \text{Id} \rightsquigarrow M \quad , \quad \text{or equivalently,} \qquad \phi^A : A \to M^A \quad .$$

In other words, prove that any such monad morphism ϕ must be equal to M's `pure` method: $\phi = \text{pu}_M$. It means that the identity monad is an initial object in the category of monads. (An object i is an **initial object** of a category if for any object x there is a unique morphism $i \to x$.)

Exercise 14.9.5.5 **(a)** Show that for any monad M, there is only one monad morphism $\varepsilon : M \rightsquigarrow \mathbb{1}$, where $\mathbb{1}$ denotes the unit monad, $L^A = \mathbb{1}$. It means that the unit monad is a terminal object in the category of monads. (An object t is a **terminal object** in a category if for any object x there is a unique morphism $x \to t$.)

(b) Show that a monad morphism $\phi : \mathbb{1} \rightsquigarrow M$ between the unit monad and some monad M is possible only if M is also a unit monad or is isomorphic to the unit monad ($M^A \cong \mathbb{1}$).

Exercise 14.9.5.6 For any given monad M, consider a function $\text{dbl} : M^A \to M^A$ defined by:

```
def dbl[M[_]: Monad, A](m: M[A]): M[A] = for {
  _ <- m
  x <- m
} yield x
```

$$m^{:M^A} \triangleright \text{dbl} \stackrel{\text{def}}{=} m \triangleright \text{flm}_M(_^{:A} \to m)$$
$$= m \triangleright (_ \to m)^{\uparrow M} \fatsemi \text{ftn}_M \quad .$$

This function "doubles" the M-effect in a given monadic value m while keeping the value $x^{:A}$ from the second copy of m.

(a) Show that the function `dbl[M, A]` is a *monadically natural* transformation $M^A \to M^A$ with respect to the monad type parameter M. The monadic naturality law requires, for any two monads M, N and any monad morphism $\phi^{:M \rightsquigarrow N}$, the following equation to hold:

$$\text{dbl}^M \fatsemi \phi = \phi \fatsemi \text{dbl}^N \quad .$$

(b) Show that the function `dbl` is *not* a monad morphism $M \rightsquigarrow M$ (at least for some monads M).

Exercise 14.9.5.7 Consider the monads $\text{Reader}^{R,A}$ and $\text{State}^{R,A}$ (with the same fixed type R).

(a) Implement a monad morphism $\text{inS} : \text{Reader}^{R,A} \to \text{State}^{R,A}$ and prove that the laws hold.

(b) Find a natural transformation $\text{outS} : \text{State}^{R,A} \to \text{Reader}^{R,A}$ such that $\text{inS} ⨾ \text{outS} = \text{id}$.

(c) Show that there is *no* lawful monad morphism of type $\text{State}^{R,A} \to \text{Reader}^{R,A}$.

Exercise 14.9.5.8 Consider the monads $\text{Writer}^{W,A}$ and $\text{State}^{W,A}$ (with a fixed monoid type W).

(a) Implement a monad morphism $\text{inS} : \text{Writer}^{W,A} \to \text{State}^{W,A}$ and prove that the laws hold.

(b) Find a natural transformation $\text{outS} : \text{State}^{W,A} \to \text{Writer}^{W,A}$ such that $\text{inS} ⨾ \text{outS} = \text{id}$.

(c) Show that there is *no* lawful monad morphism of type $\text{State}^{W,A} \to \text{Writer}^{W,A}$.

Exercise 14.9.5.9 Suppose T_L is a lawful monad transformer for a base monad L and a foreign monad M. Can we modify T_L and construct another monad transformer for L that still satisfies all transformer laws?

One possibility is to compose $T_L^{M,A}$ with an extra layer of the monads L or M. Define U and V by:

$$U^A \stackrel{\text{def}}{=} L^{T_L^{M,A}} \quad , \qquad V^A \stackrel{\text{def}}{=} M^{T_L^{M,A}} \quad .$$

In a shorter notation, $U \stackrel{\text{def}}{=} L \circ T_L^M$ and $V \stackrel{\text{def}}{=} M \circ T_L^M$. We have the `swap` functions:

$$\begin{aligned} \text{sw}_{L,T} &: T_L^M \circ L \rightsquigarrow L \circ T_L^M \quad , \\ \text{sw}_{M,T} &: T_L^M \circ M \rightsquigarrow M \circ T_L^M \quad , \end{aligned}$$

defined using the already given methods `flift` and `blift` of T_L^M as:

$$\begin{aligned} \text{sw}_{L,T} &= \text{blift}^{\uparrow T} ⨾ \text{ftn}_T ⨾ \text{pu}_L \quad , \\ \text{sw}_{M,T} &= \text{flift}^{\uparrow T} ⨾ \text{ftn}_T ⨾ \text{pu}_M \quad . \end{aligned}$$

We can define the monad methods ftn_U and ftn_V using these `swap` functions. Will U and/or V be lawful monad transformers for L?

Exercise 14.9.5.10 For any two monads L and M and a natural transformation $\text{swap} : M \circ L \rightsquigarrow L \circ M$, assume that the method ftn_T of the monad $T \stackrel{\text{def}}{=} L \circ M$ is *defined* via `swap` by Eq. (14.11). Show that the two interchange laws will then hold for ftn_T:

$$\begin{aligned} &\text{inner interchange law :} \quad \text{ftn}_L ⨾ \text{ftn}_T = \text{ftn}_T^{\uparrow L} ⨾ \text{ftn}_L \quad , \\ &\text{outer interchange law :} \quad \text{ftn}_T ⨾ \text{ftn}_M^{\uparrow L} = \text{ftn}_M^{\uparrow L \uparrow M \uparrow L} ⨾ \text{ftn}_T \quad . \end{aligned}$$

Exercise 14.9.5.11 With the same assumptions as Exercise 14.9.5.10 and additionally assuming the inner and outer identity laws for `swap` (see Statement 14.3.2.1), show that the monad $T \stackrel{\text{def}}{=} L \circ M$ satisfies two "`pure` compatibility" laws:

$$\begin{aligned} &\text{inner pure compatibility law :} \quad \text{ftn}_L = \text{pu}_M^{\uparrow L} ⨾ \text{ftn}_T \quad : L^{L^{M^A}} \to L^{M^A} \quad , \\ &\text{outer pure compatibility law :} \quad \text{ftn}_M^{\uparrow L} = \text{pu}_L^{\uparrow T} ⨾ \text{ftn}_T \quad : L^{M^{M^A}} \to L^{M^A} \quad , \end{aligned}$$

or, expressed equivalently through the `flatMap` methods instead of `flatten`:

$$\begin{aligned} \text{flm}_L\left(f^{:A \to L^{M^B}}\right) &= \text{pu}_M^{\uparrow L} ⨾ \text{flm}_T(f) \quad , \\ \left(\text{flm}_M\left(f^{:A \to M^B}\right)\right)^{\uparrow L} &= \text{pu}_L^{\uparrow T} ⨾ \text{flm}_T\left(f^{\uparrow L}\right) \quad . \end{aligned}$$

Exercise 14.9.5.12 Define a "search monad" by $\text{Search}^{P,A} \stackrel{\text{def}}{=} (A \to \mathbb{1} + P) \to \mathbb{1} + A$. The function:

```
def finder[A, P](s: (A => Option[P]) => A): (A => Option[P]) => Option[A] = {
  p: (A => Option[P]) => p(s(p)).map { _ => s(p) }
}
```

$$\text{finder} : \text{Sel}^{\mathbb{1}+P,A} \to \text{Search}^{P,A} \quad , \qquad \text{finder}\,(s^{:\text{Sel}^{\mathbb{1}+P,A}}) \stackrel{\text{def}}{=} p^{:A\to\mathbb{1}+P} \to p(s(p)) \triangleright \begin{array}{|c||cc|} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ P & \mathbb{0} & s(p) \end{array} \quad ,$$

transforms a selector monad into a search monad. Show that `finder` is *not* a monad morphism $\text{Sel}^{\mathbb{1}+P,\bullet} \rightsquigarrow \text{Search}^{P,\bullet}$.

Exercise 14.9.5.13* Show that the following function (`scc`) is a monad morphism $\text{Sel}^{P,\bullet} \rightsquigarrow \text{Cont}^{P,\bullet}$:

```
def scc[P, A](s: (A => P) => A): (A => P) => P = { f: (A => P) => f(s(f)) }
```

$$\text{scc} : \text{Sel}^{P,A} \to \text{Cont}^{P,A} \quad , \qquad \text{scc}\,(s^{:(A\to P)\to A}) \stackrel{\text{def}}{=} f^{:A\to P} \to f(s(f)) \quad .$$

Hint: Use the flipped Kleisli technique to simplify the proof of the composition law.

Exercise 14.9.5.14 **(a)** Show that the product of three monads $K^\bullet \times L^\bullet \times M^\bullet$ is again a monad.

(b) Show that the type isomorphism $(K^\bullet \times L^\bullet) \times M^\bullet \cong K^\bullet \times (L^\bullet \times M^\bullet)$ is also a *monad* isomorphism.

Exercise 14.9.5.15 The construction of the free pointed monad on M, namely $N^A \stackrel{\text{def}}{=} A + M^A$ (where M is a given monad), uses a helper function γ_M defined by:

$$\gamma_M : A + M^A \to M^A \quad , \qquad \gamma_M \stackrel{\text{def}}{=} \begin{array}{|c||c|} & M^A \\ \hline A & \text{pu}_M \\ M^A & \text{id} \end{array} \quad .$$

Show that γ_M is a monad morphism between the monads N and M.

Exercise 14.9.5.16 For any monads K and M and a given monad morphism $\phi : K \rightsquigarrow M$, show that one can implement a corresponding monad morphism $\psi : L \rightsquigarrow N$ between free pointed monads $L^A \stackrel{\text{def}}{=} A + K^A$ and $N^A \stackrel{\text{def}}{=} A + M^A$.

Exercise 14.9.5.17 **(a)** Given two natural transformations $\phi : K^A \to M^A$ and $\chi : L^A \to M^A$, show that the function ψ defined by:

$$\psi : K^A + L^A \to M^A \quad , \qquad \psi \stackrel{\text{def}}{=} \begin{array}{|c||c|} & M^A \\ \hline K^A & \phi \\ L^A & \chi \end{array} \quad ,$$

is also a natural transformation.

(b) Given a natural transformation $\psi : K^A + L^A \to M^A$, show that the functions $\phi : K^A \to M^A$ and $\chi : L^A \to M^A$ defined by:

$$\phi \stackrel{\text{def}}{=} (k \to k + \mathbb{0}) ⨾ \psi \quad , \qquad \chi \stackrel{\text{def}}{=} (l \to \mathbb{0} + l) ⨾ \psi \quad ,$$

are both natural transformations.

Exercise 14.9.5.18 For a given monad K, define L as the free pointed monad $L^A \stackrel{\text{def}}{=} A + K^A$.

(a) Show that the monad transformer for L *cannot* be defined as the monad $T_L^{M,A} \stackrel{\text{def}}{=} A + T_K^{M,A}$ because then (at least for some monads K and M) there is no monad morphism $\text{flift} : M \rightsquigarrow T_L^{M,A}$.

(b) Given a runner $\theta_K : K \rightsquigarrow M$, where M is an arbitrary (but fixed) target monad, implement a runner $\theta_L : L \rightsquigarrow M$ and show that θ_L is a monad morphism when θ_K is one.

(c) Show that it is *not* possible to implement a monad morphism $\theta_K : K \rightsquigarrow M$ given a monad morphism $\theta_L : L \rightsquigarrow M$, at least for some monads K and M.

(d)* Given a monad morphism $\theta_L : L^A \to A$, implement a lawful monad morphism $\theta_K : K^A \to A$.

Exercise 14.9.5.19 For a given monad K, define L as the free pointed monad $L^A \stackrel{\text{def}}{=} A + K^A$. Given a monad morphism $\phi : K \rightsquigarrow L$, implement a monad morphism $\delta : K \rightsquigarrow \text{Opt}$.

Exercise 14.9.5.20 It follows from Statement 14.8.4.1 that adding a type quantifier to a monad gives again a monad. Show that adding a type quantifier to any inductive typeclass (Section 8.5.4) gives again an inductive typeclass. In detail: assume an inductive typeclass defined via a structure functor $P^{R,\bullet}$ that has an extra type parameter R. All methods of the typeclass are then contained in a value of type $P^{R,A} \to A$, where the type R is fixed. Show that $\forall R.\, P^{R,A} \to A$ is again an inductive typeclass.

Exercise 14.9.5.21* Assume an arbitrary monad M and define $L^A \stackrel{\text{def}}{=} \mathbb{1} + A \times M^{L^A}$.

(a) Show that any (fully parametric in M) implementation of `flatten` for L (with type signature $\text{ftn}_L : L^{L^A} \to L^A$) must truncate the nested effectful list whenever it encounters an empty sub-list. To see the truncating behavior in a simpler way, take $M = \text{Id}$, which reduces L to the ordinary list ($L^A \stackrel{\text{def}}{=} \mathbb{1} + A \times L^A$). Show that the `flatten` method for L must ignore all sub-lists after an empty sub-list. For example, $\text{ftn}_L([[1,2],[3,4],[],[5,6]])$ must ignore the values $[5,6]$.

(b) Show that the monad laws do *not* hold for a non-standard `flatten` method that truncates the list after the first empty sub-list, such that $\text{ftn}_L([[1,2],[3,4],[],[5,6]]) = [1,2,3,4]$. (Compare with Exercise 10.2.9.4 where the `flatten` method returns `Nil` whenever there are any empty sub-lists.)

(c) Show that the functor L *cannot* be a monad transformer for the non-standard `List` monad defined in Exercise 10.2.9.4, because we cannot implement a lawful `flift` function of type $M^A \to L^A$.

Exercise 14.9.5.22* **(a)** The **composed codensity monad** $\text{Cod}_F^{L,A}$ is defined by:

$$\text{Cod}_F^{L,A} \stackrel{\text{def}}{=} \forall X.\, (A \to F^X) \to F^{L^X} \quad ,$$

where L is any monad and F is any functor. One may define the monad instance for $\text{Cod}_F^{L,A}$ via the flipped Kleisli composition:

$$f^{:\forall X.\,(B\to F^X)\to A\to F^{L^X}} \tilde{\diamond}\, g^{:\forall Y.\,(C\to F^Y)\to B\to F^{L^Y}} \stackrel{\text{def}}{=} \forall Z.\, k^{:C\to F^Z} \to \left(k \triangleright g^Z \triangleright f^{L^Z}\right) ⨾ \text{ftn}_L^{\uparrow F} \quad .$$

Prove that the type constructor $\text{Cod}_F^{L,\bullet}$ is a lawful monad.

Hints: Use the flipped Kleisli formulation of the monad laws. Assume that values of type $\text{Cod}_F^{L,A}$ are natural transformations that satisfy an appropriate naturality law.

(b) Show that the quantifier $\forall X$ cannot be removed from the definition of $\text{Cod}_F^{L,A}$ even if $F^A \stackrel{\text{def}}{=} A$: for a fixed type X and for some $L^\bullet$, the type constructor $P^A \stackrel{\text{def}}{=} (A \to X) \to L^X$ is *not* a monad.

Exercise 14.9.5.23 Rigid functors are defined in Section 14.9.3.

(a) Show that the functor $F^A \stackrel{\text{def}}{=} A \times A \times A$ is rigid.

(b) Show that the functor $F^A \stackrel{\text{def}}{=} \mathbb{1} + A \times A$ is not rigid.

(c) Show that the functor $F^A \stackrel{\text{def}}{=} (A \to Z) \to Z$ is not rigid (Z is a fixed type).

Exercise 14.9.5.24 The "Day convolution" operation ($\star$) was defined in Eq. (14.6) of Section 14.2.6. Prove the associativity law of Day convolution: $(L \star M) \star N \cong L \star (M \star N)$.

Part IV

Discussions

15 Summary of the book

The book has been written in a tutorial format, motivating and deriving the results gradually. This makes it easier to learn but harder to navigate. In this summary chapter, the results are listed without detailed explanations or proofs, referring to the places of the book where more detail is found.

15.1 Main points and results by chapter

15.1.1 From Chapter 1

Nameless functions are implicitly used in the standard mathematical notation such as $\sum_{n=1}^{100}(n^2+n)$.

Functional programming improves upon this kind of mathematical notation by using nameless functions explicitly and consistently. For example, the computation $\sum_{n=1}^{100}(n^2+n)$ is implemented in Scala as `(1 to 100).map(n => n*n + n).sum`.

The methods `sum` and `product` are examples of "aggregations". The methods `filter` and `takeWhile` implement computations that select some elements from a sequence or truncate a sequence once a certain condition is achieved. Those methods are examples of "transformations".

Functional programming is based on formulating the solution of a problem as a mathematical expression and then translating the mathematical formula into code. Mathematical formulas do not use loops and do not modify variables. Instead, iteration is expressed using special operators such as $\sum$. So, functional programming encodes transformations and aggregations without loops by using special new operations (`map`, `filter`, and so on).

15.1.2 From Chapter 2

***

15.2 Topics not covered in this book

This book focuses on mathematical theory that has proven its relevance to the functional programming practice. This section lists some topics that were omitted from this edition of the book, and explains why.

15.2.1 Recursion in functor blocks and trampolines

Recursion with applicative and monadic functors

15.2.2 Strictness and laziness

15.2.3 Combining typeclass laws

15.2.4 Lenses, prisms, and other "functional optics"

15.2.5 Comonads and comonad transformers

15.2.6 Proving program correctness via dependent types

15.2.7 Linear types and other non-standard types

***

15.3 Additional exercises and problems

The following is a sample set of problems that can be solved using techniques developed in this book.

15.3.1 Exercises

1. Find the smallest integer expressible as a sum of two cubed integers in more than one way.

2. Implement a function `norepeat` that removes consecutive repetitions from sequences, e.g.:

```
def norepeat[A]: Seq[A] => Seq[A] = ???

scala> norepeat(Seq(1,2,2,1,1,3,3,3,0,3))
res0: Seq[Int] = Seq(1,2,1,3,0,3)
```

3. Read a text file, split it by spaces into words, and print the word counts in decreasing order.

4. (Exercise 4-1 from Hu Zhenjiang's course http://www.prg.nii.ac.jp/course/2015/msp15/) Express the `filter` method for sequences via `flatMap`:

```
def filter[A](p: A => Boolean)(s: Seq[A]): Seq[A] = s.flatMap { a => ??? }
```

5. (Bird-de Moor, page 20) Derive the following identity between functions $F^A \to F^A$, for any filterable functor F and any predicate $p^{:A\to 2}$:

$$\text{filt}_F(p) = (\Delta \fatsemi \text{id} \boxtimes p)^{\uparrow F} \fatsemi \text{filt}_F(\pi_2) \fatsemi \pi_1^{\uparrow F} \quad .$$

6. Define a monoid of partial functions with fixed types $P \to Q$:

```
final case class PFM[P, Q](pf: PartialFunction[P, Q])
// After defining a monoid instance, the following code must work:
val p1 = PFM[Option[Int], String] { case Some(3) => "three" }
val p2 = PFM[Option[Int], String] {
  case Some(20)   => "twenty"
  case None       => "empty"
}
p1 |+| p2 // Must be the same as the concatenation of all ‘case‘ clauses.
```

7. Consider a typeclass called "`Splittable`" for functors F that have an additional method:

$$\text{split}^{A,B} : F^{A+B} \to F^A + B \quad ,$$

satisfying a non-degeneracy law:

$$(x^{:A} \to x + \mathbb{0}^{:B})^{\uparrow F} \fatsemi \text{split} = y^{:F^A} \to y + \mathbb{0}^{:B} \quad ,$$

and a special associativity law, which is an equation for functions of type $F^{A+B+C} \to F^A + B + C$:

$$\text{split}^{A+B,C} \fatsemi \left| \begin{array}{c|c} & F^A + B \quad C \\ \hline F^{A+B} & \text{split}^{A,B} \quad \mathbb{0} \\ C & \mathbb{0} \quad \text{id} \end{array} \right| = \text{split}^{A,B+C} \quad .$$

Show that all polynomial functors $F^\bullet$ belong to this typeclass. Show that exponential functors such as $F^A \stackrel{\text{def}}{=} Z \to A$ do not.

8. Given a monad M and a fixed type Z, consider the functor $F^A \stackrel{\text{def}}{=} (A \to M^Z) \to Z$. Show that F is a semimonad but not a full monad. Hint: use the flipped Kleisli technique.

9. Given two fixed types P, Q that are not known to be equivalent, consider the contrafunctors $F^A \stackrel{\text{def}}{=} ((A \to P) \to P) \to Q$ and $G^A \stackrel{\text{def}}{=} A \to Q$. Show that there exist natural transformations $F^\bullet \rightsquigarrow G^\bullet$ and $G^\bullet \rightsquigarrow F^\bullet$. Show that these transformations are *not* isomorphisms.

10. Given two fixed types P, Q that are not known to be equivalent, show that the functor $L^A \stackrel{\text{def}}{=} (((A \to P) \to Q) \to Q) \to P$ is a semimonad but not a full monad. (When $P \cong Q$, the functor L is also not a full monad because L is then equivalent to a composition of the continuation monad with itself. See Exercise 10.2.9.19.)

11. When M is a given semimonad, show that $M \circ M \circ ... \circ M$ (with finitely many M) is also a lawful semimonad.

12. If M is any lawful monad then M^{A+M^A} is also a lawful monad.

13. If M is a commutative monad then $M \circ M$ is also a lawful commutative monad.

14. If $M^\bullet$ is a commutative monad and W is a commutative monoid then the monoid M^W is commutative.

15. ** Define a type constructor `Triang[A]` representing "triangular matrices" with elements of type `A`. Example values t_1, t_2, t_3 of type `Triang[A]` could be of the form:

$$t_1 = \left| \begin{array}{l} a_1 \end{array} \right| \quad , \qquad t_2 = \left| \begin{array}{ll} a_1 & \\ a_2 & a_3 \end{array} \right| \quad , \qquad t_3 = \left| \begin{array}{llll} a_1 & & & \\ a_2 & a_3 & & \\ a_4 & a_5 & a_6 & \\ a_7 & a_8 & a_9 & a_{10} \end{array} \right| \quad .$$

Unlike `List[List[A]]`, it should *not* be possible to have a value of type `Triang[A]` that has an unexpected shape:

$$t = \left| \begin{array}{llll} a_1 & & & \\ a_2 & a_3 & & \\ a_7 & a_8 & a_9 & a_{10} \end{array} \right| \quad \text{is not of type Triang}^A \quad .$$

Implement `Functor`, `Applicative`, and `Traversable` instances for `Triang`.

16. Define a monad transformer $T^{M,A}_{\text{Cod}^L_F}$ for the composed codensity monad (Exercise 14.9.5.22) with type parameters F (an arbitrary but fixed functor), L (an arbitrary but fixed monad), M (a foreign monad), and A (the value type). Find out which laws hold for that transformer.

17. ** **(a)** For any fixed type Z, functor F and lawful monad P, show that $L^A \stackrel{\text{def}}{=} F^{A \to P^Z} \to P^A$ is a lawful monad. **(b)** Show that L's monad transformer is $T^{M,A}_L \stackrel{\text{def}}{=} F^{A \to T^{M,Z}_P} \to T^{M,A}_P$, where $T^{M,A}_P$ is P's monad transformer.

15.3.2 Open problems

The author of this book does not know how to answer the following questions and also could not find any answers in existing books or papers.

Problem 15.3.2.1 Do all polynomial functors of the form $P^A_n \stackrel{\text{def}}{=} \mathbb{1} + \overbrace{A \times A \times ... \times A}^{n \text{ times}, n \geq 2}$ fail to be monads? An example is the functor $P^A_2 \stackrel{\text{def}}{=} \mathbb{1} + A \times A$, which is not a monad because all possible implementations of `pure` and `flatMap` methods for P_2 fail the monad laws.[1]

Problem 15.3.2.2 Section 10.3.5 shows some constructions that make new monads:

- The polynomial monad $F^A \stackrel{\text{def}}{=} Z + W \times A$, where W is a monoid and Z is a fixed type.
- The free pointed monad $L^A \stackrel{\text{def}}{=} A + F^A$, where F is a monad.
- The product monad $L^A \stackrel{\text{def}}{=} F^A \times G^A$, where F and G are monads.
- The monad $L^A \stackrel{\text{def}}{=} F^{Z+W\times A}$, where F is a monad, W is a monoid, and Z is a fixed type.

If we do not assume any existing monads and just keep applying these constructions, we will obtain a number of polynomial monads. But are there any polynomial monads *not* obtained by a chain of these constructions?

Problem 15.3.2.3 By Statement 11.3.6.5, any co-pointed applicative functor of the form $L^A \stackrel{\text{def}}{=} A \times G^A$ satisfies the compatibility law (11.10). Statement 11.3.6.6 shows that $L^A \stackrel{\text{def}}{=} Z \times (Z \to A)$ is applicative and co-pointed but fails the compatibility law. Does there exist any co-pointed applicative functor that satisfies the law (11.10) but is *not* of the form $A \times G^A$ with some applicative functor $G^\bullet$?

Problem 15.3.2.4 Show that the applicative naturality law (12.17) and the composition law (12.23) of `traverse` guarantee that `traverse` collects each F-effect exactly once. (Can we use the composition of an applicative functor F with a `State` monad to count the number of times F-effects were collected?)

Problem 15.3.2.5 Monad transformers are defined in different ways for different monads. If someone comes up with a new monad, it is not certain that the new monad's transformer will be obtained through one of the known methods. Can we prove that a monad transformer will exist for every monad whose `pure` and `flatMap` methods are implemented via fully parametric code? Given the code for an arbitrary monad, can we derive an implementation of the corresponding monad transformer?

Problem 15.3.2.6 For certain monads L, the monad transformers T_L can be defined using a suitable `swap` function. Is this always the case for any monad stacks built out of such monads? If each of the monads L_1, L_2, ..., L_k admits a transformer defined via a lawful `swap` function, will the monad $L_1 \triangleleft L_2 \triangleleft ... \triangleleft L_k$ also admit a transformer with a `swap` function? (See Section 14.9.2 for some partial results.)

[1] See discussion here: https://stackoverflow.com/questions/49742377

Problem 15.3.2.7 Are there any monadically natural monad morphisms $M \rightsquigarrow M$ that are not identity functions? (Equivalently, any non-identical natural transformations $\text{Id}^\bullet \rightsquigarrow \text{Id}^\bullet$ between identity functors in the category of monads?) If it were possible to prove that any natural monad morphism $M \rightsquigarrow M$ equals an identity function, there would be no need to verify the non-degeneracy law for monad transformers' base runners (see page 589).

We look for a monad morphism $\varepsilon^{M,A} : M^A \to M^A$ that is defined for all monads M and is monadically natural in the parameter M. So, the function ε must satisfy the following laws:

$$\begin{aligned}
&\text{naturality law :} && \varepsilon^{M,A} \fatsemi (f^{:A\to B})^{\uparrow M} = (f^{:A\to B})^{\uparrow M} \fatsemi \varepsilon^{M,B} \quad , \\
&\text{monad morphism laws :} && \text{pu}_M \fatsemi \varepsilon = \text{pu}_M \quad , \qquad \varepsilon^{\uparrow M} \fatsemi \varepsilon \fatsemi \text{ftn}_M = \text{ftn}_M \fatsemi \varepsilon \quad , \\
&\text{monadic naturality law :} && \varepsilon^{M,A} \fatsemi \phi^{:M^{:A}\to N^A} = \phi^{:M^A\to N^A} \fatsemi \varepsilon^{N,A} \quad ,
\end{aligned}$$

where $f^{:A\to B}$ is an arbitrary function, M and N are arbitrary monads, and $\phi : M \rightsquigarrow N$ is an arbitrary monad morphism. We need to prove that any such ε must be an identity function, $\varepsilon = \text{id}^{:M^A\to M^A}$, or to find an example of such ε not equal to identity.[2]

Problem 15.3.2.8 "Rigid functors" are defined in Section 14.9.3.

(a) Are there any rigid functors that are not monads?

(b) Are there any rigid functors that are not applicative?

(c) Is it true that any applicative rigid functor is a monad?

Problem 15.3.2.9 Let L be a fixed monad and H be an L-filterable contrafunctor. Then the functor $F^A \stackrel{\text{def}}{=} H^A \to L^A$ is a lawful monad (see Section 10.3.6). What is a monad transformer for the monad F?

Two nontrivial examples of L-filterable contrafunctors are $H^A \stackrel{\text{def}}{=} A \to L^Z$ and $H^A \stackrel{\text{def}}{=} L^A \to Z$ (where Z is a fixed type). For these cases, the monad transformers are defined by:

$$\begin{aligned}
&\text{monad: } (A \to L^Z) \to L^A \quad , && \text{transformer: } (A \to T_L^{M,Z}) \to T_L^{M,A} \quad , \\
&\text{monad: } (L^A \to Z) \to L^A \quad , && \text{transformer: } (T_L^{M,A} \to Z) \to T_L^{M,A} \quad ,
\end{aligned}$$

where T_L^M is the monad L's transformer applied to the foreign monad M.

The problem is to implement the monad F's transformer for an arbitrary L-filterable contrafunctor H and to prove that the transformer laws hold.

Problem 15.3.2.10 Assume an arbitrary unknown monad M and define recursively $L^A \stackrel{\text{def}}{=} \mathbb{1} + A \times M^{L^A}$. Can one define a lawful monad instance for the functor L? (This is the `List` monad's transformer without the outer layer of M. See Exercise 14.9.5.21.)

Problem 15.3.2.11 These questions concern the `ListT` transformer:

$$T^A \stackrel{\text{def}}{=} M^{L^A} \quad , \qquad L^A \stackrel{\text{def}}{=} \mathbb{1} + A \times M^{L^A} \quad ,$$

where M is an arbitrary foreign monad. Normally, we cannot implement fully parametric base runner $T^A \to M^A$ because we cannot have a fully parametric runner $\text{List}^A \to A$ operating on arbitrary types A. However, for a *monoid* type R with binary operation $\oplus_R$ and empty element e_R, the type signature $\text{List}^R \to R$ is implemented by the standard `reduce` operation:

$$\text{reduce} : \text{List}^R \to R \quad , \qquad \text{reduce} \stackrel{\text{def}}{=} \left| \begin{array}{c|c} & R \\ \hline \mathbb{1} & 1 \to e_R \\ R \times \text{List}^R & h \times t \to h \oplus_R \overline{\text{reduce}}\,(t) \end{array} \right| \quad .$$

We can similarly implement a base runner (`brun`) for the transformer T_{List} if we restrict its usage to *monoid* types R. The function `brun` with the type signature $M^{L^R} \to M^R$ aggregates all elements of the

[2] See discussion here: https://stackoverflow.com/questions/61444425/

effectful list into a single value of type M^R (which is also a monoid type):

$$\text{brun} : M^{L^R} \to M^R \quad , \qquad \text{brun} \stackrel{\text{def}}{=} \text{flm}_M \left(\begin{array}{c|c} & M^R \\ \hline \mathbb{1} & 1 \to \text{pu}_M(e_R) \\ R \times M^{L^R} & h \times t \to \text{pu}_M(h) \oplus_M \overline{\text{brun}}(t) \end{array} \right) \quad .$$

Here, we use the binary operation $\oplus_M$ of the monoid M^R, which is defined by:

$$p^{:M^R} \oplus_M q^{:M^R} \stackrel{\text{def}}{=} p \triangleright \text{flm}_M \left(u^{:R} \to q \triangleright (v^{:R} \to u \oplus_R v)^{\uparrow M} \right) \quad .$$

(a) Is `brun` a monoid morphism $T^A \to A$? (Note that T^A is a monoid since T is a lawful monad.) The monoid morphism identity law holds for `brun`. Does the composition law hold?

(b) Do the monad morphism laws of `brun` hold when restricted to a monoid type A?

$$\text{for all monoid types } A: \quad a^{:A} \triangleright \text{pu}_T \,\fatsemi\, \text{brun} = a^{:A} \triangleright \text{pu}_M \quad ,$$

$$\text{composition law}: \quad p^{:T^{T^A}} \triangleright \text{ftn}_T \,\fatsemi\, \text{brun} = p^{:T^{T^A}} \triangleright \text{brun} \,\fatsemi\, \text{brun}^{\uparrow M} \,\fatsemi\, \text{ftn}_M \quad .$$

(If so, Exercise 12.3.9.14 would show that `brun` is also a *monoid* morphism $M^{L^A} \to M^A$.)

Problem 15.3.2.12 The continuation monad's operations `reset` and `shift` are defined by:

$$\text{reset} : \forall S. \text{Cont}^{R,R} \to \text{Cont}^{S,R} \quad , \qquad \text{reset} \stackrel{\text{def}}{=} c^{:(R\to R)\to R} \to k^{:R\to S} \to k(c(\text{id})) \quad ,$$

$$\text{shift} : \forall A. ((A \to R) \to \text{Cont}^{R,R}) \to \text{Cont}^{R,A} \quad , \qquad \text{shift} \stackrel{\text{def}}{=} g^{:(A\to R)\to \text{Cont}^{R,R}} \to k^{:A\to R} \to g(k)(\text{id}) \quad .$$

How to lift these operations to an arbitrary monad stack P that contains a continuation monad?[3] What are the appropriate type signatures for the lifted operations?

Problem 15.3.2.13 For any fully parametric type constructor P^A covariant in A, the lifting of a function $f^{:A\to B}$ to P is performed via the `fmap` method of P, so that `fmap(f)` is a function of type $P^A \to P^B$ denoted by $f^{\uparrow F}$ in this book. The standard code of `fmap` is defined by induction on the type structure of P and satisfies the functor laws, as shown in Chapter 6. The question is to show that there is no non-standard, alternative implementation `fmap'` that still satisfies the functor laws. If the code of `fmap'` is fully parametric, Section D.1.2 shows that `fmap' = fmap`. However, parametricity does not seem to follow from functor laws alone. Does there exist an implementation of `fmap'` that satisfies the functor laws but is *not* fully parametric?

[3] See https://github.com/renormalist/pugs/blob/master/src/Pugs/AST/Eval.hs for custom code (in Haskell) that lifts `reset` and `shift` to the `ContT` monad transformer.

16 "Applied functional type theory": A proposal

What exactly is the extent of "theory" that a software engineer should know in order to be a proficient functional programmer? This book proposes an answer to that question by presenting a coherent body of theoretical knowledge that, in the author's view, *is* the theory that underlies the practice of functional programming and guides software engineers in writing code. This body of knowledge may be viewed as a new emerging sub-branch of computer science, tentatively called **applied functional type theory** (AFTT).

In order to discover the proper scope of AFTT, this book appraises the various inventions made in the field of functional programming in the last 30 years, such as the "functional pearls" papers[1] and various online tutorials, looking for theoretical material that has demonstrated its pragmatic usefulness. As a first step towards formulating AFTT from the ground up, the results are presented in the form of a tutorial, with motivations and rigorous derivations of substantially all relevant mathematical facts.

In this book, code examples are written in Scala because the author is fluent in that language. However, most of the material will work equally well in Haskell, OCaml, and other FP languages. This is because AFTT is the science of functional programming and not a set of tricks specific to Scala or Haskell. An advanced user of any functional programming language will have to face the same questions and struggle with the same practical issues.

16.1 AFTT is not covered by courses in computer science

Traditional courses of computer science (algorithms and data structures, complexity theory, distributed systems, databases, network systems, compilers, operating systems) are largely not relevant to AFTT. Courses in programming language theory are more relevant but are not presented at an appropriate level. To an academic computer scientist, the theory behind Haskell is "System F_ω", a version of λ-calculus.[2] This theory guided the design of the Haskell language and defines rigorously what a Haskell program means in a mathematical sense. The theory behind Scala is called the "DOT" (dependent object type) calculus.[3] That theory guided the design of Scala version 3.

However, a practicing Haskell or Scala programmer is not concerned with designing Haskell or Scala, or with proving theoretical properties of those languages. Instead, the programmer is mainly concerned with *using* a chosen programming language to write code.

Knowing how to prove various properties of DOT or System F_ω will not actually help programmers to write code. So, these theories are outside the scope of AFTT. The practice of functional programming does not require graduate-level theoretical studies.

As an example of theoretical material that *is* within the scope of AFTT, consider applicative functors (Chapter 11). It is helpful for a practicing functional programmer to be able to recognize and use applicative functors. An applicative functor is a data structure specifying declaratively some operations that can run independently of each other. Programs may combine these operations, execute them in parallel, check for validity, or refactor for optimization or better maintainability.

[1] https://wiki.haskell.org/Research_papers/Functional_pearls

[2] https://babel.ls.fi.upm.es/~pablo/Papers/Notes/f-fw.pdf

[3] https://www.scala-lang.org/blog/2016/02/03/essence-of-scala.html

To use this functionality, the programmer must begin by checking whether a given data structure satisfies the laws of applicative functors. In a given application, the choice of a data structure may be dictated in part by the business logic. The programmer first writes down the type of that data structure and the code implementing the required methods. The programmer can then check whether the laws hold. The data structure and the code may need to be adjusted in order to fit the definition of an applicative functor and to make the laws hold.

So, the programmer needs to perform a certain amount of symbolic derivations before coding. The derivations can be done using pen and paper by writing equations in a concise mathematical notation. Once the laws are verified, the programmer proceeds to write code.

The mathematical proofs and derivations assure that the chosen data structure will satisfy the laws of applicative functors, no matter how the rest of the program is written. So, for example, it is assured that the relevant operations can be automatically parallelized and will still work correctly. In this way, AFTT directly guides the programmer and helps write correct code.

Applicative functors were discovered by practitioners who were using Haskell to implement parser combinators for compilers. However, applicative functors are not a feature of Haskell. Rather, they are a design pattern that may be used in Scala or in any other functional programming language. A prominent example of an applicative functor is Apache Spark's `RDD` data type, which is widely used for implementing large-scale parallel computations.[4] And yet, no standard computer science course or textbook defines applicative functors, motivates their laws, explores their structure on examples, or shows data types that are *not* applicative functors (and explains why not).

16.2 AFTT is not category theory, type theory, or formal logic

One often hears that functional programming is based on category theory.[5] Indeed, the material shown in this book includes a (small) number of notions from category theory, as well as from formal logic and type theory. However, software engineers would not benefit from traditional academic courses in those subjects: their presentation is too abstract and at the same time lacks specific results necessary for practical programming. Those courses answer questions that academic mathematicians have, not questions that practicing functional programmers have.

There exist books intended as presentations of category theory for computer scientists[6] or for programmers.[7] However, those books do not cover certain concepts relevant to programming, such as applicative[8] or traversable functors. Instead, those books dwell on concepts (e.g., limits, enriched categories, topoi) that have no applications in practical functional programming today.

Typical questions in academic books are "Is X an introduction rule or an elimination rule" and "Does property Y hold in non-small categories or only in the category of sets". Questions a Scala programmer might ask are "Can we compute a value of type `Either[Z, R => A]` from a value of type `R => Either[Z, A]`" and "Is the type constructor `F[A] = Option[(A,A,A)]` a monad or only an applicative functor". The scope of AFTT includes answering the last two questions but *not* the first two.

A software engineer hoping to understand the theory behind functional programming will not find the concepts of filterable, applicative, or traversable functors in any currently available books on category theory, including books intended for programmers. And yet these concepts are necessary for correct implementations of the important and widely used operations `filter`, `zip`, and `fold`.

To compensate for the lack of AFTT textbooks, programmers have written many online tutorials, aiming to explain the theoretical concepts necessary for practical work. The term "monad tutorial" became infamous because so many were posted online.[9] Tutorials were also written about applica-

[4] https://spark.apache.org/docs/latest/rdd-programming-guide.html
[5] https://www.47deg.com/blog/science-behind-functional-programming/
[6] See, e.g., https://www.amazon.com/dp/0262660717 or https://www.math.mcgill.ca/triples/Barr-Wells-ctcs.pdf
[7] https://github.com/hmemcpy/milewski-ctfp-pdf
[8] Applicative functors are known in mathematics as "monoidal": https://en.wikipedia.org/wiki/Monoidal_functor
[9] https://www.johndcook.com/blog/2014/03/03/monads-are-hard-because/

tive functors, traversable functors, free monads, etc., showing a real unfulfilled need for presenting practice-relevant fragments of theory in an applied setting.

For example, "free monads" became popular in the Scala community around 2015. Many talks about free monads were presented at Scala engineering conferences, giving different implementations but never formulating rigorously the properties required for a piece of code to be a valid implementation of the free monad. Without knowing the required mathematical properties of free monads, a programmer cannot make sure that a given implementation is correct. However, books on category theory define free monads in a way that is unsuitable for programming applications (a free monad is an adjoint functor to a forgetful functor from a Kleisli category to the category of sets).[10] Such "academic" definitions can be used neither as guidance for writing code or checking code correctness, nor as a conceptual explanation that a learner would find helpful.

Perhaps the best selection of AFTT tutorial material today can be found in the "Haskell Wikibooks".[11] However, those tutorials are incomplete and limited to explaining the use of Haskell. Many of them are suitable neither as a first introduction nor as a reference on AFTT. Also, the Haskell Wikibooks tutorials rarely show any derivations of laws or explain the required techniques.

Apart from referring to some notions from category theory, AFTT also uses concepts from type theory and formal logic. However, existing textbooks on type theory and formal logic focus on domain theory and proof theory. From a practicing programmer's viewpoint, these books present a lot of difficult-to-learn material that will not help in writing code. At the same time, those academic books never mention practical techniques used in many functional programming libraries today, such as reasoning about and implementing types with quantifiers, types parameterized by type constructors, or partial type-level functions (known as "typeclasses").

The proper scope of AFTT is to help the programmer with practical tasks such as:

1. Deciding whether two data types are equivalent and implementing the isomorphism transformations. For example, the Scala type `(A, Either[B, C])` is equivalent to `Either[(A, B), (A, C)]`, but the type `A => Either[B, C]` is *not* equivalent to `Either[A => B, A => C]`.

2. Checking whether a definition of a recursive type is "valid", i.e., does not lead to infinite loops. A simple example of an "invalid" recursive type definition in Scala is `class Bad(x: Bad)`. A small change transforms that example into a "valid" recursive type: `class Good(x: Option[Good])`.

3. Deciding whether a function with a given type signature can be implemented. For example,

```
def f[Z,A,R]: (R => Either[Z, A]) => Either[Z, R => A] = ???   // Cannot be implemented.
def g[Z,A,R]: Either[Z, R => A] => (R => Either[Z, A]) = ???   // Can be implemented.
```

4. Deriving an implementation of a function from its type signature and checking required laws. For example, deriving the `flatMap` method and checking its laws for the `Reader` monad:

```
def flatMap[Z, A, B](r: Z => A)(f: A => Z => B): Z => B = ???
```

5. Deriving a simpler but equivalent code by calculating with functions and laws.

These are real-world applications of type theory and the Curry-Howard correspondence, but existing books on type theory and logic do not give practical recipes for performing these tasks.[12]

Books such as *Scala with Cats*,[13] *Functional programming simplified*,[14] and *Functional programming for mortals*[15] are primarily focused on explaining practical aspects of functional programming and do not derive the mathematical laws for, e.g., filterable, monadic, applicative, or traversable functors.

[10] "*A monad is just a monoid in the category of endofunctors. What's the problem?*" as the joke goes. For background information about that joke, see https://stackoverflow.com/questions/3870088/

[11] https://en.wikibooks.org/wiki/Haskell

[12] Task 5 is addressed in several programming-oriented books such as *Pearls of functional algorithm design* by Richard Bird (https://www.cambridge.org/9780521513388).

[13] https://underscore.io/books/scala-with-cats/

[14] https://alvinalexander.com/scala/functional-programming-simplified-book

[15] http://www.lulu.com/shop/search.ep?contributorId=1600066

Functional programming simplified, by A. Alexander

Functional programming for mortals, by S. Halliday

Practical functional programming in Scala, by G. Volpe (https://leanpub.com/pfp-scala)

Mastering advanced Scala, by D. Kalinin (https://leanpub.com/mastering-advanced-scala)

Figure 16.1: Randomly chosen pages from books on Scala programming.

The only currently available Scala-based AFTT textbook is *Functional Programming in Scala*.[16] It balances practical coding with theoretical developments and laws. *Program design by calculation*[17] is another (Haskell-oriented) AFTT book in progress. The present book is written at about the same level but aims at better motivation for mathematical concepts and a wider range of pedagogical examples that help build the necessary intuition and facility with the techniques of formal derivation.

Figures 16.1–16.3 illustrate the difference between AFTT books, programming books, and academic science books, by showing randomly chosen pages from such books. Programming-oriented books contain code examples and explanations in words, but no formal derivations. Books on AFTT, as well as books on mathematics and science, will typically show equations, diagrams, and derivations. The present book contains both code examples and mathematical manipulations.

[16] https://www.manning.com/books/functional-programming-in-scala
[17] http://www4.di.uminho.pt/~jno/ps/pdbc.pdf

Linear algebra, by J. Hefferon (http://joshua.smcvt.edu/linearalgebra/)

Classical mechanics, by T. W. B. Kibble and F. H. Berkshire (https://archive.org/details/116772449ClassicalMechanics)

Figure 16.2: Randomly chosen pages from books on mathematics and physics.

Functional programming in Scala, by P. Chiusano and R. Bjarnason

Program design by calculation, by J. N. Oliveira

The science of functional programming (this book)

Figure 16.3: Randomly chosen pages from books on applied functional type theory.

17 Essay: Software engineers and software artisans

Let us examine what we ordinarily understand by *engineering* as opposed to artisanship or craftsmanship. It will then become apparent that today's computer programmers must be viewed as "software artisans" rather than software engineers.[1]

17.1 Engineering disciplines

Consider the way mechanical engineers, chemical engineers, or electrical engineers work, and look at the studies they require for becoming proficient in their fields.

A mechanical engineer studies calculus, linear algebra, differential geometry, and several areas of physics such as theoretical mechanics, thermodynamics, and elasticity theory,[2] and then uses calculations to guide the design of a bridge. A chemical engineer studies chemistry, thermodynamics, calculus, linear algebra, differential equations, some areas of physics such as thermodynamics and kinetic theory,[3] and uses calculations to guide the design of a chemical process. An electrical engineer studies advanced calculus, linear algebra, and several areas of physics such as electrodynamics and quantum theory,[4] and uses calculations to design an antenna or a microchip.

The common pattern is that engineers use mathematics and natural sciences in order to create new devices. Mathematical calculations and scientific reasoning are performed *before* designing or building a real machine.

Some of the studies required for engineers include arcane abstract concepts such as the "Fourier transform of the delta function"[5] and the "inverse Z-transform"[6] in digital signal processing, "rank 4 tensors"[7] in calculations of elasticity of materials, "Lagrangians with non-holonomic constraints"[8] in robotics, and the "Gibbs free energy" in chemical reactor design.[9]

To be sure, a significant part of what engineers do is not covered by any theory: the *know-how*, the informal reasoning, the traditional knowledge passed on from expert to novice, — all those skills that are hard to formalize are important. Nevertheless, engineering is crucially based on natural science and mathematics for some of its decision-making about new designs.

17.2 Artisanship: Trades and crafts

Now consider what kinds of things shoemakers, plumbers, or home painters do, and what they have to learn in order to become proficient in their profession.

[1] The book reviewed in https://www.developerdotstar.com/mag/bookreviews/bitner_craftsmanship.html proposes a different definition of "software artisans" than this book, but agrees that software developers work largely as artisans.

[2] https://www.colorado.edu/mechanical/academics/undergraduate-program/curriculum

[3] https://www.colorado.edu/engineering/sample-undergraduate-curriculum-chemical

[4] http://archive.is/XYLyE

[5] https://www.youtube.com/watch?v=KAbqISZ6SHQ

[6] http://archive.is/SsJqP

[7] https://serc.carleton.edu/NAGTWorkshops/mineralogy/mineral_physics/tensors.html

[8] https://arxiv.org/abs/math/0008147

[9] https://www.amazon.com/Introduction-Chemical-Engineering-Kinetics-Reactor/dp/1118368258

A novice shoemaker, for example, begins by copying some drawings[10] and goes on to cutting leather in a home workshop. Apprenticeships proceed via learning by doing, with comments and instruction from an expert. After a few years of study (for example, a painter apprenticeship in California can be as short as 2 years[11]), a new artisan is ready to start productive work.

All trades operate entirely from tradition and practical experience. It takes a prolonged learning effort to become a good artisan in any profession. But the trades do not require academic study because there is no formal theory from which to proceed. There are no Fourier transforms applied to delta functions, no Lagrangians with non-holonomic constraints, no fourth rank tensors to calculate, nor any differential equations to solve.

Artisans do not study science or mathematics because their professions do not make use of any formal theory for guiding their designs or processes.

17.3 Programmers today are artisans, not engineers

Programmers are *not engineers* in the sense normally used in the engineering professions.

17.3.1 No requirements of licensing or formal study

Mechanical, electrical, chemical engineers are required to pass a license exam to become certified to work. The license exam certifies that the person is proficient in applying a known set of engineering tools and methods. But in software engineering, no certifications or licenses are required for the job (although many certification programs exist).

Working software engineers are also not required to have studied software engineering or computer science (CS). According to a recent Stack Overflow survey,[12] about 56% of working programmers have no CS degree. The author of this book is a self-taught programmer who has degrees in physics but never formally studied CS or taken any academic courses in algorithms, data structures, computer networks, compilers, programming languages, or other standard CS topics.

A large fraction of successful programmers have no college degrees and perhaps *never* studied formally. They acquired all their knowledge and skills through self-study and practical work. A prominent example is Robert C. Martin,[13] an outspoken guru in the arts of programming. He routinely refers to programmers as artisans[14] and uses the appropriate imagery: novices and masters, trade and craft, the honor of the guild, etc. He compares programmers to plumbers, electricians, lawyers, and surgeons, but never to mathematicians, physicists, or engineers of any kind. According to one of his blog posts,[15] he started working at age 17 as a self-taught programmer. He never went to college and holds no degrees.[16] It is clear that R. C. Martin *is* an expert craftsman and that he did *not* need academic study to master the craft of programming.

Here is another opinion[17] (emphasis is theirs):

> Software Engineering is unique among the STEM careers in that it absolutely does *not* require a college degree to be successful. It most certainly does not require licensing or certification. *It requires experience*.

This description fits a career in crafts — but certainly not a career, say, in electrical engineering.

The high demand for software developers gave rise to "developer boot camps"[18] — vocational schools that educate new programmers in a few months through purely practical training, with no

[10] https://youtu.be/cY5MY0czMAk?t=141
[11] http://www.calapprenticeship.org/programs/painter_apprenticeship.php
[12] https://thenextweb.com/insider/2016/04/23/dont-need-go-college-anymore-programmer/
[13] https://en.wikipedia.org/wiki/Robert_C._Martin
[14] https://blog.cleancoder.com/uncle-bob/2013/02/01/The-Humble-Craftsman.html
[15] https://blog.cleancoder.com/uncle-bob/2013/11/25/Novices-Coda.html
[16] https://hashnode.com/post/i-am-robert-c-martin-uncle-bob-ask-me-anything-cjr7pnh8g000k2cs18o5nhulp/2
[17] http://archive.is/tAKQ3
[18] http://archive.is/GkOL9

formal theory or mathematics involved. These vocational schools are successful[19] in job placement. But it is unimaginable that a 6-month crash course or even a 2-year vocational school could prepare engineers to work successfully on designing, say, analog quantum computers[20] without ever learning quantum physics or calculus.

17.3.2 No mathematical formalism guides software development

Most books on software engineering contain no formulas or equations, no mathematical derivations, and no precise definitions of the various technical terms they are using (such as "object-oriented" or "module's responsibilities"). Some of those books[21] also have almost no program code in them. Some of those books are written by practitioners such as R. C. Martin never studied any formalisms and do not think in terms of formalisms. Instead, they summarize their programming experience in vaguely formulated heuristic "principles".[22] The programmers are told: "code is about detail", "never abandon the big picture", "avoid tight coupling in your modules", "a class must serve a single responsibility", "strive for good interfaces", etc.

In contrast, textbooks on mechanical or electrical engineering include a significant amount of mathematics. The design of a microwave antenna is guided not by an "open and closed module principle" but by solving the relevant differential equations[23] of electrodynamics.

Another example of programmers' avoidance of mathematical tools is given by the "Liskov substitution principle" for subtyping.[24] Its rigorous formulation ("for any property $\phi(x)$ that holds for all x of type T, and for any subtype S of T, the property $\phi(y)$ must also hold for all y of type S") is not used by programmers. Instead, the literature on object-oriented programming formulates the principle as "objects of type T may be substituted by objects of type S while keeping the correctness of the program". This formulation is both vague (it does not specify how to choose the substituted objects of type S) and, strictly speaking, incorrect: If the program consists of a function $f(t)$ where t is a value of type T, it is not always possible to find some value s of type S such that $f(s) = f(t)$. The reason is that some subtyping relations are not surjective, as shown in Section 6.4.2 of this book.

Donald Knuth's classic textbook "*The Art of Programming*" indeed treats programming as an art and not as a science. Knuth shows many algorithms and derives their mathematical properties but gives almost no examples of realistic program code and does not provide any theory that could guide programmers in actually *writing* programs (say, choosing the data types to be used). Knuth assumes that the reader who understands the mathematical properties of an algorithm will be able *somehow* to write correct code.

The books "The Science of Programming"[25] and "Program derivation"[26] are attempts to provide a mathematical basis for writing programs starting from formal specifications. The books give some methods that guide programmers in writing code and at the same time produce a proof that the code conforms to the specification. However, the scope of proposed methods is limited to designing algorithms for iterative manipulation of data, such as sorting and searching algorithms. The procedures suggested in those books are far from a formal mathematical *derivation* of a wide range of software programs from specifications. In any case, most programmers today are unaware of these books and do not use the methods explained there, even when those methods could apply.

Today's computer science courses do not teach a true engineering approach to software construction. Some courses teach analysis of programs using mathematical methods. Two such methods are

[19] http://archive.is/E9FXP

[20] https://www.dwavesys.com/quantum-computing

[21] E.g., https://www.amazon.com/Object-Oriented-Software-Engineering-Unified-Methodology/dp/0073376256

[22] https://blog.cleancoder.com/uncle-bob/2016/03/19/GivingUpOnTDD.html

[23] https://youtu.be/8KpfVsJ5Jw4?t=447

[24] See https://en.wikipedia.org/wiki/Liskov_substitution_principle. The LSP always holds in functional programming if values are immutable and subtyping is viewed as an automatic type conversion function (see Definition 6.1.8.1). A property $\phi(y)$ is rewritten as $\phi(c(y))$ by inserting a suitable type conversion function, $c : S \to T$. Since $c(y)$ has type T and $\phi(x)$ holds for all values $x : T$, we find that the property $\phi(c(y))$ holds automatically.

[25] https://www.amazon.com/Science-Programming-Monographs-Computer/dp/0387964800

[26] https://www.amazon.com/Program-Derivation-Development-Specifications-International/dp/0201416247

complexity analysis[27] (the "big-O notation") and formal verification.[28] But programs are analyzed or verified only *after* they are somehow written. Theory does not guide the actual *process* of writing code: it does not define good ways of organizing the code (e.g., how to decompose the code into modules, classes, or functions) and does not tell programmers which data structures and type signatures of functions will be useful to implement. Programmers make these design decisions on the basis of experience and intuition, trial-and-error, copy-paste, guesswork, and debugging.

In a sense, program analysis and verification is analogous to writing mathematical equations for the surface of a shoe made by a fashion designer. The resulting "shoe equations" are mathematically rigorous and can be analyzed or "verified". But the equations are merely written after the fact, they do not guide the fashion designers in actually making shoes. It is understandable that fashion designers do not study the mathematical theory of surfaces.

17.3.3 Programmers avoid academic terminology

Programmers jokingly grumble about terms such as "functor", "monad", or "lambda-functions":

> Those fancy words used by functional programmers purists really annoy me. Monads, functors... Nonsense!!! [29]

Perhaps only a small minority of software developers complain about this, as the majority seems to be unaware of "applicative functors", "free monads", and other arcane terminology. Indeed, that sort of terminology is intentionally avoided by most books and tutorials aimed at programmers.

But why would a software *engineer* wince at "functors" or at having to verify the laws of a "monad"? Other branches of engineering use lots of terminology that is far from self-explanatory and requires some study. Chemical engineers learn about "Gibbs free energy", which is a technical term that denotes a certain function. (It does not mean getting energy from J. W. Gibbs for free.) Chemical engineers accept the need for studying "phase diagrams" or "Fourier's law". Electrical engineers do not avoid "Fourier transforms" or "delta functions" because those are weird things to say. Mechanical engineers take it for granted that they need "rank 4 tensors", "Lagrangians", and "non-holonomic constraints". The arcane terminology seems to be the least of their difficulties, as their textbooks are full of complicated equations and long derivations.

Textbooks on true software engineering would have been full of equations and derivations, teaching engineers how to perform certain calculations that are required *before* starting to write code.

17.4 Towards true engineering in software

It is now clear that we do not presently have true software engineering. The people employed under that job title are actually artisans. They work using artisanal methods, and their culture and processes are that of a crafts guild.

True software engineering means having a mathematical theory that guides the process of writing programs, — not just theory that describes or analyzes programs after they are *somehow* written.

It is not enough that the numerical methods required for physics or the matrix calculations required for data science are "mathematical". These programming tasks are indeed formulated using mathematical theory. However, mathematical *subject matter* (aerospace control, physics or astronomy simulations, or statistics) does not mean that mathematics is used to guide the process of writing code. Data scientists, aerospace engineers, and physicists almost always work as artisans when converting their computations into program code.

We expect that software engineers' textbooks should be full of equations and derivations. What theory would those equations represent?

[27] https://www.cs.cmu.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html
[28] https://en.wikipedia.org/wiki/Formal_verification
[29] http://archive.is/65K3D

This theory is what this book calls **applied functional type theory** (see Chapter 16). It represents the mathematical foundation of the modern practice of functional programming, as implemented in languages such as OCaml, Haskell, and Scala. This theory is a blend of set theory, category theory, and logical proof theory, adapted for the needs of programmers. It has been in development since late 1990s and is still being actively worked on by a community of software practitioners and academic computer scientists.

To appreciate that functional programming, unlike any other programming paradigm, is based on a *theory that guides coding*, we can look at some recent software engineering conferences such as "Scala By the Bay"[30] or BayHac,[31] or at the numerous FP-related online tutorials and blogs. We cannot fail to notice that speakers devote significant time to a peculiar kind of applied mathematical reasoning. Rather than focusing on one or another API or algorithm, as it is often the case with other software engineering blogs or presentations, an FP speaker describes a *mathematical structure* — such as the "applicative functor"[32] or the "free monad"[33] — and illustrates its use for practical coding.

These people are not graduate students showing off their theoretical research. They are practitioners, software engineers who use FP on their jobs. It is just the nature of FP that certain mathematical tools and constructions are directly applicable to practical programming tasks.

These mathematical tools are not mere tricks for a specific programming language; they apply equally to all FP languages. Before starting to write code, the programmer can jot down certain calculations in a mathematical notation (see Fig. 17.1). The results of those calculations will help design the code fragment the programmer is about to write. This activity is similar to that of an engineer who performs some mathematical calculations before embarking on a design project.

Figure 17.1: A programmer performs a derivation before writing Haskell code.

A recent example of a development in applied functional type theory is the "free applicative functor" construction. It was first described in a 2014 paper;[34] a couple of years later, a combined free applicative / free monad data type was designed and its implementation proposed in Scala[35] as well as in Haskell.[36] This technique allows programmers to implement declarative side-effect computations where some parts are sequential but other parts are computed in parallel, and to achieve the parallelism *automatically* while maintaining the composability of the resulting programs. The new technique has advantages over monad transformers, which was a previously known method of composing declarative side-effects. The combined "free applicative / free monad" code was designed and implemented by true software engineers. They first derived the type constructor that has the necessary algebraic properties. Guided by the resulting type formula, they wrote code that was guaranteed to work as intended.

Another example of using applied functional type theory is the "tagless final" encoding of effects, first described[37] in 2009. That technique (called "Church-encoded free monad" in the present book) has advantages over the ordinary free monad in a number of cases — just as the free monad itself

[30] http://2015.scala.bythebay.io/
[31] http://bayhac.org/
[32] http://www.youtube.com/watch?v=bmIxIslimVY
[33] http://www.youtube.com/watch?v=U0lK0hnbc4U
[34] https://arxiv.org/pdf/1403.0749.pdf
[35] https://github.com/typelevel/cats/issues/983
[36] https://elvishjerricco.github.io/2016/04/08/applicative-effects-in-free-monads.html
[37] http://okmij.org/ftp/tagless-final/index.html

was used to cure certain problems with monad transformers.[38] The new encoding is not tied to a specific programming language. Rather, it is a language-agnostic construction that was originally described in OCaml and later used in Haskell and Scala, but can be made to work even in Java,[39] which is not an FP language.

This example shows that we may need several more years of work before the practical aspects of using applied functional type theory are sufficiently well understood by the FP community. The theory is in active development, and its design patterns — as well as the exact scope of the requisite theoretical material — are still being figured out. If the 40-year gap hypothesis[40] holds, we should expect applied functional type theory (perhaps under a different name) to become mainstream by 2030. This book is a step towards a clear designation of the scope of that theory.

17.5 Does software need engineers, or are artisans sufficient?

The demand for programmers is growing. "Software developer" was #1 best job[41] in the US in 2018. But is there a demand for engineers or just for artisans?

We do not seem to be able to train enough software artisans.[42] So, it is probably impossible to train as many software engineers in the true sense of the word. Modern computer science courses do not actually train engineers in that sense. Instead, they train academic researchers who can also work as software artisans and write code.

Looking at the situation in construction business in the U.S.A., we find that it employs about 10 times more construction workers as architects. We might conclude that perhaps one software engineer is required per dozen software artisans.

What is the price of *not* having engineers, of replacing them with artisans?

Software practitioners have long bemoaned the permanent state of "crisis" in software development. Code "rots with time", its complexity grows "out of control", and operating systems have been notorious for ever-appearing security flaws[43] despite many thousands of programmers and testers employed. It appears that the growing complexity of software tends to overwhelm the capacity of the human brain for correct *artisanal* programming.

It is precisely in designing large and robust software systems that we would benefit from true engineering. Artisans has been building bridges and using chemical reactions by trial and error and via tradition, long before mechanical or chemical engineering disciplines were developed and founded upon rigorous theory. But once the theory became available, engineers were able to design unimaginably more powerful and complicated structures, devices, and processes. So, we may expect that trial, error, and adherence to tradition is inadequate for some of the more complex software development tasks in front of us.

To build large and reliable software, such as new mobile or embedded operating systems or distributed peer-to-peer trust architectures, we will most likely need the qualitative increase in productivity and reliability that can only come from replacing artisanal programming by a true engineering discipline. Applied functional type theory and functional programming are steps in that direction.

[38] http://blog.ezyang.com/2013/09/if-youre-using-lift-youre-doing-it-wrong-probably/
[39] https://oleksandrmanzyuk.wordpress.com/2014/06/18/
[40] https://www.linkedin.com/pulse/40-year-gap-what-has-academic-computer-science-ever-done-winitzki
[41] http://money.usnews.com/money/careers/articles/how-us-news-ranks-the-best-jobs
[42] http://archive.is/137b8
[43] http://archive.fo/HtQzw

18 Essay: Towards functional data engineering with Scala

Data engineering is among the highest-demand[1] novel occupations in the IT world today. Data engineers create software pipelines that process large volumes of data efficiently. Why did the Scala programming language emerge as a premier tool[2] for crafting the foundational data engineering technologies such as Spark or Akka? Why is Scala in high demand[3] within the world of big data?

There are reasons to believe that the choice of Scala was not accidental.

18.1 Data is math

Humanity has been working with data at least since Babylonian tax tables[4] and the ancient Chinese number books.[5] Mathematics summarizes several millennia's worth of data processing experience in a few fundamental tenets:

- Data is *immutable*, because facts are immutable.
- Each *type* of values (population count, land area, distance, price, location, time, etc.) needs to be handled separately; it is meaningless to add a distance to a population count.
- Data processing should be performed according to *mathematical formulas*.

Violating these tenets produces nonsense (see Fig. 18.1 for a real-life illustration).

The power of the principles of mathematics extends over all epochs and all cultures; math is the same in San Francisco, in Rio de Janeiro, in Kuala-Lumpur, and in Pyongyang (Fig. 18.2).

Figure 18.1: Mixing incompatible data types produces nonsensical results.

18.2 Functional programming is math

The functional programming paradigm is based on mathematical principles: values are immutable, data processing is coded through formula-like expressions, and each type of data is required to match correctly during the computations. The type-checking process automatically prevents programmers from making many kinds of coding errors. In addition, programming languages such as Scala and Haskell have a set of features adapted to building powerful abstractions and domain-specific languages. This power of abstraction is not accidental. Since mathematics is the ultimate art of building abstractions, math-based functional programming languages capitalize on having several millennia of mathematical experience.

[1] http://archive.is/mK59h
[2] https://www.slideshare.net/noootsab/scala-the-unpredicted-lingua-franca-for-data-science
[3] https://techcrunch.com/2016/06/14/scala-is-the-new-golden-child/
[4] https://www.nytimes.com/2017/08/29/science/trigonometry-babylonian-tablet.html
[5] https://quatr.us/china/science/chinamath.htm

A prominent example of how mathematics informs the design of programming languages is the connection between constructive logic[6] and the programming language's type system, called the Curry-Howard (CH) correspondence. The main idea of the CH correspondence is to think of programs as mathematical formulas that compute a value of a certain type A. The CH correspondence is between programs and logical propositions: To any program that computes a value of type A, there corresponds a proposition stating that "a value of type A can be computed".

This may sound rather theoretical so far. To see the real value of the CH correspondence, recall that formal logic has operations "***and***", "***or***", and "***implies***". For any two propositions A, B, we can construct the propositions "A ***and*** B", "A ***or*** B", "A ***implies*** B". These three logical operations are foundational; without one of them, the logic is *incomplete* (you cannot derive some theorems).

A programming language **obeys the CH correspondence** with the logic if for any types A, B, the language also contains composite types corresponding to the logical formulas "A ***or*** B", "A ***and*** B", "A ***implies*** B". In Scala, these composite types are `Either[A, B]`, the tuple `(A,B)`, and the function type `A => B`. All modern functional languages such as OCaml, Haskell, Scala, F#, Swift, Elm, and PureScript support these three type constructions and thus obey the CH correspondence. Having a *complete* logic in a language's type system enables declarative domain-driven code design.[7]

Figure 18.2: The Pyongyang method of error-free software engineering.

It is interesting to note that most older programming languages (C/C++, Java, JavaScript, Python) do not support some of these composite types. In other words, these programming languages have type systems based on an incomplete logic. As a result, users of these languages have to implement burdensome workarounds that make for error-prone code. Failure to follow mathematical principles has real costs (Figure 18.2).

18.3 The power of abstraction

Early adopters of Scala, such as Netflix, LinkedIn, and Twitter, were implementing what is now called "big data engineering". The required software needs to be highly concurrent, distributed, and resilient to failure. Those software companies used Scala as their main implementation language and reaped the benefits of functional programming.

What makes Scala suitable for big data tasks? The only reliable way of managing massively concurrent code is to use sufficiently high-level abstractions that make application code declarative. The two most important such abstractions are the "resilient distributed dataset" (RDD) of Apache Spark and the "reactive stream" used in systems such as Kafka, Akka Streams, and Apache Flink. While these abstractions are certainly implementable in Java or Python, a fully declarative and type-safe usage is possible only in a programming language with a sophisticated type system. Among the currently available mature functional languages, only Scala and Haskell are technically adequate for that task, due to their support for typeclasses and higher-order types. The early adopters of Scala were able to benefit from the powerful abstractions Scala supports. In this way, Scala enabled those businesses to engineer and to scale up their massively concurrent computations.

It remains to see why Scala (and not, say, OCaml or Haskell) became the *lingua franca* of big data.

[6] https://en.wikipedia.org/wiki/Intuitionistic_logic
[7] https://fsharpforfunandprofit.com/ddd/

18.4 Scala is Java on math

The recently invented general-purpose functional programming languages may be divided into "academic" (OCaml, Haskell) and "industrial" (F#, Scala, Swift).

The "academic" languages are clean-room implementations of well-researched mathematical principles of programming language design (the CH correspondence being one such principle). These languages are not limited by requirements of compatibility with any existing platforms or libraries. Because of this, the "academic" languages have been designed and used for pursuing various mathematical ideas to their logical conclusion.[8] At the same time, software practitioners struggle to adopt these programming languages due to a steep learning curve, a lack of enterprise-grade libraries and tool support, and immature package management.

The languages from the "industrial" group are based on existing and mature software ecosystems: F# on .NET, Scala on JVM, and Swift on the MacOS/iOS platform. One of the important design requirements for these languages is 100% binary compatibility with their "parent" platforms and languages (F# with C#, Scala with Java, and Swift with Objective-C). Because of this, developers can immediately take advantage of the existing tooling, package management, and industry-strength libraries, while slowly ramping up the idiomatic usage of new language features. However, the same compatibility requirements dictate certain limitations in the languages, making their design less than fully satisfactory from the functional programming viewpoint.

It is now easy to see why the adoption rate of the "industrial" group of languages is much higher[9] than that of the "academic" languages. The transition to the functional paradigm is also smoother for software developers because F#, Scala, and Swift seamlessly support the familiar object-oriented programming paradigm. At the same time, these new languages still have logically complete type systems, which gives developers an important benefit of type-safe domain modeling.

Nevertheless, the type systems of these languages are not equally powerful. For instance, F# and Swift are similar to OCaml in many ways but omit OCaml's parameterized modules and some other features. Of all mentioned languages, only Scala and Haskell directly support typeclasses and higher-order functions on types, which are helpful for expressing abstractions such as automatically parallelized data sets or asynchronous data streams.

To see the impact of these advanced features, consider LINQ, a domain-specific language for database queries on .NET, implemented in C# and F# through a special built-in syntax supported by Microsoft's compilers. Analogous functionality is provided in Scala as a *library*, without need to modify the Scala compiler, by several open-source projects such as Slick and Quill. Similar libraries exist for Haskell — but not in languages with less powerful type systems.

18.5 Summary

Only Scala has all of the features required for industrial-grade functional programming:

1. Functional collections in the standard library.
2. A sophisticated type system with support for typeclasses and higher-order types.
3. Seamless compatibility with a mature software ecosystem (JVM).

Based on this assessment, we may be confident in Scala's future as a main implementation language for big data engineering.

[8] OCaml has arbitrary recursive product and co-product types that can be freely combined with object-oriented types. Haskell removes all side effects from the language and supports partial type functions of arbitrarily high order.

[9] `https://www.tiobe.com/tiobe-index/`, archived in 2019 at `http://archive.is/RsNH8`

19 Essay: Why category theory is useful in functional programming

This essay is for readers who are already somewhat familiar with category theory.

19.1 A "types/functions" category for a programming language

We consider programming languages that support various data types, such as integers (`Int`), floating-point numbers (`Float`), strings (`String`), arrays of strings (`Array[String]`), and so on. Such languages allow programmers to define functions with specified types of arguments and return values. The compiler will then verify that all functions are always applied to arguments of correct types, and also that all return values have the expected types.

To each programming language of that kind, there corresponds a "types/functions" category:

- The objects of the category are all the data types supported by the language (including user-defined data types). As an example, for Scala there will be an object `Int`, an object `Float`, an object `String`, an object `Array[String]`, and so on.
- The morphisms between objects `A` and `B` are all functions implementable in the language (with finitely long program code) that take a single argument of type `A` and return a value of type `B`.
- We assume that the computer has countably infinite memory, so objects can be viewed as (at most) countably infinite sets. Morphisms will also form at most countably infinite sets.

The category defined in this way will typically have a large number of morphisms between most objects. For example, morphisms between objects `Boolean` and `Int` are functions that take a single argument of type `Boolean` and return a value of type `Int`. There are as many such functions as pairs of integers. Scala code for one of those morphisms looks like this:

```
def morphismBooleanToInt: Boolean => Int = { b => if (b) 123 else 456 }
```

Why do the category laws hold? The composition of morphisms corresponds to composition of functions, which we can implement by writing code that applies the first function and then applies the second function. In Scala:

```
def composeMorphisms[A, B, C](f: A => B, g: B => C): A => C   =   { a => g(f(a)) }
```

Equivalent functionality can be implemented in most programming languages.

The category's identity law says that there must be a morphism between objects `A` and `A`. This can be implemented in most programming languages as a function that returns its argument unchanged:

```
def identity[A]: A => A = { x => x }
```

One can check that morphism composition is associative and agrees with the identity morphism.

For a given programming language, we have thus defined the "types/functions category", which can be seen as a subcategory of the category of sets. Most of the time, we will be working with that category, or with the category of its endofunctors, or with a sub-category of these categories.

Different programming languages will give rise to different "types/functions" categories, but all those categories have many common features that are especially important in languages designed for functional programming (the "FP languages", such as OCaml, Haskell, Scala and others).

19.2 The use of endofunctors

An endofunctor in the "types/functions" category is a mapping of types together with a mapping of functions. A good example is the `Array` data type. In some programming languages, the type of an array's elements can be specified and enforced throughout the code. For example, in Scala one can use the type `Array[Int]` for an array of integers, `Array[String]` for an array of strings, `Array[Array[Int]]` for an array containing nested arrays of integers, etc. So, `Array` can be seen as a mapping from types to types: it maps the type `Int` to the type `Array[Int]`, the type `String` to the type `Array[String]`, etc. For any type `X`, we have the type `Array[X]`. This is the object-to-object map of an endofunctor.

An endofunctor also needs a map from morphisms to morphisms. Given a function `f: X => Y`, we need to implement a function of type `Array[X] => Array[Y]`. This can be done by writing a loop over the array and applying the function `f` to each element (of type `X`). The resulting values (of type `Y`) are then collected in a new array, of type `Array[Y]`.

This code can be written in many programming languages in a generic manner, using type parameters such as `X` and `Y`. The same code will then work for arrays and functions of any given type. In Scala, the code could be written as the following function (usually called `fmap` in FP libraries):

```
def fmap[X, Y: ClassTag](f: X => Y): Array[X] => Array[Y] = { arrayX: Array[X] =>
  val arrayY = new Array[Y](arrayX.size)
  for { i <- arrayX.indices } arrayY(i) = f(arrayX(i))
  arrayY  // Return this array of type Array[Y].
}
```

One can then check that the code of `fmap` satisfies the identity and composition laws of endofunctors. This completes the implementation of the `Array` endofunctor.

Why does `fmap` satisfy the laws of endofunctors? The categorical properties of functions are preserved if we apply functions to each element of an array and collect the results *in the same order*. An identity function applied to every element will not modify the array. Function composition is preserved because a composition of two functions will be applied separately to each array element.

The same construction can be applied to many data structures other than arrays. It turns out that many programs can be reformulated using the operation of applying a function to every value in a data structure (i.e., the function `fmap`). This reformulation leads to code that avoids loops: the loops are replaced by `fmap` functions of some endofunctors, and all those `fmap` functions are implemented in a standard library. In practice, code written via `fmap` instead of loops is more concise and admits fewer opportunities for errors. The programmer's intuition about "applying functions to every value held within a data structure" is then directly represented by the formal laws of endofunctors. Once those laws are verified, the programmer is assured that the code written via `fmap` will work according to the programmer's intuitive expectations.

19.3 The use of natural transformations

What is a natural transformation between endofunctors in the "types/functions" category? For two given endofunctors `F` and `G`, a natural transformation `t: F ~> G` is defined by its components. The component at object `X` is a function of type `F[X] => G[X]`; this must be defined for all `X`. Some programming languages support functions with type parameters. In Scala, the syntax is

```
def t[X]: F[X] => G[X] = ...
```

The code of such a function is written once and will work in the same way for all types `X`.

An example of natural transformation is a function that reverses the order of elements in an array:

```
def reverse[X]: Array[X] => Array[X] = ...
```

The algorithm is "fully parametric": it is written in the same way for all type parameters `X`.

It turns out that, by the Reynolds-Wadler parametricity theorem, any code written in a fully parametric manner will satisfy the law of a natural transformation (the naturality law). The naturality

law states that applying the endofunctor `F`'s morphism map before a natural transformation `t` must be equal to applying the endofunctor `G`'s map after `t`. In Scala syntax, the law is written as

```
t(fmap_F(f)(x)) == fmap_G(f)(t(x))
```

This law can be verified directly for a given code of `t` and with known code for `fmap_F` and `fmap_G`.

Naturality laws are satisfied by transformations that rearrange data items in a data structure in some way that does not depend on specific values or types. In this way, the formal laws of natural transformations directly represent programmers' intuitions about code that works "in the same way for all type parameters".

As we have just seen, the notions of endofunctors and natural transformations are useful in programming languages that support types with type parameters (such as `Array[X]`) and functions with type parameters (such as `reverse[X]`). Programming languages that do not support those features cannot benefit from the powerful reasoning tools of category theory.

19.4 Other properties of the "types/functions" category

Morphisms in the "types/functions" category are always functions of a single argument. However, programming languages usually support functions with many arguments. There are two ways to imitate such functions: tupling and currying.

Tupling means that we put all arguments into a compound structure (a pair, a triple, etc.). The function is still viewed as having a single argument, but the type of that argument is the type of a pair, or a triple, or a longer tuple type. This works when the programming language supports tuple types. A tupled function's type is written (in Scala) as, e.g., `((A, B, C)) => D`.

Tuple types correspond to finite products of objects in the "types/functions" category. So, it is useful if the category has (finite) products.

Currying means that we create a function that takes the first argument and returns a curried function that handles the rest of the arguments in the same way (takes the second argument and again returns a function, etc.). A curried function's type is written in Scala as, e.g., `A => B => C => D`. To support this method, the programming language should have function types. The corresponding categorical construction is the "exponential" object.

In the practice of functional programming, it has been found useful to have the type `Unit`, which has exactly one value, and the type `Nothing`, which has no values. In the "types/functions" category, these types correspond to the terminal and the initial objects.

Finally, disjunctive types correspond to co-products in the "types/functions" category.

In this way, we find that various well-known mathematical properties of the "types/functions" category (initial and terminal objects, finite products and co-products, exponentials) correspond to properties of the programming language that proved useful in the practice of software engineering.

19.5 Some useful sub-categories of endofunctors

Besides loops that apply functions to array elements, other frequently used computations are nested loops and "while"-loops that are repeated while a given condition holds and then stopped. It turns out that category theory provides a convenient language for reasoning about such computations. Similarly to representing loops via endofunctors, the various kinds of loops are encoded via certain sub-categories of endofunctors in the "types/functions" category.

To see how this works, we need to define an auxiliary sub-category called the "`F`-lifted" (where `F` may be any given endofunctor, such as `Array`). The `F`-lifted sub-category is the image of the endofunctor `F`. The objects of that sub-category are types of the form `F[A]`. The morphisms of that sub-category are functions of type `F[A] => F[B]` (not necessarily obtained by lifting some function `f: A => B` through the endofunctor `F`'s `fmap` method).

19.5.1 Filterable endofunctors

To describe "while"-loops using category theory, we begin by reformulating the loop as a *mathematical function* rather than as a sequence of computer instructions. To be specific, consider a program with a loop that stops when a certain condition first becomes `false`. A loop of this kind may be modeled by a function that takes an initial array as argument and returns a new array that is truncated when a given condition first becomes `false`. The condition is represented by a function evaluated on each element of the array. The Scala standard library includes such a function (`takeWhile`). An example of its usage is:

```
scala> Array(1, 2, 3, 4, 5).takeWhile(x => x < 4)    // Truncate the array when x ≥ 4 is first found.
res0: Array[Int] = Array(1, 2, 3)
```

The next step is to extend this function to work with arbitrary types instead of integers. The type signature of `takeWhile` may be written as

```
def takeWhile[X](p: X => Boolean): Array[X] => Array[X]
```

Here `X` is a type parameter. The first argument is a predicate of type `X => Boolean`.

Finally, we write the laws that we expect this function to satisfy. For instance, if the given predicate `p` always returns `true`, the function should not change the given array (the "identity law"):

```
takeWhile(x => true)  ==  identity
```

Another plausible law is called the "composition law". If we apply `takeWhile` with a predicate `p1` and then again apply `takeWhile` with another predicate `p2`, the resulting truncated array should be the same as if we applied `takeWhile` just once with a Boolean conjunction of `p1` and `p2`:

```
takeWhile(p1) andThen takeWhile(p2) == takeWhile( x => p1(x) && p2(x) )
```

The identity and composition laws of `takeWhile` are analogous to the identity and composition laws of functors. More precisely, one can derive the laws of `takeWhile` from the laws of an auxiliary functor between a certain Kleisli category and the `F`-lifted category (see Example 9.4.3.4 for details). That auxiliary functor exists only for some endofunctors `F`, which are called "filterable" in this book. Filterable endofunctors are a sub-category of all endofunctors of the "types/functions" category.

With this construction, one may now regard the laws of `takeWhile` not as arbitrarily postulated properties but as a consequence of the functor laws. In this way, category theory validates the programmer's intuition for the choice of the laws for `takeWhile`.

19.5.2 Monadic endofunctors

To evaluate a nested loop, which may be written in Scala as, e.g.,

```
for {
  x <- 1 to 10
  y <- 1 to x / 2
} yield f(x, y) // Some computation that may use x and y.
```

the computer will perform ten repetitions of the inner loop over `y`. This computation is equivalent to converting the nested loop into an ordinary, "flattened" loop that has a larger total number of repetitions (in this example, 25 repetitions). To describe this situation using category theory, we start by reformulating a nested loop into a mathematical function. The arguments of that function are the first array (`1 to 10`) for iterating with `x`, and a function from a value of `x` to the nested array (`x => 1 to x / 2`). The function returns a "flattened" array of 25 values.

The Scala library contains such a function, named `flatMap`. An example of usage is:

```
scala> (1 to 10).toArray.flatMap(x => 1 to x / 2)
res0: Array[Int] = Array(1, 1, 1, 2, 1, 2, 1, 2, 3, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 5)
```

This function can be used repeatedly to convert arbitrarily deeply nested loops into "flat" loops.

The next step is to formulate a fully parametric type signature for `flatMap`:

```
def flatMap[A, B](f: A => Array[B]): Array[A] => Array[B]
```

In this way, `flatMap` can transform arrays with elements of any type.

The `flatMap` function must satisfy certain properties that are useful for practical programming. One of these properties is "associativity". A deeply nested loop may be flattened by applying `flatMap` first to the outer layers and then to the inner layers, or by applying flatMap first to the inner layers; the results must be the same. This and other properties of `flatMap` are analogous to the laws of a category: there are two identity laws and one associativity law. More precisely, one can derive the laws of `flatMap` from the requirement that the Kleisli category on the endofunctor `Array` is well-defined (see Section 10.3.4 for details). This is equivalent to saying that `Array` is a monad. Monads form a sub-category of endofunctors of the "types/functions" category.

19.6 Category theory and the laws of FP idioms

We have seen that "while"-loops and nested loops can be reformulated through type-parameterized functions satisfying certain laws. Those laws are then equivalent to the laws of suitably chosen functors or categories. This turns out to be a general pattern:

- Begin with a known idiom of computation (e.g., a certain kind of a loop).
- Reformulate that idiom through functions with parameterized argument types.
- Write the laws that programmers expect those functions to satisfy.
- Prove that those laws are equivalent to the laws of a suitable functor and/or category.

The derivations in Chapters 6–14 of this book follow this pattern. One can show that this pattern holds for at least *eleven* sub-categories of endofunctors used in FP practice: functors, contrafunctors, filterable functors, filterable contrafunctors, applicative functors, applicative contrafunctors, monads, comonads, traversable functors, monad transformers, and comonad transformers.

It appears that category theory and its basic tools (functors, natural transformations, commutative diagrams) provide a powerful and versatile language for reasoning about laws of various FP idioms. By invoking category theory, programmers avoid having to memorize a large number of laws and constructions. Without the underlying categorical justification, the laws for different endofunctors will appear to be chosen arbitrarily, with no clearly recognizable system or pattern.

In addition, category theory guides programmers in creating highly abstract libraries that work uniformly with all endofunctors of a certain sub-category. In programmer's terms, such libraries contain functions parameterized by type constructors satisfying appropriate constraints. Examples are functions that define the product or the co-product of any two given functors, or define a free monad on a given functor. Implementing libraries of that kind requires formulating and verifying the relevant laws. Category theory is a reliable foundation for such libraries.

Part V

Appendixes

A Notations

This book chooses certain notations differently from what the functional programming community currently uses. The proposed notation is well adapted to reasoning about types and code, and especially for designing data types and proving the laws of various typeclasses.

A.1 Summary of notations

A — type parameter. Names of type parameters are always capitalized.

F^A — type constructor F with type argument A. In Scala, `F[A]`

$F^\bullet$ — the type constructor F understood as a type-level function. In Scala, `F[_]`

$x^{:A}$ — value x has type A; in Scala, `x:A`. Value names are always in lowercase.

$\mathbb{1}$, 1 — the unit type and its value; in Scala, `Unit` and `()`

$\mathbb{0}$ — the void type. In Scala, `Nothing`

$A + B$ — a disjunctive type (co-product). In Scala, this type is `Either[A, B]`

$x^{:A} + \mathbb{0}^{:B}$ — a value of a disjunctive type $A + B$. In Scala, `Left(x)`

$A \times B$ — a product (tuple) type. In Scala, this type is `(A,B)`

$a^{:A} \times b^{:B}$ value of a tuple type $A \times B$. In Scala, `(a, b)`

$A \to B$ — the function type, mapping from A to B

$x^{:A} \to f$ — a nameless function (as a value). In Scala, `{ x:A => f }`

id — an identity function; in Scala, `identity[A]`

$\stackrel{\text{def}}{=}$ — "is defined to be" or "is equal by definition"

$\stackrel{!}{=}$ — "must be equal according to what we know"

$\stackrel{?}{=}$ — "we ask — is it equal? — because we still need to prove that"

$\square$ — "this proof or this derivation or this definition or this example is finished"

$\cong$ — for types, a natural isomorphism between types; for values, "equivalent" values according to an already established isomorphism

$A^{:F^B}$ — special type annotation, used for defining unfunctors (GADTs)

$\wedge$ — logical conjunction; $\alpha \wedge \beta$ means "both α and β are true"

$\vee$ — logical disjunction; $\alpha \vee \beta$ means "either α or β or both are true"

$\Rightarrow$ — logical implication; $\alpha \Rightarrow \beta$ means "if α is true then β is true"

$\textbf{fmap}_F$ — the standard method `fmap` of a functor F. In Scala, `Functor[F].fmap`

$\mathbf{flm}_F, \mathbf{ftn}_F, \mathbf{pu}_F$ — the standard methods `flatMap`, `flatten`, and `pure` of a monad F

$F^\bullet$ — the type constructor F understood as a type-level function. In Scala, `F[_]`

$F^\bullet \rightsquigarrow G^\bullet$ — or $F \rightsquigarrow G$ a natural transformation between functors F and G. In Scala, `F ~> G`

$\forall A.\, P^A$ — a universally quantified type expression. In Scala 3, `[A] => P[A]`

$\exists A.\, P^A$ — an existentially quantified type expression. In Scala, `{ type A; val x: P[A] }`

$f \fatsemi g$ — the forward composition of functions: $f \fatsemi g$ is $x \to g(f(x))$. In Scala, `f andThen g`

$f \circ g$ — the backward composition of functions: $f \circ g$ is $x \to f(g(x))$. In Scala, `f compose g`

$F \circ G$ — the backward composition of type constructors: $F \circ G$ is $F^{G^\bullet}$. In Scala, `F[G[A]]`

$\triangleright$ — use a value as the argument of a function: $x \triangleright f$ is $f(x)$. In Scala, `x.pipe(f)`

$f^{\uparrow G}$ — a function f lifted to a functor G; same as $\text{fmap}_G(f)$

$f^{\uparrow G \uparrow H}$ — a function lifted first to G and then to H. In Scala, `h.map(_.map(f))`

$f^{\downarrow H}$ — a function f lifted to a contrafunctor H

$\diamond_M$ — the Kleisli product operation for the monad M

$L \not\triangleleft M$ or equivalently T_L^M — the monad L's transformer applied to a monad M

$\oplus$ — the binary operation of a monoid. In Scala, `x |+| y`

Δ — the "diagonal" function of type $\forall A.\, A \to A \times A$

$\pi_1, \pi_2, \ldots$ — the projections from a tuple to its first, second, ..., parts

$\boxtimes$ — pair product of functions: $(f \boxtimes g)(a \times b) \stackrel{\text{def}}{=} f(a) \times g(b)$

$\boxplus$ — pair co-product of functions

$\boxtimes$ — pair mapper of relations

$[a, b, c]$ — an ordered sequence of values. In Scala, `Seq(a, b, c)`

$\left\| \begin{array}{cc} x \to x & \mathbb{0} \\ \mathbb{0} & a \to a \times a \end{array} \right|$ — a function that works with disjunctive types (a "disjunctive function")

A.2 Detailed explanations

F^A means a type constructor F with a type parameter A. In Scala, this is `F[A]`. Type parameters are uppercase (A, B, ...). Type constructors with multiple type parameters are denoted by $F^{A,B,C}$. Nested type constructors such as Scala's `F[G[A]]` are denoted by F^{G^A}, meaning $F^{(G^A)}$.

$x^{:A}$ means a value x that has type A; this is a **type annotation**. In Scala, a type annotation is `x:A`. The colon symbol, :, in the superscript shows that A is not a type argument (as it would be in a type constructor, F^A). A less concise notation for $x^{:A}$ is $x : A$.

$\mathbb{1}$ means the unit type, and 1 means the value of the unit type. In Scala, the unit type is `Unit`, and its value is `()`. An example of this notation is $\mathbb{1} + A$, which corresponds to `Option[A]` in Scala.

$\mathbb{0}$ means the void type (the type with no values). In Scala, this is the type `Nothing`. The notation $\mathbb{0}$ is often used to denote an empty part of disjunctive types or values. E.g., the disjunctive type `Option[A]` has two parts: `Some[A]` and `None`. These types are denoted by $\mathbb{0} + A$ and $\mathbb{1} + \mathbb{0}$ respectively. Similarly,

$A + \mathbb{0}$ denotes the first part of the type $A + B$ (in Scala, `Left[A]`), while $\mathbb{0} + B$ denotes its second part (in Scala, `Right[A]`). Values of disjunctive types are denoted similarly. E.g., $x^{:A} + \mathbb{0}^{:B}$ denotes a value of the left part of the type $A + B$. In Scala, this value is written with fully annotated types as `Left[A,B](x)`.

$A + B$ means the disjunctive type made from types A and B. In Scala, this is the type `Either[A, B]`.

$x^{:A} + \mathbb{0}^{:B}$ denotes a value of a disjunctive type $A + B$, where x is the value of type A, which is the chosen case, and $\mathbb{0}$ stands for other possible cases. For example, $x^{:A} + \mathbb{0}^{B}$ is `Left[A,B](x)` in Scala. Type annotations $^{:A}$ and $^{:B}$ may be omitted if the types are unambiguous from the context.

$A \times B$ means the product type made from types A and B. In Scala, this is the tuple type `(A,B)`.

$a^{:A} \times b^{:B}$ means a value of a tuple type $A \times B$; in Scala, this is the tuple value `(a, b)`. Type annotations $^{:A}$ and $^{:B}$ may be omitted if the types are unambiguous from the context.

$A \to B$ means the type of functions A to B. In Scala, this is the type `A => B`. The function type's arrow binds weaker than $+$, which binds weaker than $\times$. So, $A + B \to C \times D$ means $(A + B) \to (C \times D)$.

$x^{:A} \to y$ means a nameless function with argument x of type A and function body y. (Usually, the body y will be an expression that uses x. In Scala, this is `{ x: A => y }`. Type annotation $^{:A}$ may be omitted if the type is unambiguous from the context.

id means the identity function. The type of its argument should be either specified as id^A or $\text{id}^{:A \to A}$, or else should be unambiguous from the context. In Scala, `identity[A]` corresponds to id^A.

$\stackrel{\text{def}}{=}$ means "equal by definition". A definition of a function f is written as $f \stackrel{\text{def}}{=} (x^{:\text{Int}} \to x + 10)$; in Scala, this is `val f = { x: Int => x + 10 }`. A definition of a type constructor F is written as $F^A \stackrel{\text{def}}{=} \mathbb{1} + A$; in Scala, this is `type F[A] = Option[A]`.

$\cong$ for types means an equivalence (an isomorphism) of types. For example, $A + A \times B \cong A \times (\mathbb{1} + B)$. The same symbol $\cong$ for *values* means "equivalent" according to an equivalence relation that needs to be established in the text. For example, if we have established an equivalence that allows nested tuples to be reordered whenever needed, we can write $(a \times b) \times c \cong a \times (b \times c)$, meaning that these values are mapped to each other by the established isomorphism functions.

$A^{:F^B}$ in type definitions means that the definition assigns the type F^B to the type expression A. This notation is used for defining unfunctors (GADTs). For example, the Scala code:

```
sealed trait F[A]
case class F1() extends F[Int]
case class F2[A](a: A) extends F[(A, String)]
```

defines an unfunctor denoted by $F^A \stackrel{\text{def}}{=} \mathbb{1}^{:F^{\text{Int}}} + A^{:F^{A \times \text{String}}}$.

$\wedge$ (conjunction), $\vee$ (disjunction), and $\Rightarrow$ (implication) are used in formulas of Boolean as well as constructive logic in Chapter 5, e.g., $\alpha \wedge \beta$, where Greek letters stand for logical propositions.

fmap_F is a functor F's the standard method `fmap` of the `Functor` typeclass. In Scala, this may be written as `Functor[F].fmap`. Since each functor F has its own specific implementation of fmap_F, the subscript "F" is *not* a type parameter of fmap_F. The function fmap_F has two type parameters, which may be written as $\text{fmap}_F^{A,B}$, and we may write its type signature as $\text{fmap}_F^{A,B} : (A \to B) \to F^A \to F^B$. In most cases, the type parameters A, B can be omitted without loss of clarity.

pu_F denotes a monad F's method `pure`. This function has type signature $A \to F^A$ and has a type parameter A. In the code notation, the type parameter may be either omitted or denoted as pu_F^A. If we are using `pure` with a complicated type, e.g., $\mathbb{1} + P^B$, as of the type parameter A, we may write the type parameter for clarity as $\text{pu}_F^{\mathbb{1}+P^B}$. The type signature of that function then becomes:

$$\text{pu}_F^{1+P^B} : \mathbb{1} + P^B \to F^{\mathbb{1}+P^B} \quad .$$

But in most cases we will not need to write out the type parameters.

flm_F denotes the curried version of a monad F's method `flatMap`. The type signature of flm_F is $\text{flm}_F : (A \to F^B) \to F^A \to F^B$. Note that Scala's standard `flatMap` type signature is not curried. The curried method flm_F is easier to use in calculations involving the monad laws.

ftn_F denotes a monad F's method `flatten` with the type signature $\text{ftn}_F : F^{F^A} \to F^A$.

$F^\bullet$ means the type constructor F understood as a type-level function, — that is, with a type parameter unspecified. In Scala, this is `F[_]`. The bullet symbol, $\bullet$, is used as a placeholder for the missing

type parameter. When no type parameter is needed, F means the same as $F^{\bullet}$. (For example, "a functor F" and "a functor $F^{\bullet}$" mean the same thing.) However, it is useful for clarity to be able to indicate the place where the type parameter would appear. For instance, functor composition is denoted as $F^{G^{\bullet}}$; in Scala 2, this is `Lambda[X => F[G[X]]]` when using the "kind projector" plugin.[1] When the type parameter B of a bifunctor $P^{A,B}$ is fixed to Z, we get a functor (with respect to A) denoted by $P^{\bullet,Z}$. Another example: $T_L^{M,\bullet}$ denotes a monad transformer for the base monad L and the foreign monad M. The foreign monad M is a type parameter in $T_L^{M,\bullet}$. The symbol $\bullet$ stands for the transformer's second type parameter. (The base monad L is not a type parameter in $T_L^{M,\bullet}$ because the construction of the monad transformer depends on the internal details of L.)

$F^{\bullet} \rightsquigarrow G^{\bullet}$ or $F \rightsquigarrow G$ means a natural transformation between two functors F and G. In some Scala libraries, this is denoted by `F ~> G`.

$\forall A.\, P^A$ is a universally quantified type expression, in which A is a bound type parameter.

$\exists A.\, P^A$ is an existentially quantified type expression, in which A is a bound type parameter.

$\fatsemi$ means the forward composition of functions: $f \fatsemi g$ (reads "f before g") is the function defined as $x \to g(f(x))$.

$\circ$ means the backward composition of functions: $f \circ g$ (reads "f after g") is the function defined as $x \to f(g(x))$.

$\circ$ with type constructors means their (backward) composition, for example $F \circ G$ denotes the type constructor $F^{G^{\bullet}}$. In Scala, this is `F[G[A]]`.

$x \triangleright f$ (the **pipe notation**) is a different syntax for $f(x)$. The value x is passed as the argument to the function f. In Scala, the expression $x \triangleright f$ is written as `x.pipe(f)` or, if `f` is a method, `x.f`. This syntax is used with many standard methods such as `size` or `toSeq`. Because the argument x is to the left of the function f in this notation, forward compositions of functions such as $x \triangleright f \triangleright g$ are naturally grouped to the left as it is done in Scala code, for example `x.toSeq.sorted`. The operation $\triangleright$ (pronounced "pipe") groups weaker than the forward composition ($\fatsemi$), and so we have $x \triangleright f \fatsemi g = x \triangleright f \triangleright g$ in this notation. Reasoning about code in the pipe notation uses the identities:

$$x \triangleright f = f(x) \quad , \qquad (x \triangleright f) \triangleright g = x \triangleright f \triangleright g \quad ,$$
$$x \triangleright f \fatsemi g = x \triangleright (f \fatsemi g) \quad , \qquad x \triangleright f \triangleright g = x \triangleright f \fatsemi g \quad .$$

The pipe symbol groups stronger than the function arrow, so $x \to y \triangleright f$ is the same as $x \to (y \triangleright f)$. Here are some examples of reasoning with functions in the pipe notation:

$$(a \to a \triangleright f) = (a \to f(a)) = f \quad ,$$
$$f \triangleright (y \to a \triangleright y) = a \triangleright f = f(a) \quad ,$$
$$f(y(x)) = x \triangleright y \triangleright f \neq x \triangleright (y \triangleright f) = f(y)(x) \quad .$$

The correspondence between the forward composition and the backward composition:

$$f \fatsemi g = g \circ f \quad ,$$
$$x \triangleright (f \fatsemi g) = x \triangleright f \fatsemi g = x \triangleright f \triangleright g = g(f(x)) = (g \circ f)(x) \quad .$$

$f^{\uparrow G}$ means a function f lifted to a functor G. For a function $f^{:A \to B}$, the application of $f^{\uparrow G}$ to a value $g^{:G^A}$ is written as $f^{\uparrow G}(g)$ or as $g \triangleright f^{\uparrow G}$. In Scala, this is `g.map(f)`. Nested lifting (i.e., lifting to the functor composition $H \circ G$) can be written as $f^{\uparrow G \uparrow H}$, which means $\left(f^{\uparrow G}\right)^{\uparrow H}$, and produces a function of type $H^{G^A} \to H^{G^B}$. Applying a nested lifting to a value h of type H^{G^A} is written as $h \triangleright f^{\uparrow G \uparrow H}$. In Scala, this is `h.map(_.map(f))`. The functor composition law is written as:

$$p^{\uparrow G} \fatsemi q^{\uparrow G} = (p \fatsemi q)^{\uparrow G} \quad .$$

The notation $x \triangleright p^{\uparrow G} \triangleright q^{\uparrow G}$ is intended to be similar to the Scala code `x.map(p).map(q)`.

[1] https://github.com/typelevel/kind-projector

$f^{\downarrow H}$ means a function f lifted to a contrafunctor H. For a function $f^{:A\to B}$, the application of $f^{\downarrow H}$ to a value $h : H^B$ is written as $h \triangleright f^{\downarrow H}$ and yields a value of type H^A. In Scala, this may be written as `h.contramap(f)`. Nested lifting is denoted as, e.g., $f^{\downarrow H\uparrow G} \stackrel{\text{def}}{=} (f^{\downarrow H})^{\uparrow G}$.

$\diamond_M$ means the Kleisli product operation for a given monad M. This is a binary operation working on two Kleisli functions of types $A \to M^B$ and $B \to M^C$ and yields a new function of type $A \to M^C$.

$L \triangleleft M$ denotes the monad L's transformer applied to a foreign monad M. We define $(K \triangleleft L) \triangleleft M \stackrel{\text{def}}{=} K \triangleleft (L \triangleleft M)$, which makes the monad transformer application into an associative operation.

$\oplus$ means the binary operation of a monoid, e.g., $x \oplus y$. For this expression to make sense, a specific monoid type should be defined . In Scala libraries, $x \oplus y$ is often denoted as `x |+| y`.

Δ means the standard "diagonal" function of type $\forall A.\, A \to A \times A$, i.e., $\Delta \stackrel{\text{def}}{=} a^{:A} \to a \times a$. In Scala:

```
def delta[A](a: A): (A, A) = (a, a)
```

$\pi_1, \pi_2, \ldots$ denote the functions extracting the first, second, ..., parts in a tuple. In Scala, π_1 is `(_._1)`.

$\boxtimes$ means the pair product of functions, defined by $(f \boxtimes g)(a \times b) = f(a) \times g(b)$. In Scala, the pair product can be implemented as a higher-order function:

```
def pair_product[A,B,P,Q](f: A => P, g: B => Q): ((A, B)) => (P, Q) = {
  case (a, b) => (f(a), g(b))
}
```

The operations Δ, π_i (where $i = 1, 2, \ldots$), and $\boxtimes$ allow us to express any function operating on tuples. Useful properties for reasoning about code of such functions:

$$\begin{aligned}
\text{identity law}: &\quad \Delta \fatsemi \pi_i = \text{id} \quad , \\
\text{naturality law}: &\quad f \fatsemi \Delta = \Delta \fatsemi (f \boxtimes f) \quad , \\
\text{left and right projection laws}: &\quad (f \boxtimes g) \fatsemi \pi_1 = \pi_1 \fatsemi f \quad , \qquad (f \boxtimes g) \fatsemi \pi_2 = \pi_2 \fatsemi g \quad , \\
\text{composition law}: &\quad (f \boxtimes g) \fatsemi (p \boxtimes q) = (f \fatsemi p) \boxtimes (g \fatsemi q) \quad ,
\end{aligned}$$

as well as the functor lifting laws for Δ and π_i:

$$\begin{aligned}
&f^{\uparrow F} \fatsemi \Delta = \Delta \fatsemi f^{\uparrow (F\times F)} = \Delta \fatsemi (f^{\uparrow F} \boxtimes f^{\uparrow F}) \quad , \\
&(f^{\uparrow F} \boxtimes f^{\uparrow G}) \fatsemi \pi_1 = f^{\uparrow (F\times G)} \fatsemi \pi_1 = \pi_1 \fatsemi f^{\uparrow F} \quad .
\end{aligned}$$

$[a, b, c]$ means an ordered sequence of values, such as a list or an array. In Scala, this can be `List(a, b, c)`, `Vector(a, b, c)`, `Array(a, b, c)`, or another collection type.

$f^{:Z+A\to Z+A\times A} \stackrel{\text{def}}{=} \left\| \begin{array}{cc} z \to z & \mathbb{0} \\ \mathbb{0} & a \to a \times a \end{array} \right|$ is the **matrix notation** for a function whose input and/or output type is a disjunctive type (a **disjunctive function**). In Scala, the function f is written as:

```
def f[Z, A]: Either[Z, A] => Either[Z, (A, A)] = {
  case Left(z)   => Left(z)       // Identity function on Z.
  case Right(a)  => Right((a, a)) // Delta on A.
}
```

The rows of the matrix indicate the different `case`s in the function's code, corresponding to the different parts of the input disjunctive type. If the input type is not disjunctive, there will be only one row. The columns of the matrix indicate the parts of the output disjunctive type. If the output type is not disjunctive, there will be only one column.

A matrix may show all parts of the disjunctive types in separate "type row" and "type column":

$$f^{:Z+A\to Z+A\times A} \stackrel{\text{def}}{=} \left\| \begin{array}{c|cc} & Z & A\times A \\ \hline Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & a \to a \times a \end{array} \right| \quad . \tag{A.1}$$

This notation clearly indicates the input and the output types of the function and is useful for reasoning about the code. The vertical double line separates the *input* types from the function code (output types have a single line). In the code above, the "type column" shows the parts of the input disjunctive type $Z + A$. The "type row" shows the parts of the output disjunctive type $Z + A \times A$.

The matrix notation is adapted to *forward* function composition ($f \fatsemi g$). Assume that A is a monoid type, and consider the composition of the function f shown above and the function g defined as:

```
def g[Z, A: Monoid]: Either[Z, (A, A)] => A = {
  case Left(_)           => Monoid[A].empty
  case Right((a1, a2))   => a1 |+| a2
}
```

In the matrix notation, the function g is written (with and without types) as:

$$g \stackrel{\text{def}}{=} \left|\begin{array}{c||c} & A \\ \hline Z & _ \to e^{:A} \\ A \times A & a_1 \times a_2 \to a_1 \oplus a_2 \end{array}\right| \quad , \qquad g \stackrel{\text{def}}{=} \left\|\begin{array}{c} _ \to e^{:A} \\ a_1 \times a_2 \to a_1 \oplus a_2 \end{array}\right| \quad .$$

The forward composition $f \fatsemi g$ is computed by forward-composing the matrix elements using the rules of the ordinary matrix multiplication, omitting any terms containing $\mathbb{0}$:

$$\begin{aligned} f \fatsemi g &= \left\|\begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & a \to a \times a \end{array}\right| \fatsemi \left\|\begin{array}{c} _ \to e^{:A} \\ a_1 \times a_2 \to a_1 \oplus a_2 \end{array}\right| \\ &= \left\|\begin{array}{c} \text{id} \fatsemi (_ \to e^{:A}) \\ (a \to a \times a) \fatsemi (a_1 \times a_2 \to a_1 \oplus a_2) \end{array}\right| = \left\|\begin{array}{c} _ \to e^{:A} \\ a \to a \oplus a \end{array}\right| \quad . \end{aligned}$$

Applying a function to a disjunctive value such as $x^{:Z+A}$ is computed by writing x as a row vector:

$$x = z^{:Z} + \mathbb{0}^{:A} = \left| \begin{array}{cc} z^{:Z} & \mathbb{0} \end{array} \right| \quad .$$

Then the computation $x \triangleright f \fatsemi g$ again follows the rules of matrix multiplication:

$$x \triangleright f \fatsemi g = \left| \begin{array}{cc} z^{:Z} & \mathbb{0} \end{array} \right| \triangleright \left\|\begin{array}{c} _ \to e^{:A} \\ a \to a \oplus a \end{array}\right| = z \triangleright (_ \to e) = e \quad .$$

Since the standard rules of matrix multiplication are associative, the properties of the $\triangleright$-notation such as $x \triangleright (f \fatsemi g) = (x \triangleright f) \triangleright g$ are guaranteed to hold with matrices.

To use the matrix notation with *backward* compositions ($f \circ g$), all code matrices need to be transposed. (A standard identity of matrix calculus is that the transposition reverses the order of composition: $(AB)^T = B^T A^T$.) The input types will then appear in the *top row* and the output types in the left column. The double line is at the top of a code matrix since that is where the function inputs come from. The above calculations are then rewritten as:

$$\begin{aligned} g \circ f &= \left|\begin{array}{c|cc} & Z & A \times A \\ \hline A & _ \to e^{:A} & a_1 \times a_2 \to a_1 \oplus a_2 \end{array}\right| \circ \left|\begin{array}{c|cc} & Z & A \\ \hline Z & \text{id} & \mathbb{0} \\ A \times A & \mathbb{0} & a \to a \times a \end{array}\right| \\ &= \left|\begin{array}{cc} \text{id} \fatsemi (_ \to e^{:A}) & (a \to a \times a) \fatsemi (a_1 \times a_2 \to a_1 \oplus a_2) \end{array}\right| = \left|\begin{array}{cc} _ \to e^{:A} & a \to a \oplus a \end{array}\right| \quad . \end{aligned}$$

$$(g \circ f)(x) = \left|\begin{array}{cc} _ \to e^{:A} & a \to a \oplus a \end{array}\right| \left|\begin{array}{c} z^{:Z} \\ \mathbb{0} \end{array}\right| = (_ \to e^{:A})(z) = e \quad .$$

The *forward* composition ($\fatsemi$) may be easier to read and to reason about in the matrix notation.

B Glossary of terms

Code notation A mathematical notation developed in this book for deriving properties of code in functional programs. Variables have optional type annotations, such as $x^{:A}$ or $f^{:A\to B}$. Nameless functions are denoted by$x^{:A} \to f$, products by $a \times b$, and values of a disjunctive type $A + B$ are written as $x^{:A} + \mathbb{0}^{:B}$ or $\mathbb{0}^{:A} + y^{:B}$. Functions working with disjunctive types are denoted by matrices. Lifting of functions to functors, such as $\text{fmap}_L(f)$, is denoted by $f^{\uparrow L}$; function compositions are denoted by $f \fatsemi g$ (forward composition) and $f \circ g$ (backward composition); and function applications by $f(x)$ or equivalently $x \triangleright f$. See Appendix A for details.

Contrafunctor A type constructor having the properties of a contravariant functor with respect to a type parameter. Instead of "contravariant functor", this book uses the shorter name "contrafunctor".

Disjunctive type A type representing one of several distinct possibilities. In Scala, this is usually implemented as a sealed trait extended by several case classes. The standard Scala disjunction types are `Option[A]` and `Either[A, B]`. Also known as **sum** type, **tagged union** type, **co-product** type, and variant type (in Object Pascal and in OCaml). The shortest name is "sum type," but the English word "disjunctive" is less ambiguous to the ear than "sum".

Exponential-polynomial type A type constructor built using products, disjunctions (sums or co-products), and function types ("exponentials"), as well as type parameters and fixed types. For example, `type F[A] = Either[(A,A), Int=>A]` is an exponential-polynomial type constructor. Such type constructors are always profunctors and can also be functors or contrafunctors.

Functor block A short syntax for composing several `map`, `flatMap`, and `filter` operations applied to a functor-typed value. The type constructor corresponding to that value must be a functor and is fixed throughout the entire functor block. For example, the Scala code

```
for { x <- List(1,2,3); y <- List(10, x); if y > 2 }
  yield 2 * y
```

is equivalent to the code

```
List(1, 2, 3).flatMap(x => List(10, x))
  .filter(y => y > 1).map(y => 2 * y)
```

and computes the value `List(20, 20, 20, 6)`. This is a functor block that "raises" computations to the `List` functor. Similar syntax exists in a number of languages and is called a **"for-comprehension"** or a "list comprehension" in Python, **"do-notation"** in Haskell, and **"computation expressions"** in F#. I use the name "functor block" in this book because it is shorter and more descriptive. (The type constructor used in a functor block needs to be at least a functor but does not have to be a monad.)

Kleisli function A function with type signature $A \to M^B$ (in Scala, `A => M[B]`) for some fixed monad M. Also called a Kleisli morphism (a morphism in the Kleisli category corresponding to the monad M). The monadic method $\text{pure}_M : A \to M^A$ has the type signature of a Kleisli function. The Kleisli composition operation, $\diamond_M$, is a binary operation that combines two Kleisli functions (of types $A \to M^B$ and $B \to M^C$) into a new Kleisli function (of type $A \to M^C$).

Method This word is used in two ways: 1) A method_1 is a Scala function defined as a member of a typeclass. For example, `flatMap` is a method defined in the `Monad` typeclass. 2) A method_2 is a

Scala function defined as a member of a data type declared as a Java-compatible `class` or `trait`. Trait methods$_2$ are necessary in Scala when implementing functions whose arguments have type parameters (because Scala function values defined via `val` cannot have type parameters). So, many typeclasses such as `Functor` or `Monad`, whose methods$_1$ require type parameters, will use Scala `traits` with methods$_2$ for their implementation. The same applies to type constructions with quantified types, such as the Church encoding.

Nameless function An expression of function type, representing a function. For example, `(x: Int) => x * 2`. Also known as function expression, function literal, anonymous function, closure, lambda-function, lambda-expression, or simply a "lambda".

Partial type-to-value function (PTVF) A function with a type parameter but defined only for a certain subset of types. In Scala, PTVFs are implemented via a typeclass constraint:

```
def double[T: Semigroup](t: T): T = implicitly[Semigroup[T]].combine(t, t)
```

This PTVF is defined only for types `T` for which a `Semigroup` typeclass instance is available.

Polynomial functor A type constructor built using disjunctions (sums), products (tuples), type parameters and fixed types. For example, in Scala, `type F[A] = Either[(Int, A), A]` is a polynomial functor with respect to the type parameter `A`, while `Int` is a fixed type (not a type parameter). Polynomial functors are also known as **algebraic data types**.

Product type A type representing several values given at once. In Scala, product types are the tuple types, for example `(Int, String)`, and case classes. Also known as **tuple** type, **struct** (in C and C++), and **record**.

Profunctor A type constructor whose type parameter occurs in both covariant and contravariant positions and satisfying the appropriate laws; see Section 6.4.1.

Type notation A mathematical notation for type expressions developed in this book for easier reasoning about types in functional programs. Disjunctive types are denoted by +, product types by $\times$, and function types by $\to$. The unit type is denoted by $\mathbb{1}$, and the void type by $\mathbb{0}$. The function arrow $\to$ groups weaker than +, which in turn groups weaker than $\times$. This means

$$Z + A \to Z + A \times A \quad \text{is the same as} \quad (Z + A) \to (Z + (A \times A)) \quad .$$

Type parameters are denoted by superscripts. As an example, the Scala definition

```
type F[A] = Either[(A, A => Option[Int]), String => List[A]]
```

is written in the type notation as

$$F^A \stackrel{\text{def}}{=} A \times (A \to \mathbb{1} + \text{Int}) + (\text{String} \to \text{List}^A) \quad .$$

Unfunctor A type constructor that cannot possibly be a functor, nor a contrafunctor, nor a profunctor. An example is a type constructor with explicitly indexed type parameters, such as $F^A \stackrel{\text{def}}{=} (A \times A)^{:F^{\text{Int}}} + (\text{Int} \times A)^{:F^1}$. The Scala code for this type constructor is

```
sealed trait F[A]
final case class F1[A](x: A, y: A)   extends F[Int]
final case class F2[A](s: Int, t: A) extends F[Unit]
```

This can be seen as a **GADT** (generalized algebraic data type) that uses specific values of type parameters instead of the type parameter A in at least some of its case classes.

B.1 How the term "algebra" is used in functional programming

This book avoids using the terms "algebra" or "algebraic" because these terms are too ambiguous. As we will see, the FP community uses the word "algebra" in at least *three* incompatible ways.

Definition 0. In mathematics, an "algebra" is a vector space with a multiplication operation ($*$) satisfying certain properties, such as distributivity: $z * (x + y) = z * x + z * y$. For example, the set of all 10×10 matrices with real coefficients is a 100-dimensional vector space that satisfies the required properties. Such matrices form an "algebra" in the mathematical sense. (This meaning of the word "algebra" is *not* used in FP.)

Definition 1. For a given functor F, an "F-algebra" is a type T and a function with type signature $F^T \to T$. (This definition comes from category theory. This book does not use F-algebras because knowing their category-theoretic properties does not help in practical programming.) The name "F-algebra" emphasizes the dependence on a chosen functor F. There is no direct connection between this "algebra" and Definition 0, except when the functor F is defined by $F^A \stackrel{\text{def}}{=} A \times A$: a function of type $T \times T \to T$ may then be interpreted as a "multiplication" operation for values of type T. However, even in that case T will not necessarily satisfy the laws of a vector space.

A recursive type T defined by $F^T \cong T$ has an isomorphism $F^T \to T$ and so is an F-algebra. Recursive polynomial types are known as "algebraic" in this sense:[1] they are F-algebras with a polynomial recursion scheme $F^\bullet$ and an added requirement of isomorphism for the map $F^T \to T$.

The "algebra" defined in this sense is also relevant to the Church encoding of a free monad (also known as the "tagless final" pattern), which is the type $\forall E^\bullet. (S^{E^\bullet} \rightsquigarrow E^\bullet) \to E^A$. This type uses a higher-order type $S^{E^\bullet}$ parameterized by a *type constructor* parameter $E^\bullet$. In this context, one calls an "algebra" a value of type $S^{E^\bullet} \rightsquigarrow E^\bullet$, which is an S-algebra in the category of type constructors. However, knowing that $S^{E^\bullet} \rightsquigarrow E^\bullet$ is an S-algebra in the category of type constructors does not provide any help or additional insights for practical work with the Church encoding of free monads.

The higher-order type constructor S is used to parameterize the effects described by a Church-encoded free monad, so this book calls it the "effect constructor".

Definition 2. Polynomial data types (for example, `type F[A] = Option[(A, A)]`) are called "algebraic data types". Here, the word "algebraic" suggests "school-level algebra" dealing with polynomials: those data types are built from "sums" and "products" of types. The type `F[A]` may admit some binary or unary operations, but this does not turn it into an algebra in the sense of Definitions 0 or 1. Instead of calling types "algebraic", this book uses more precise terms "polynomial types" and "exponential-polynomial types". Data types containing functions, such as `Option[Int => (A, A)]`, are exponential-polynomial (but not polynomial).

Definition 3. One talks about the "algebra" of properties of functions such as `map` or `flatMap`, meaning that these functions satisfy certain "algebraic" laws (e.g., the composition, naturality, or associativity laws). But these laws do not make functions `map` or `flatMap` into an algebra in the sense of Definition 0 or in the sense of Definition 1. There is also no relation to the algebraic data types of Definition 2. So, this is a different usage of the word "algebra". However, there is no general "algebra of laws" that we can use; every derivation proceeds in a different way, specific to the laws being proved. In mathematics, "algebraic" equations are distinguished from differential or integral equations. In that sense, the laws used in FP are *always* algebraic: they are just equations with compositions and applications of functions. So, there is no advantage in calling laws "algebraic" within the context of FP. This book calls them "equations" or just "laws".

We find that the current usage of the word "algebra" in FP is both inconsistent and unhelpful to software practitioners. In this book, the word "algebra" always means a branch of mathematics, as in "high-school algebra". Instead of "algebras" as in Definitions 1 to 3, this book talks about "polynomial types" or "recursive polynomial types"; "equations" or "laws"; and an "effect constructor" S.

[1] For some proofs, see https://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt

C Deriving code from types. The LJT algorithm

We have seen in Chapter 5 how to derive code from a type signature as long as one has a proof of the sequent corresponding to that type signature. But Section 5.2.3 gave an example showing that the rules in Table 5.7 do not provide an algorithm for finding a proof for a given sequent.

To illustrate this problem on another example, let us try proving the sequent:

$$A, B \vee C \vdash (A \wedge B) \vee C \quad .$$

We expect that this sequent is provable because we can write the corresponding Scala code:

```
def f[A, B, C](a: A): Either[B, C] => Either[(A, B), C] = {
  case Left(b)    => Left((a, b))
  case Right(c)   => Right(c)
}
```

How can we obtain a proof of this sequent according to the rules in Table 5.7? We find that we could potentially apply the rules "create `Left`", "create `Right`", "use `Either`", and "use function". However, no matter what rule we choose, we will get stuck at the next step. Let us see why:

To apply "create `Left`", we first need to prove the sequent $A, B \vee C \vdash A \wedge B$. But this sequent cannot be proved: we do not necessarily have values of both types A and B if we are only given values of type A and of type `Either[B, C]`. To apply "create `Right`", we need to prove the sequent $A, B \vee C \vdash C$. Again, we find that this sequent cannot be proved. The next choice is the rule "use `Either`" that matches any goal of the sequent as the proposition γ. But we are then required to choose two new propositions (α and β) such that we can prove $A, B \vee C \vdash \alpha \vee \beta$ as well as $A, B \vee C, \alpha \vdash (A \wedge B) \vee C$ and $A, B \vee C, \beta \vdash (A \wedge B) \vee C$. It is not clear how we should choose α and β in order to make progress in the proof. The remaining rule, "use function", similarly requires us to choose a new proposition α such that we can prove $A, B \vee C \vdash \alpha$ and $A, B \vee C \vdash \alpha \Rightarrow ((A \wedge B) \vee C)$. Again, the algorithm gives us no guidance about choosing α.

We see that the rules in Table 5.2 are not helpful for proof search. Certain rules ("use function" and "use `Either`") require us to choose new unknown propositions and prove more complicated sequents than the ones we had before. For instance, the rule "use function" gives a proof of $\Gamma \vdash \beta$ if we first choose some other proposition α and prove the sequents $\Gamma \vdash \alpha$ and $\Gamma \vdash \alpha \Rightarrow \beta$. The rule does not say how to choose the proposition α correctly. We need to guess the correct α by trial and error. Even after choosing α in some way, we will have to prove a more complicated sequent ($\Gamma \vdash \alpha \Rightarrow \beta$). It is not guaranteed that we are getting closer to finding the proof of the initial sequent ($\Gamma \vdash \beta$).

It is far from obvious how to overcome that difficulty. Mathematicians have studied the constructive logic for more than 60 years, trying to replace the rules in Table 5.7 by a different but equivalent set of derivation rules that require no guessing when looking for a proof. The first partial success came in 1935 with an algorithm called "LJ".[1] The LJ algorithm still had a significant problem: one of its derivation rules may be applied infinitely many times. So, the LJ algorithm is not guaranteed to terminate without some heuristics for avoiding an infinite loop. A terminating version of the LJ algorithm, called LJT, was formulated in 1992.[2]

We will first present the LJ algorithm. Although that algorithm does not guarantee termination, it is simpler to apply by hand. Then we will show how to pass from the LJ algorithm to LJT.

[1] See https://en.wikipedia.org/wiki/Sequent_calculus#Overview

[2] The history of this research is outlined in https://research-repository.st-andrews.ac.uk/handle/10023/8824. An often cited paper by R. Dyckhoff is https://philpapers.org/rec/DYCCSC

$$\frac{}{\Gamma, X \vdash X}\ (\text{Id}) \qquad \frac{}{\Gamma \vdash \top}\ (\text{True})$$

$$\frac{\Gamma, A \Rightarrow B \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C}\ (\text{Left} \Rightarrow) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ (\text{Right} \Rightarrow)$$

$$\frac{\Gamma, A_i \vdash C}{\Gamma, A_1 \wedge A_2 \vdash C}\ (\text{Left}\wedge_i) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\text{Right}\wedge)$$

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}\ (\text{Left}\vee) \qquad \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 \vee A_2}\ (\text{Right}\vee_i)$$

Figure C.1: Axioms and derivation rules of the LJ algorithm. Each of the rules "(Left$\wedge_i$)" and "(Right$\vee_i$)" have two versions, with $i = 1$ or $i = 2$.

C.1 The LJ algorithm

Figure C.1 shows the LJ algorithm's axioms and derivation rules. Each rule says that the bottom sequent will be proved if proofs are given for sequent(s) at the top. For each possible sub-expression (conjunction $X \wedge Y$, disjunction $X \vee Y$, and implication $X \Rightarrow Y$) there is one rule where that sub-expression is a premise (at "left") and one rule where that sub-expression is the goal (at "right"). Those sub-expressions are shown in blue in Figure C.1 to help us look for a proof. To find out which rules apply, we just need to match some part of the sequent to a blue sub-expression in the rules.

It turns out that the rules in Figure C.1 are equivalent to the rules in Table 5.2. The proof is beyond the scope of this book. We only remark that this equivalence is far from obvious. To prove it, one needs to demonstrate that any sequent derived through the first set of rules is also derivable through the second set, and vice versa.

To illustrate the LJ algorithm, let us prove the sequent (5.6). Denote that sequent by S_0:

$$S_0 \stackrel{\text{def}}{=} \emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta \quad .$$

Since the goal of S_0 contains an implication, we use the rule "(Right $\Rightarrow$)" and get a sequent S_1:

$$S_1 \stackrel{\text{def}}{=} (\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta \quad .$$

Now the implication is in the premise, so we use the rule "(Left $\Rightarrow$)" and get two new sequents:

$$S_2 \stackrel{\text{def}}{=} (\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \alpha \Rightarrow \alpha \quad , \qquad S_3 \stackrel{\text{def}}{=} \beta \vdash \beta \quad .$$

Sequent S_3 follows from the "(Id)" axiom, so it remains to prove S_2. Since S_2 contains an implication both as a premise and as the goal, we may apply either the rule "(Left $\Rightarrow$)" or the rule "(Right $\Rightarrow$)". We choose to apply "(Left $\Rightarrow$)" and get two new sequents:

$$S_4 \stackrel{\text{def}}{=} (\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \alpha \Rightarrow \alpha \quad , \qquad S_5 : \beta \vdash \alpha \Rightarrow \alpha \quad .$$

Notice that $S_4 = S_2$. So, our proof search is getting into an infinite loop trying to prove the same sequent S_2 over and over again. We can prove S_5 but this will not help us break the loop.

Once we recognize the problem, we backtrack to the point where we chose to apply "(Left $\Rightarrow$)" to S_2. That was a bad choice, so let us instead apply "(Right $\Rightarrow$)" to S_2. This yields a new sequent S_6:

$$S_6 \stackrel{\text{def}}{=} (\alpha \Rightarrow \alpha) \Rightarrow \beta, \alpha \vdash \alpha \quad .$$

This sequent follows from the "(Id)"axiom. There are no more sequents to prove, so the proof of S_0 is finished. It can be drawn as a **proof tree** like this:

$$\xrightarrow{S_0} (\text{Right} \Rightarrow) \xrightarrow{S_1} (\text{Left} \Rightarrow) \xrightarrow{S_2} (\text{Right} \Rightarrow) \xrightarrow{S_6} (\text{Id}) \qquad (\text{Left} \Rightarrow) \xrightarrow{S_3} (\text{Id})$$

$$\frac{}{\Gamma, A \vdash A}\ (\text{Id}) \qquad \text{Proof}\,(\Gamma, A \vdash A)_{\text{given } p^{:\Gamma}, x^{:A}} = x$$

$$\frac{}{\Gamma \vdash \top}\ (\text{True}) \qquad \text{Proof}\,(\Gamma \vdash \top)_{\text{given } p^{:\Gamma}} = 1$$

$$\frac{\Gamma, A \Rightarrow B \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C}\ (\text{Left} \Rightarrow) \qquad \text{Proof}\,(\Gamma, A \Rightarrow B \vdash C)_{\text{given } p^{:\Gamma}, q^{:A \to B}} = \text{Proof}\,(\Gamma, B \vdash C)_{\text{given } p, b^{:B}}$$

$$\text{where} \quad b^{:B} \stackrel{\text{def}}{=} q\big(\text{Proof}\,(\Gamma, A \Rightarrow B \vdash A)_{\text{given } p, q}\big)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ (\text{Right} \Rightarrow) \qquad \text{Proof}\,(\Gamma \vdash A \Rightarrow B)_{\text{given } p^{:\Gamma}} = x^{:A} \to \text{Proof}\,(\Gamma, A \vdash B)_{\text{given } p^{:\Gamma}, x^{:A}}$$

$$\frac{\Gamma, A \vdash C}{\Gamma, A \wedge B \vdash C}\ (\text{Left}\wedge_1) \qquad \text{Proof}\,(\Gamma, A \wedge B \vdash C)_{\text{given } p^{:\Gamma}, (a^{:A} \times b^{:B})}$$

$$= \text{Proof}\,(\Gamma, A \vdash C)_{\text{given } p^{:\Gamma}, a^{:A}}$$

$$\frac{\Gamma, B \vdash C}{\Gamma, A \wedge B \vdash C}\ (\text{Left}\wedge_2) \qquad \text{Proof}\,(\Gamma, A \wedge B \vdash C)_{\text{given } p^{:\Gamma}, (a^{:A} \times b^{:B})}$$

$$= \text{Proof}\,(\Gamma, B \vdash C)_{\text{given } p^{:\Gamma}, b^{:B}}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\text{Right}\wedge) \qquad \text{Proof}\,(\Gamma \vdash A \wedge B)_{\text{given } p^{:\Gamma}}$$

$$= \text{Proof}\,(\Gamma \vdash A)_{\text{given } p^{:\Gamma}} \times \text{Proof}\,(\Gamma \vdash B)_{\text{given } p^{:\Gamma}}$$

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}\ (\text{Left}\vee) \qquad \text{Proof}\,(\Gamma, A \vee B \vdash C)_{\text{given } p^{:\Gamma}, q^{:A+B}}$$

$$= q \triangleright \begin{array}{c||c|} & C \\ \hline A & x^{:A} \to \text{Proof}\,(\Gamma, A \vdash C)_{\text{given } p, x} \\ B & y^{:B} \to \text{Proof}\,(\Gamma, B \vdash C)_{\text{given } p, y} \end{array}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\text{Right}\vee_1) \qquad \text{Proof}\,(\Gamma \vdash A \vee B)_{\text{given } p^{:\Gamma}} = \text{Proof}\,(\Gamma \vdash A) + \mathbb{0}^{:B}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\text{Right}\vee_2) \qquad \text{Proof}\,(\Gamma \vdash A \vee B)_{\text{given } p^{:\Gamma}} = \mathbb{0}^{:A} + \text{Proof}\,(\Gamma \vdash B)$$

Figure C.2: Proof transformers for the rules of the LJ algorithm.

The nodes of the proof tree are axioms or derivation rules, and the edges are intermediate sequents required by the rules. Some rule nodes branch into several sequents because some rules require more than one new sequent to be proved. The leaves of the tree are axioms that do not require proving any further sequents.

C.1.1 Extracting code from proofs

In the Curry-Howard correspondence, a sequent (such as $A, B, ..., C \vdash X$) represents the task of writing a fully parametric code expression of type X that uses some given values of types A, B, ..., C. The sequent is true (i.e., provable) if that code expression can be found. So, the code serves as an "evidence of proof" for the sequent.

In the previous section, we have found a proof of the sequent S_0, which represents the task of writing a fully parametric function with type signature $((A \to A) \to B) \to B)$. Let us now see how we can extract the code of that function from the proof of the sequent S_0.

We start from the leaves of the proof tree and move step by step towards the initial sequent. At each step, we shorten the proof tree by replacing some sequent by its corresponding evidence-of-proof code. Eventually we will replace the initial sequent by its corresponding code. Let us see how

this procedure works for the proof tree of the sequent S_0 shown in the previous section.

Since the leaves are axioms, let us write the code corresponding to each axiom of LJ:

$$\frac{}{\Gamma, X \vdash X}\,(\text{Id}) \quad : \quad \text{Proof}\,(\Gamma, X \vdash X)_{\text{given } p^{:\Gamma}, x^{:X}} = x \quad ;$$

$$\frac{}{\Gamma \vdash \top}\,(\text{True}) \quad : \quad \text{Proof}\,(\Gamma \vdash \top)_{\text{given } p^{:\Gamma}} = 1 \quad .$$

Here we denote explicitly the values (such as p and x) given as premises to the sequent. The notation $p^{:\Gamma}$ means all values given in the set of premises Γ. Below we will assume that the propositions α and β correspond to types A and B; that is, $\alpha \stackrel{\text{def}}{=} \mathcal{CH}(A)$ and $\beta \stackrel{\text{def}}{=} \mathcal{CH}(B)$.

The leaves in the proof tree for S_0 are the "(Id)" axioms used to prove the sequents S_3 and S_6. Let us write the code that serves as the "evidence of proof" for these sequents. For brevity, we denote $\gamma \stackrel{\text{def}}{=} (\alpha \Rightarrow \alpha) \Rightarrow \beta$ and $C \stackrel{\text{def}}{=} (A \to A) \to B$, so that $\gamma = \mathcal{CH}(C)$. Then we can write:

$$S_3 \stackrel{\text{def}}{=} \beta \vdash \beta \quad , \qquad \text{Proof}\,(S_3)_{\text{given } y^{:B}} = y \quad ,$$

$$S_6 \stackrel{\text{def}}{=} \gamma, \alpha \vdash \alpha \quad , \qquad \text{Proof}\,(S_6)_{\text{given } q^{:C}, x^{:A}} = x \quad .$$

Note that the proof of S_6 does not use the first given value $q^{:C}$ (corresponding to the premise γ).

We now shorten the proof tree by replacing the sequents S_3 and S_6 by their "evidence of proof":

$$\xrightarrow{S_0} (\text{Right} \Rightarrow) \xrightarrow{S_1} (\text{Left} \Rightarrow) \xrightarrow{S_2} (\text{Right} \Rightarrow) \xrightarrow{(x)_{\text{given } q^{:C}, x^{:A}}} \square$$

$$(\text{Left} \Rightarrow) \xrightarrow{(y)_{\text{given } y^{:B}}} \square$$

The next step is to consider the proof of S_2, which is found by applying the rule "(Right $\Rightarrow$)". This rule promises to give a proof of S_2 if we have a proof of S_6. In order to extract code from that rule, we can write a function that transforms a proof of S_6 into a proof of S_2. We call this function the **proof transformer** corresponding to the rule "(Right $\Rightarrow$)". That rule and its transformer are defined as:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\,(\text{Right} \Rightarrow) \quad : \quad \text{Proof}\,(\Gamma \vdash A \Rightarrow B)_{\text{given } p^{:\Gamma}} = x^{:A} \to \text{Proof}\,(\Gamma, A \vdash B)_{\text{given } p^{:\Gamma}, x^{:A}} \quad .$$

Applying the proof transformer to the known proof of S_6, we obtain a proof of S_2:

$$\text{Proof}\,(S_2)_{\text{given } q^{:C}} = x^{:A} \to \text{Proof}\,(S_6)_{\text{given } q^{:C}, x^{:A}} = (x^{:A} \to x)_{\text{given } q^{:C}} \quad .$$

The proof tree can be now shortened to:

$$\xrightarrow{S_0} (\text{Right} \Rightarrow) \xrightarrow{S_1} (\text{Left} \Rightarrow) \xrightarrow{(x^{:A} \to x)_{\text{given } q^{:C}}} \square$$

$$(\text{Left} \Rightarrow) \xrightarrow{(y)_{\text{given } y^{:B}}} \square$$

The next step is to get the proof of S_1 obtained by applying the rule "(Left $\Rightarrow$)". That rule requires two previous sequents, so its transformer is a function of two previously obtained proofs:

$$\frac{\Gamma, A \Rightarrow B \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C}\,(\text{Left} \Rightarrow) \quad :$$

$$\text{Proof}\,(\Gamma, A \Rightarrow B \vdash C)_{\text{given } p^{:\Gamma}, q^{:A \to B}} = \text{Proof}\,(\Gamma, B \vdash C)_{\text{given } p^{:\Gamma}, b^{:B}}$$

$$\text{where} \quad b^{:B} \stackrel{\text{def}}{=} q\big(\text{Proof}\,(\Gamma, A \Rightarrow B \vdash A)_{\text{given } p^{:\Gamma}, q^{:A \to B}}\big) \quad .$$

In the proof tree shown above, we obtain a proof of S_1 by applying this proof transformer to the proofs of S_2 and S_3:

$$\text{Proof}\,(S_1)_{\text{given } q^{:C}} = \text{Proof}\,(S_3)_{\text{given } b^{:B}} \text{ where } b^{:B} \stackrel{\text{def}}{=} q(\text{Proof}\,(S_2))_{\text{given } q^{:C}}$$

$$= b \text{ where } b^{:B} \stackrel{\text{def}}{=} q(x^{:A} \to x)_{\text{given } q^{:C}} = q(x^{:A} \to x)_{\text{given } q^{:C}} \quad .$$

Substituting this proof into the proof tree, we shorten the tree to:

$$\xrightarrow{S_0} (\text{Right} \Rightarrow) \xrightarrow{q(x^{:A} \to x)_{\text{given } q^{:C}}} \square$$

It remains to obtain the proof of S_0 by applying the proof transformer of the rule "(Right ⇒)":

$$\begin{aligned}&\text{Proof}\,(S_0) = \text{Proof}\,(\emptyset \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta)\\&= q^{:(A\to A)\to B} \to \text{Proof}\,(S_1)_{\text{given } q^{:C}} = q^{:(A\to A)\to B} \to q(x^{:A} \to x) \quad .\end{aligned}$$

The proof tree is now shortened to a single code expression, $q^{:(A\to A)\to B} \to q(x^{:A} \to x)$, that serves as an evidence of proof for S_0 because this code has type $((A \to A) \to B) \to B$. In this way, we have derived the code of a fully parametric function from its type signature.

Figure C.2 shows the proof transformers for all the rules of the LJ algorithm. Apart from the special rule "(Left ⇒)", all other rules have proof transformers using just one of the code constructions ("create function", "create tuple", "use tuple", etc.) allowed within fully parametric code.

C.2 The LJT algorithm

As we have seen, the LJ algorithm can enter a loop if we apply the rule "(Left ⇒)" and obtain a sequent we already had at a previous step. That rule requires us to prove two new sequents:

$$\frac{\Gamma, A \Rightarrow B \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C}\ (\text{Left} \Rightarrow) \quad .$$

A sign of trouble is that the first of these sequents ($\Gamma, A \Rightarrow B \vdash A$) does not have a simpler form than the initial sequent ($\Gamma, A \Rightarrow B \vdash C$). So, it is not clear that we are getting closer to completing the proof. If $A = C$, the new sequent will simply repeat the initial sequent, immediately creating a loop.

In some cases, a repeated sequent will occur after more than one step. It is not easy to formulate rigorous conditions for stopping the loop or for avoiding the rule "(Left ⇒)".

The LJT algorithm solves this problem by removing the rule "(Left ⇒)" from the LJ algorithm. Instead, *four* new rules are introduced. Each of these rules contains a different pattern instead of A in the premise $A \Rightarrow C$:

$$(A \text{ is atomic})\ \frac{\Gamma, A, B \vdash D}{\Gamma, A, A \Rightarrow B \vdash D}\ (\text{Left} \Rightarrow_A) \qquad \frac{\Gamma, A \Rightarrow B \Rightarrow C \vdash D}{\Gamma, (A \wedge B) \Rightarrow C \vdash D}\ (\text{Left} \Rightarrow_\wedge)$$

$$\frac{\Gamma, B \Rightarrow C \vdash A \Rightarrow B \qquad \Gamma, C \vdash D}{\Gamma, (A \Rightarrow B) \Rightarrow C \vdash D}\ (\text{Left} \Rightarrow_\Rightarrow) \qquad \frac{\Gamma, A \Rightarrow C, B \Rightarrow C \vdash D}{\Gamma, (A \vee B) \Rightarrow C \vdash D}\ (\text{Left} \Rightarrow_\vee)$$

The rule "Left $\Rightarrow_A$" applies only if the implication starts with an "atomic" type expression, i.e., a single type parameter or a unit type. In all other cases, the implication must start with a conjunction, a disjunction, or an implication, which means that one of the three remaining rules will apply.

The LJT algorithm retains all the rules in Figure C.2 except the rule "(Left ⇒)", which is replaced by the four new rules. It is far from obvious that the new rules are equivalent to the old ones. It took mathematicians several decades to come up with the LJT rules and to prove their validity. This book will rely on that result and will not attempt to prove it.

The proof transformers for the new rules are shown in Figure C.3. Figures C.2–C.3 define the set of proof transformers sufficient for using the LJT algorithm in practice. The `curryhoward` library[3] implements these proof transformers.

The most complicated of the new rules is the rule "(Left $\Rightarrow_\Rightarrow$)". Applying that rule's transformer results in evidence-of-proof code that is longer than the code obtained via LJ's rule transformers. The resulting code will need to be simplified symbolically.

[3] See https://github.com/Chymyst/curryhoward

$$\frac{\Gamma, A, B \vdash D}{\Gamma, A, A \Rightarrow B \vdash D}\ (\text{Left} \Rightarrow_A) \qquad \begin{aligned} &\text{Proof}\,(\Gamma, A, A \Rightarrow B \vdash D)_{\text{given } p^{:\Gamma}, x^{:A}, q^{:A\to B}} \\ &\quad = \text{Proof}\,(\Gamma, A, B \vdash D)_{\text{given } p, x, q(x)} \end{aligned}$$

$$\frac{\Gamma, A \Rightarrow B \Rightarrow C \vdash D}{\Gamma, (A \wedge B) \Rightarrow C \vdash D}\ (\text{Left} \Rightarrow_\wedge) \qquad \begin{aligned} &\text{Proof}\,(\Gamma, (A \wedge B) \Rightarrow C \vdash D)_{\text{given } p^{:\Gamma}, q^{:A\times B\to C}} \\ &\quad = \text{Proof}\,(\Gamma, A \Rightarrow B \Rightarrow C \vdash D)_{\text{given } p, (a^{:A}\to b^{:B}\to q(a\times b))} \end{aligned}$$

$$\frac{\Gamma, A \Rightarrow C, B \Rightarrow C \vdash D}{\Gamma, (A \vee B) \Rightarrow C \vdash D}\ (\text{Left} \Rightarrow_\vee) \qquad \begin{aligned} &\text{Proof}\,(\Gamma, (A \vee B) \Rightarrow C \vdash D)_{\text{given } p^{:\Gamma}, q^{:A+B\to C}} \\ &\quad = \text{Proof}\,(\Gamma, A \Rightarrow C, B \Rightarrow C \vdash D)_{\text{given } p, r, s} \\ &\quad \text{where } r \stackrel{\text{def}}{=} a^{:A} \to q(a + \mathbb{0}) \text{ and } s \stackrel{\text{def}}{=} b^{:B} \to q(\mathbb{0} + b) \end{aligned}$$

$$\frac{\Gamma, B \Rightarrow C \vdash A \Rightarrow B \qquad \Gamma, C \vdash D}{\Gamma, (A \Rightarrow B) \Rightarrow C \vdash D}\ (\text{Left} \Rightarrow_\Rightarrow) \qquad \begin{aligned} &\text{Proof}\,(\Gamma, (A \Rightarrow B) \Rightarrow C \vdash D)_{\text{given } p^{:\Gamma}, q^{:(A\to B)\to C}} \\ &\quad = \text{Proof}\,(\Gamma, C \vdash D)_{\text{given } p, c} \\ &\quad \text{where } c^{:C} \stackrel{\text{def}}{=} q\big(\text{Proof}\,(\Gamma, B \Rightarrow C \vdash A \Rightarrow B)_{\text{given } p, r}\big) \\ &\quad \text{and } r^{:B\to C} \stackrel{\text{def}}{=} b^{:B} \to q(_^{:A} \to b) \end{aligned}$$

Figure C.3: Proof transformers for the four new rules of the LJT algorithm.

As an example of using the LJT algorithm, we prove the sequent $S_0 = \emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta$ from the previous section. At each step, only one LJT rule applies to each sequent. The initial part of the proof tree looks like this:

$$\xrightarrow{\emptyset\vdash((\alpha\Rightarrow\alpha)\Rightarrow\beta)\Rightarrow\beta} (\text{Right} \Rightarrow) \xrightarrow{(\alpha\Rightarrow\alpha)\Rightarrow\beta\vdash\beta} (\text{Left} \Rightarrow_\Rightarrow) \begin{array}{c} \xrightarrow{\beta\vdash\beta} \\ \xrightarrow{\alpha\Rightarrow\beta\vdash\alpha\Rightarrow\alpha} \end{array}$$

The proofs for the sequents $\beta \vdash \beta$ and $\alpha \Rightarrow \beta \vdash \alpha \Rightarrow \alpha$ are the same as before:

$$\text{Proof}\,(\beta \vdash \beta)_{\text{given } y^{:B}} = y \quad , \qquad \text{Proof}\,(\alpha \Rightarrow \beta \vdash \alpha \Rightarrow \alpha)_{\text{given } r^{:A\to B}} = x^{:A} \to x \quad .$$

Substituting these proofs into the proof transformer of the rule "(Left $\Rightarrow_\Rightarrow$)" produces this code:

$$\begin{aligned} &\text{Proof}\,((\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta)_{\text{given } q^{:(A\to A)\to B}} = q\big(\text{Proof}\,(\alpha \Rightarrow \beta \vdash \alpha \Rightarrow \alpha)_{\text{given } r^{:A\to B}}\big) \\ &\qquad \text{where } r^{:A\to B} = a^{:A} \to q(_^{:A} \to a) \\ &= q(x^{:A} \to x) \quad . \end{aligned}$$

The proof of $\alpha \Rightarrow \beta \vdash \alpha \Rightarrow \alpha$ does not actually use the intermediate value $r^{:A\to B}$ provided by the proof transformer. As a symbolic simplification step, we may simply omit the code of r. The `curryhoward` library always performs symbolic simplification after applying the LJT algorithm.

The reason the LJT algorithm terminates is that each rule replaces a given sequent by one or more sequents with simpler premises or goals.[4] This guarantees that the proof search will terminate either with a complete proof or with a sequent to which no more rules apply. An example of such a "dead-end" sequent is $\alpha \vdash \beta$ where α and β are different, unrelated propositions. In that situation, the LJT algorithm concludes that the initial sequent cannot be proved.

To *prove* that there is no proof, one needs to use methods that are beyond the scope of this book. An introduction to the required techniques is in the book "Proof and Disproof in Formal Logic" by R. Bornat (see footnote 1 on page 129).

[4]The paper `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.2618` shows that the LJT algorithm terminates by giving an explicit decreasing measure on proof trees.

D Parametricity theorem and naturality laws

Functional programming focuses on a small set of language features — the six type constructions and the nine code constructions, introduced in Sections 5.1.2 and 5.2.2. These constructions, summarized again in Tables 7.1 and D.1–D.2, create **fully parametric** programs and are sufficient to implement all design patterns of functional programming. At the same time, restricting programs to be fully parametric will enable several ways of mathematical reasoning about code. Examples of such reasoning are treating programs as mathematical values (referential transparency); deriving the most general type from code (type inference); and deriving code from type (code inference).

Additionally, all fully parametric programs will automatically satisfy certain laws derived from the so-called "parametricity theorem". The most often used laws of this kind are naturality laws for natural transformations, i.e., for functions of type $\forall A.\, F^A \to G^A$ where both $F^\bullet$ and $G^\bullet$ are functors (or both contrafunctors). Not having to verify naturality laws by hand saves a lot of time.

Other laws that follow automatically from parametricity are composition laws for functors, commutativity laws for bifunctors, and dinaturality laws for profunctors. This Appendix presents sufficient theoretical material to be able to derive all those laws.

Type construction	Scala example	Type notation
unit or "named unit"	`Unit` or `None`	$\mathbb{1}$
type parameter	`A` as in `F[A]`	A as in F^A
product type	`(A, B)`	$A \times B$
co-product type	`Either[A, B]`	$A + B$
function type	`A => B`	$A \to B$
recursive type	`Fix[F[_]]`	$\text{Fix}^{F^\bullet}$

Table D.1: The six type constructions that may be used in fully parametric programs.

The practical uses of parametricity were showcased in P. Wadler's paper[1] based on results derived by J. Reynolds.[2] The Reynolds-Wadler approach first replaces functions of type $A \to B$ by many-to-many relations between values of types A and B. Then the parametricity theorem shows that a certain law holds for relations. Since we are usually interested in deriving laws as equations rather than relations, the last step replaces all relations by functions.

To use the relational approach to parametricity, one needs to learn the techniques for working with relations, which will be unfamiliar to most programmers.[3] Only a few parametricity properties can be proved without using relations. However, a practicing programmer who only uses the *formulations* of the resulting laws will not need to learn any proofs. Adapting the parametricity theorem to the needs of FP practitioners, this Appendix will begin by summarizing the naturality laws and other properties of fully parametric code. Then Section D.3 will prove the parametricity theorem in the relational approach.

Parametricity properties only hold for fully parametric programs. This restriction prohibits, for instance, any use of GADTs ("unfunctors"), subtyping, `null` values, mutable values, multithreading, exceptions, run-time JVM reflection, or calling any external libraries that are not known to be fully parametric. Code that uses those features of Scala will not be covered by the parametricity theorem proved in this Appendix and may fail naturality laws.

[1] See https://people.mpi-sws.org/~dreyer/tor/papers/wadler.pdf and some further explanations in the blog posts https://reasonablypolymorphic.com/blog/theorems-for-free/ and https://bartoszmilewski.com/2014/09/22/ The title of the paper ("Theorems for free") is misleading. As we will see, the theorems are obtained only at the cost of restricting all code to the nine FP code constructions, which excludes most of the code in today's real-world applications.

[2] See https://people.mpi-sws.org/~dreyer/tor/papers/reynolds.pdf. This book does not use those results.

[3] Beginner-level tutorials on relations and relational parametricity are rare. One is in a paper by R. Backhouse, see https://www.researchgate.net/publication/262348393_On_a_Relation_on_Functions

Code construction	Scala example	Code notation
1. Use unit value	`()`	1
2. Use given argument	`x`	x
3. Create function	`x => expression`	$x \to expression$
4. Use function	`f(x)`	$f(x)$ or $x \triangleright f$
5. Create tuple	`(a, b)`	$a \times b$
6. Use tuple	`{ case (a, b) => ... }` or `p._1` or `p._2`	$a \times b \to ...$ or $p \triangleright \pi_1$ or $p \triangleright \pi_2$
7. Create disjunctive value	`Left[A, B](a)`	$a^{:A} + \mathbb{0}^{:B}$ or $\begin{array}{\|cc\|} a & \mathbb{0} \end{array}$

8. Use disjunctive value	`p match { case Left(a) => f(a)`
	`case Right(b) => g(b)`
	`}`
		$p \triangleright \begin{array}{\|c\|\|c\|} & C \\ \hline A & a \to f(a) \\ B & b \to g(b) \end{array}$

9. Recursive call	`def f(x) = { ... f(y) ... }`	$f(x) \stackrel{\text{def}}{=} ... \overline{f}(y) ...$

Table D.2: The nine code constructions of fully parametric programs.

An example of such code is the following function:

```
def headOptBad[A]: List[A] => Option[A] = {
  case Nil                    => None
  case (head: Int) :: tail    => Some((head + 100).asInstanceOf[A])
  case head :: tail           => Some(head)
}
```

This code has no side effects, is purely functional and referentially transparent. But it is clear that `headOptBad[A]` does not work in the same way for all type parameters `A`. When `A` is set to `Int`, the behavior is different than for all other types. This leads to a failure of the naturality law:

```
scala> headOptBad[String](  List(1, 2, 3).map(x => s"value = $x")  )
res0: Option[String] = Some(value = 1)

scala> headOptBad[Int]( List(1, 2, 3) ).map(x => s"value = $x")
res1: Option[String] = Some(value = 101)
```

So, full parametricity is a stronger restriction on code than referential transparency.

Here is an example of fully parametric code that uses all nine constructions:

```
def fmap[A, B](f: A => B): List[(A, A)] => List[(B, B)] = {      // 3
   case Nil             => Nil
//   8   1                 1,7
   case head :: tail    => (f (head._1), f (head._2)) :: fmap(f)(tail)
//   8       6              2 4     6  5 2 4     6    7   9
}  // This code uses each of the nine fully parametric constructions.
```

For instance, the expression `head :: tail` is a tuple pattern that extracts the two parts of a tuple (construction 6). The recursive call `fmap(f)(tail)` corresponds to construction 9.

D.1 Practical use of parametricity properties

D.1.1 Naturality and dinaturality laws

Parametricity properties apply to fully parametric expressions (most often, functions) with at least one type parameter. Examples of such functions are the `map`, `filter`, and `fold` methods for the `Option`

functor (denoted by $\text{Opt}^A \stackrel{\text{def}}{=} \mathbb{1} + A$), with the following type signatures:

$$\begin{aligned}
&\text{fmap}^{A,B}_{\text{Opt}} : (A \to B) \to \text{Opt}^A \to \text{Opt}^B \quad , \\
&\text{filt}^{A}_{\text{Opt}} : (A \to \mathbb{2}) \to \text{Opt}^A \to \text{Opt}^A \quad , \\
&\text{fold}^{A,B}_{\text{Opt}} : B \times (A \times B \to B) \to \text{Opt}^A \to B \quad .
\end{aligned}$$

These methods satisfy appropriate naturality laws — one law per type parameter.

When a function's type signature is that of a *natural transformation* between functors or between contrafunctors, the naturality laws have the form derived in Section 9.4.1. If $p : \forall A.\, G^A \to H^A$ where $G^\bullet$ and $H^\bullet$ are functors then p obeys the naturality law:

$$\text{for all } A, B, f^{:A\to B} \quad : \quad p^A \,⨾\, f^{\uparrow G} \,⨾\, p^B \quad . \tag{D.1}$$

If $p : \forall A.\, G^A \to H^A$ where $G^\bullet$ and $H^\bullet$ are contrafunctors then p obeys the naturality law:

$$\text{for all } A, B, f^{:A\to B} \quad : \quad f^{\downarrow G} \,⨾\, p^A = p^B \,⨾\, f^{\downarrow H} \quad . \tag{D.2}$$

For example, if we fix the type parameter A in the `fmap` method, we obtain a type signature of the form $F^B \to G^B$ where $F^\bullet$ and $G^\bullet$ are functors:

$$\text{fmap}^{A,B}_{\text{Opt}} : F^B \to G^B \quad , \qquad F^B \stackrel{\text{def}}{=} A \to B \quad , \qquad G^B \stackrel{\text{def}}{=} \text{Opt}^A \to \text{Opt}^B \quad .$$

The corresponding naturality law is:

$$\text{fmap}_{\text{Opt}} \,⨾\, f^{\uparrow G} = f^{\uparrow F} \,⨾\, \text{fmap}_{\text{Opt}} \quad .$$

Fixing the type parameter B in $\text{fold}^{A,B}_{\text{Opt}}$ produces a type signature of the form:

$$\text{fold}^{A}_{\text{Opt}} : H^A \to K^A \quad , \qquad H^A \stackrel{\text{def}}{=} B \times (A \times B \to B) \quad , \qquad K^A \stackrel{\text{def}}{=} \text{Opt}^A \to B \quad ,$$

where K and H are contrafunctors. The corresponding naturality law is:

$$f^{\downarrow H} \,⨾\, \text{fold}_{\text{Opt}} = \text{fold}_{\text{Opt}} \,⨾\, f^{\downarrow K} \quad .$$

However, fixing the type parameter A in $\text{fold}^{A,B}_{\text{Opt}}$ gives a type signature that is *not* of the form $\forall B.\, P^B \to Q^B$ (with any functors or contrafunctors P, Q) because B is used in too many places. In general, it is not obvious how to write naturality laws for transformations with such type signatures.

To make progress, note that each occurrence of a type parameter in a type signature must be either in a covariant or in a contravariant position. So, any fully parametric type signature may be written in the form $\forall A.\, P^{A,A}$ using some *profunctor* $P^{X,Y}$ (contravariant in X and covariant in Y). For instance, we can write in this way the type signature of `fold` with the parameter A fixed:

$$\text{fold}^{A,B}_{\text{Opt}} : P^{B,B} \quad , \qquad \text{where} \quad P^{X,Y} \stackrel{\text{def}}{=} X \times (A \times Y \to X) \to \text{Opt}^A \to Y \quad .$$

The profunctor $P^{X,Y}$ is complicated but the form of the type signature ($P^{B,B}$) is simpler.

Given a fully parametric expression p with the type signature $p : \forall A.\, P^{A,A}$, where $P^{X,Y}$ is some profunctor, one can write the **wedge law** of p:

$$\begin{tikzcd}
\forall Z.\, P^{Z,Z} \arrow[r, "\text{set } Z=B"] \arrow[d, "\text{set } Z=A"] & P^{B,B} \arrow[d, "f^{\downarrow P^{\bullet,B}}"] \\
P^{A,A} \arrow[r, "f^{\uparrow P^{A,\bullet}}"] & P^{A,B}
\end{tikzcd}$$

$$\text{for all } A, B, f^{:A\to B} \quad : \quad p^A \triangleright f^{\uparrow P^{A,\bullet}} = p^B \triangleright f^{\downarrow P^{\bullet,B}} \quad . \tag{D.3}$$

The diagram is read by starting from a value p of type $\forall Z.\, P^{Z,Z}$. We set $Z = A$ and $Z = B$ in p and obtain two values, p^A and p^B, of types $P^{A,A}$ and $P^{B,B}$ respectively. The values p^A and p^B must then satisfy Eq. (D.3).

We will prove later in this Appendix that the wedge law (D.3) is a consequence of the parametricity theorem. In turn, the naturality laws (D.1) and (D.2) are consequences of the wedge law, as will be shown in Statement D.1.1.3 below.

The wedge law is not easy to use in practice because the profunctors P are often complicated (as the example of `fold` shows). When we need to derive naturality laws for *function types*, it helps to specialize the wedge law to the case $P^{A,A} = G^{A,A} \to H^{A,A}$, where $G^{X,Y}$ and $H^{X,Y}$ are some profunctors. The result (also derived in Statement D.1.1.3) is the **dinaturality law**:

$$\text{for all } A, B, f^{:A\to B} \quad : \quad f^{\downarrow G^{\bullet,A}} \fatsemi p^A \fatsemi f^{\uparrow H^{A,\bullet}} = f^{\uparrow G^{B,\bullet}} \fatsemi p^B \fatsemi f^{\downarrow H^{\bullet,B}} \quad . \tag{D.4}$$

$$\begin{tikzcd} G^{A,A} \arrow[r, "p^A"] \arrow[d, "f^{\uparrow G^{B,\bullet}}"'] & H^{A,A} \arrow[d, "f^{\uparrow H^{A,\bullet}}"] \\ G^{B,A} \arrow[r] & H^{A,B} \end{tikzcd}$$

The diagram at left illustrates the dinaturality law as an equation between functions of type $G^{B,A} \to H^{A,B}$. To build up intuition for that law, notice that Eq. (D.4) combines the laws (D.1)–(D.2) in the way required for all types to match. On the other hand, the laws (D.1) and (D.2) will follow from Eq. (D.4) when $G^{A,A}$ and $H^{A,A}$ are both functors or both contrafunctors with respect to A.

Functions $p : \forall A. G^{A,A} \to H^{A,A}$ satisfying Eq. (D.4) are called **dinatural transformations**. The property of dinaturality is weaker than naturality. A *natural* transformation between profunctors G and H would be a function t with type signature $\forall (A, B). G^{A,B} \to H^{A,B}$ defined for arbitrary (not necessarily equal) type parameters A, B. If we are given a transformation $p^A : G^{A,A} \to H^{A,A}$ then, as a rule, it will be impossible to extend the code of p^A to some $t^{A,B} : G^{A,B} \to H^{A,B}$ that works with arbitrary type parameters A, B.

With the formulas (D.1)–(D.4), we can write naturality laws more quickly, starting from any given type signature of the form $\forall A. P^{A,A}$. The following examples show how we may derive naturality laws by specializing the general law (D.4) to certain profunctors G and H.

Example D.1.1.1 (naturality law of `filter`) To derive the naturality law of `filter`, express `filter`'s type signature through profunctors G and H as:

$$\text{filt}_F^A : G^{A,A} \to H^{A,A} \quad , \qquad G^{X,Y} \stackrel{\text{def}}{=} (X \to 2) \quad , \qquad H^{X,Y} \stackrel{\text{def}}{=} F^X \to F^Y \quad ,$$

and then write the law (D.4):

$$f^{\downarrow G^{\bullet,A}} \fatsemi \text{filt}_F^A \fatsemi f^{\uparrow H^{A,\bullet}} \stackrel{?}{=} f^{\uparrow G^{B,\bullet}} \fatsemi \text{filt}_F^B \fatsemi f^{\downarrow H^{\bullet,B}} \quad . \tag{D.5}$$

It remains to substitute the code for the liftings using the specific types of H and G:

$$(f^{:A\to B})^{\downarrow G^{\bullet,A}} = p^{:B\to 2} \to f \fatsemi p \quad , \qquad f^{\uparrow G^{B,\bullet}} = \text{id} \quad ,$$
$$(f^{:A\to B})^{\downarrow H^{\bullet,B}} = q^{:F^B\to F^B} \to f^{\uparrow F} \fatsemi q \quad , \qquad f^{\uparrow H^{A,\bullet}} = q^{:F^A\to F^A} \to q \fatsemi f^{\uparrow F} \quad .$$

Then we rewrite Eq. (D.5) as:

$$(p \to f \fatsemi p) \fatsemi \text{filt}_F \fatsemi (q \to q \fatsemi f^{\uparrow F}) \stackrel{?}{=} \text{id} \fatsemi \text{filt}_F \fatsemi (q \to f^{\uparrow F} \fatsemi q) \quad .$$

To simplify the form of the naturality law, apply both sides to an arbitrary $p^{:P^{B,A}} = p^{:B\to 2}$:

$$\begin{aligned}
&\text{left-hand side}: && p \triangleright (p \to f \fatsemi p) \fatsemi \text{filt}_F \fatsemi (q \to q \fatsemi f^{\uparrow F}) \\
&\triangleright\text{-notation}: && = \underline{p \triangleright (p \to f \fatsemi p)} \triangleright \text{filt}_F \triangleright (q \to q \fatsemi f^{\uparrow F}) \\
&\text{apply functions}: && = \underline{(f \fatsemi p) \triangleright \text{filt}_F} \triangleright (q \to q \fatsemi f^{\uparrow F}) = \underline{\text{filt}_F (f \fatsemi p) \triangleright (q \to q \fatsemi f^{\uparrow F})} = \text{filt}_F (f \fatsemi p) \fatsemi f^{\uparrow F} \quad , \\
&\text{right-hand side}: && p \triangleright \underline{\text{id} \fatsemi} \text{filt}_F \fatsemi (q \to f^{\uparrow F} \fatsemi q) = p \triangleright \text{filt}_F \triangleright (q \to f^{\uparrow F} \fatsemi q) \\
& && = \underline{\text{filt}_F (p) \triangleright (q \to f^{\uparrow F} \fatsemi q)} = f^{\uparrow F} \fatsemi \text{filt}_F (p) \quad .
\end{aligned}$$

We obtained the naturality law (9.3) of `filter`:

$$\text{filt}_F (f \fatsemi p) \fatsemi f^{\uparrow F} = f^{\uparrow F} \fatsemi \text{filt}_F (p) \quad .$$

Example D.1.1.2 (naturality law of `fold`) To derive the naturality law of `fold` with respect to the type parameter B, we write the type signature of `fold` as $G^{B,B} \to H^{B,B}$ with some profunctors G, H:

$$\text{fold}_F^{A,B} : G^{B,B} \to H^{B,B} \qquad \text{where} \quad G^{X,Y} \stackrel{\text{def}}{=} Y \times (A \times X \to Y) \quad \text{and} \quad H^{X,Y} \stackrel{\text{def}}{=} F^A \to Y \quad .$$

Since the type parameter A is fixed, let us write the law (D.4) with an arbitrary function $f^{:B\to C}$:

$$(f^{:B\to C})^{\downarrow G^{\bullet,B}} \,\text{⨾}\, \text{fold}_F^B \,\text{⨾}\, f^{\uparrow H^{B,\bullet}} \stackrel{!}{=} f^{\uparrow G^{C,\bullet}} \,\text{⨾}\, \text{fold}_F^C \,\text{⨾}\, f^{\downarrow H^{\bullet,C}} \quad . \tag{D.6}$$

The lifting code required for the profunctors $G^{X,Y} \stackrel{\text{def}}{=} Y \times (A \times X \to Y)$ and $H^{X,Y} \stackrel{\text{def}}{=} F^A \to Y$ is:

$$(f^{:B\to C})^{\downarrow G^{\bullet,B}} = \text{id}^B \boxtimes (h^{:A\times C\to B} \to a^{:A} \times b^{:B} \to h(a \times f(b))) \quad , \qquad f^{\uparrow G^{C,\bullet}} = f \boxtimes (h^{:A\times C\to B} \to h \,\text{⨾}\, f) \quad ,$$
$$(f^{:B\to C})^{\downarrow H^{\bullet,C}} = \text{id} \quad , \qquad f^{\uparrow H^{B,\bullet}} = q^{:F^A\to B} \to q \,\text{⨾}\, f \quad .$$

Substituting this code into the law (D.6) and applying to an arbitrary $p^{:G^{C,B}} = z^{:B} \times h^{:A\times C\to B}$, we get:

$$\begin{aligned}
\text{left-hand side :} \quad & (z \times h) \triangleright \underline{(f^{:B\to C})^{\downarrow G^{\bullet,B}}} \,\text{⨾}\, \text{fold}_F \,\text{⨾}\, \underline{f^{\uparrow H^{B,\bullet}}} \\
\text{definitions of liftings :} \quad & = (z \times h) \triangleright (\text{id} \boxtimes (h \to a \times b \to h(a \times f(b)))) \,\underline{\text{⨾}}\, \text{fold}_F \,\underline{\text{⨾}}\, (q \to q \,\text{⨾}\, f) \\
\triangleright\text{-notation :} \quad & = (z \times h) \triangleright (\text{id} \boxtimes (h \to a \times b \to h(a \times f(b)))) \triangleright \text{fold}_F \triangleright (q \to q \,\text{⨾}\, f) \\
\text{apply functions :} \quad & = \text{fold}_F (z \times (a \times b \to h(a \times f(b)))) \,\text{⨾}\, f \quad , \\
\text{right-hand side :} \quad & (z \times h) \triangleright \underline{f^{\uparrow G^{C,\bullet}}} \,\text{⨾}\, \text{fold}_F \,\text{⨾}\, \underline{f^{\downarrow H^{\bullet,C}}} = (z \times h) \triangleright (f \boxtimes (h^{:A\times C\to B} \to h \,\text{⨾}\, f)) \,\text{⨾}\, \text{fold}_F \,\text{⨾}\, \text{id} \\
\text{apply functions :} \quad & = \text{fold}_F (f(z) \times (h \,\text{⨾}\, f)) \quad .
\end{aligned}$$

We obtain the following law of `fold`:

$$\text{fold}_F (f(z) \times (h \,\text{⨾}\, f)) = \text{fold}_F (z \times (a \times b \to h(a \times f(b)))) \,\text{⨾}\, f \quad .$$

Statement D.1.1.3 The laws (D.1)–(D.4) are special cases of the wedge law (D.3).

Proof We assume that the wedge law holds for any values p of type $\forall A. P^{A,A}$.

To prove Eq. (D.1), we define $P^{X,Y} \stackrel{\text{def}}{=} G^X \to H^Y$. The two liftings of a function $f^{:A\to B}$ to P are:

$$f^{\uparrow P^{A,\bullet}} = p^{:G^A\to H^A} \to p \,\text{⨾}\, f^{\uparrow H} \quad , \qquad f^{\downarrow P^{\bullet,B}} = p^{:G^B\to H^B} \to f^{\uparrow G} \,\text{⨾}\, p \quad .$$

So, the wedge law (D.3) gives:

$$\begin{aligned}
\text{left-hand side :} \quad & p^A \triangleright f^{\uparrow P^{A,\bullet}} = p^A \,\text{⨾}\, f^{\uparrow H} \quad , \\
\text{right-hand side :} \quad & p^B \triangleright f^{\downarrow P^{\bullet,B}} = f^{\uparrow G} \,\text{⨾}\, p^B \quad .
\end{aligned}$$

We obtain $p^A \,\text{⨾}\, f^{\uparrow H} = f^{\uparrow G} \,\text{⨾}\, p^B$, which is the same as Eq. (D.1).

To prove Eq. (D.2), we define $P^{X,Y} \stackrel{\text{def}}{=} G^Y \to H^X$. The two liftings of a function $f^{:A\to B}$ to P are:

$$f^{\uparrow P^{A,\bullet}} = p^{:G^A\to H^A} \to f^{\downarrow G} \,\text{⨾}\, p \quad , \qquad f^{\downarrow P^{\bullet,B}} = p^{:G^B\to H^B} \to p \,\text{⨾}\, f^{\downarrow H} \quad .$$

So, the wedge law (D.3) gives:

$$\begin{aligned}
\text{left-hand side :} \quad & p^A \triangleright f^{\uparrow P^{A,\bullet}} = f^{\downarrow G} \,\text{⨾}\, p^A \quad , \\
\text{right-hand side :} \quad & p^B \triangleright f^{\downarrow P^{\bullet,B}} = p^B \,\text{⨾}\, f^{\downarrow H} \quad .
\end{aligned}$$

We obtain $f^{\downarrow G} \,\text{⨾}\, p^A = p^B \,\text{⨾}\, f^{\downarrow H}$, which is the same as Eq. (D.2).

To prove Eq. (D.4), we define $P^{X,Y} \stackrel{\text{def}}{=} G^{Y,X} \to H^{X,Y}$. Note that we need to swap X and Y in $G^{Y,X}$ in order to conform to the required variance of $P^{X,Y}$ (contravariant in X and covariant in Y).

The two liftings of a function $f^{:A\to B}$ to P are expressed as:

$$f^{\uparrow P^{A,\bullet}} = p^{:G^{A,A}\to H^{A,A}} \to f^{\downarrow G^{\bullet,A}} \fatsemi p \fatsemi f^{\uparrow H^{A,\bullet}} \quad , \qquad f^{\downarrow P^{\bullet,B}} = p^{:G^{B,B}\to H^{B,B}} \to f^{\uparrow G^{B,\bullet}} \fatsemi p \fatsemi f^{\downarrow H^{\bullet,B}} \quad .$$

So, the wedge law (D.3) gives:

$$\begin{aligned} \text{left-hand side}: \quad & p^A \triangleright f^{\uparrow P^{A,\bullet}} = f^{\downarrow G^{\bullet,A}} \fatsemi p^A \fatsemi f^{\uparrow H^{A,\bullet}} \quad , \\ \text{right-hand side}: \quad & p^B \triangleright f^{\downarrow P^{\bullet,B}} = f^{\uparrow G^{B,\bullet}} \fatsemi p^B \fatsemi f^{\downarrow H^{\bullet,B}} \quad . \end{aligned}$$

We obtain $f^{\downarrow G^{\bullet,A}} \fatsemi p^A \fatsemi f^{\uparrow H^{A,\bullet}} = f^{\uparrow G^{B,\bullet}} \fatsemi p^B \fatsemi f^{\downarrow H^{\bullet,B}}$, which is the same as Eq. (D.4). □

D.1.2 Uniqueness of functor and contrafunctor liftings

All the laws shown here need to use function liftings such as $f^{\uparrow G}$ or $f^{\uparrow P^{A,\bullet}}$. How are these liftings defined? Sections 6.2.3 and 6.2.4 derived lawful and fully parametric implementations of the `fmap` and `cmap` methods for all functors and contrafunctors as long as they are built up from the six type constructions. The naturality laws obtained from the parametricity theorem must use precisely those "standard" implementations of `fmap` and `cmap` because the proof of the parametricity theorem significantly depends on the code of those implementations.

The structure of a given fully parametric type constructor F^A dictates a unique implementation of a lifting $f^{\uparrow P}$ or $f^{\downarrow P}$. Let us summarize these implementations for the six type constructions:

Constant type If $F^A \stackrel{\text{def}}{=} Z$ where Z is a fixed type then $f^{\uparrow F} = \text{id}^{:Z\to Z}$.

Type parameter If $F^A \stackrel{\text{def}}{=} A$ then $f^{\uparrow F} = f$. If $F^A \stackrel{\text{def}}{=} G^{H^A}$ then $f^{\uparrow F} = (f^{\uparrow H})^{\uparrow G}$ if both G and H are functors and $f^{\uparrow F} = (f^{\downarrow H})^{\downarrow G}$ if both G and H are contrafunctors.

Products If $F^A \stackrel{\text{def}}{=} G^A \times H^A$ then $f^{\uparrow F} = f^{\uparrow G} \boxtimes f^{\uparrow H}$.

Co-products If $F^A \stackrel{\text{def}}{=} G^A + H^A$ then $f^{\uparrow F} = f^{\uparrow G} \boxplus f^{\uparrow H}$.

Function types If $F^A \stackrel{\text{def}}{=} G^A \to H^A$ then $f^{\uparrow F} = p^{:G^A\to H^A} \to f^{\downarrow G} \fatsemi p \fatsemi f^{\uparrow H}$.

Recursive type If $F^A \stackrel{\text{def}}{=} S^{A,F^A}$ then $f^{\uparrow F} = f^{\uparrow S^{\bullet,F^A}} \fatsemi (\overline{f^{\uparrow F}})^{\uparrow S^{B,\bullet}}$. Here $\overline{f^{\uparrow F}}$ is a recursive call to $f^{\uparrow F}$.

In addition, we will need a seventh type construction: the universally quantified type.

Quantified type If $F^A \stackrel{\text{def}}{=} \forall X. P^{X,A}$ where $P^{X,A}$ is covariant in A then:

$$f^{\uparrow F} = \forall Y. p^{:\forall X. P^{X,A}} \to p^Y \triangleright f^{\uparrow P^{Y,\bullet}} \quad .$$

It turns out that there no other lawful implementations of `fmap` for these type constructions:

Statement D.1.2.1 There is only one implementation of a given functor F's `fmap` method that satisfies the identity, composition, and naturality laws. In particular, fully parametric lawful implementations of `fmap` are unique.

Proof Section 6.2.3 derived lawful implementations of the `fmap` method for all functors F built up via the six type constructions. Throughout this book, this standard lifting code is denoted by $\text{fmap}_F(f)$ or by $f^{\uparrow F}$. Now suppose that there exists *another* lawful implementation of `fmap` for F, denoted by $\text{fmap}'_F(f)$:

$$\text{fmap}'_F : (A \to B) \to F^A \to F^B \quad , \qquad \text{fmap}'_F(f^{:A\to B}) = ???^{:F^A\to F^B} \quad .$$

We will now show that $\text{fmap}'_F = \text{fmap}_F$. Let us fix the type parameter A and apply the naturality law to fmap'_F with respect to B. The resulting law involves an arbitrary $g^{:B\to C}$:

$$\text{fmap}'_F(f^{:A\to B} \fatsemi g^{:B\to C}) \stackrel{!}{=} \text{fmap}'_F(f) \fatsemi g^{\uparrow F} \quad .$$

By definition of naturality laws, the lifting $g^{\uparrow F}$ must use the "standard" lifting code $g^{\uparrow F} \stackrel{\text{def}}{=} \text{fmap}_F(g)$. By assumption, fmap'_F obeys the composition law, so we may write:

$$\text{fmap}'_F(f \fatsemi g) = \text{fmap}'_F(f) \fatsemi \text{fmap}'_F(g) \stackrel{!}{=} \text{fmap}'_F(f) \fatsemi g^{\uparrow F} \quad .$$

Since $f^{:A\to B}$ is arbitrary, we can choose $A = B$ and $f = \text{id}^{:B\to B}$ to obtain:

$$\text{fmap}'_F(\text{id}) \fatsemi \text{fmap}'_F(g) \overset{!}{=} \text{fmap}'_F(\text{id}) \fatsemi g^{\uparrow F} \quad .$$

The identity law for fmap'_F gives $\text{fmap}'_F(\text{id}) = \text{id}$, so we can simplify the last equation to:

$$\text{fmap}'_F(g) \overset{!}{=} g^{\uparrow F} = \text{fmap}_F(g) \quad .$$

This must hold for arbitrary $g^{:B\to C}$, which proves that $\text{fmap}'_F = \text{fmap}_F$.

Statement D.1.2.2 Any contrafunctor H has a unique implementation of a lawful `cmap` method.

Proof We use similar arguments as in the proof of Statement D.1.2.1. For any lawful alternative implementation cmap'_H, the naturality law is:

$$\text{cmap}'_H(f^{:A\to B} \fatsemi g^{:B\to C}) \overset{!}{=} (g^{:B\to C})^{\downarrow H} \fatsemi \text{cmap}'_H(f) \quad .$$

By assumption, the identity and composition law hold for cmap'_H. Setting $f = \text{id}^{:B\to B}$, we get:

$$\text{cmap}'_H(\text{id} \fatsemi g) = \text{cmap}'_H(g) \overset{!}{=} g^{\downarrow H} \fatsemi \text{cmap}'_H(\text{id}) = g^{\downarrow H} \quad .$$

This must hold for arbitrary $g^{:B\to C}$, which shows that $\text{cmap}'_H(g) = g^{\downarrow H} = \text{cmap}_H(g)$ as required. □

In this section, we proved that lawful implementations of `fmap` and `cmap` are unique when they are fully parametric. However, this does not yet prove that any implementation of `fmap` and `cmap` satisfying only the functor laws must also be fully parametric (Problem 15.3.2.13).

D.2 Commutativity laws for bifunctors and profunctors

A special property that holds as a consequence of parametricity is the bifunctor commutativity law (6.9) introduced in Section 6.2.2. If a type constructor $P^{A,B}$ is a functor with respect to A and B separately then the liftings with respect to the two type parameters will commute:

$$\begin{tikzcd} P^{A,B} \arrow[r, "f^{\uparrow P^{\bullet,B}}"] \arrow[d, "g^{\uparrow P^{A,\bullet}}"] & P^{C,B} \arrow[d, "g^{\uparrow P^{C,\bullet}}"'] \\ P^{A,D} \arrow[r, "f^{\uparrow P^{\bullet,D}}"] & P^{C,D} \end{tikzcd}$$

$$\text{for all } A, B, C, D, f^{:A\to C}, g^{:B\to D} \quad : \quad f^{\uparrow P^{\bullet,B}} \fatsemi g^{\uparrow P^{C,\bullet}} = g^{\uparrow P^{A,\bullet}} \fatsemi f^{\uparrow P^{\bullet,D}} \quad .$$

Similar properties hold for type constructors with any number of type parameters that are either covariant or contravariant with respect to each of those type parameters.

We will now prove that the commutativity law holds for any fully parametric bifunctors and profunctors. The proof proceeds by induction on the type structure of a given type constructor.[4] It turns out that the proof also requires the commutativity laws for type constructors with *contravariant* type parameters. All those proofs are completely analogous, so we will first prove the commutativity law for profunctors, which are type constructors with one contravariant and one covariant type parameter. The profunctor commutativity law will be used below in Statement D.3.5.1.

D.2.1 Proof of the profunctor commutativity law

Given an arbitrary type constructor $P^{A,B}$ contravariant with respect to A and covariant with respect to B, we formulate the profunctor **commutativity law** by analogy with the bifunctor law (6.9):

$$\begin{aligned} &\text{commutativity law of } P: \quad \text{cmap}_{P^{\bullet,B}}(f^{:A\to C}) \fatsemi \text{fmap}_{P^{A,\bullet}}(g^{:B\to D}) = \text{fmap}_{P^{C,\bullet}}(g) \fatsemi \text{cmap}_{P^{\bullet,D}}(f) \quad , \\ &\text{in a shorter notation}: \quad f^{\downarrow P^{\bullet,B}} \fatsemi g^{\uparrow P^{A,\bullet}} = g^{\uparrow P^{C,\bullet}} \fatsemi f^{\downarrow P^{\bullet,D}} \quad . \end{aligned} \tag{D.7}$$

[4] In the blog post https://byorgey.wordpress.com/2018/03/30/, Brent Yorgey gave a proof of the commutativity law for bifunctors based on the Reynolds-Wadler relational parametricity theorem.

$$\begin{tikzcd}
P^{C,B} \arrow[r, "\text{cmap}_{P^{\bullet,B}}(f^{:A\to C})"] \arrow[d, "\text{fmap}_{P^{C,\bullet}}(g)"'] & P^{A,B} \arrow[d, "\text{fmap}_{P^{A,\bullet}}(g^{:B\to D})"] \\
P^{C,D} \arrow[r, "\text{cmap}_{P^{\bullet,D}}(f)"'] & P^{A,D}
\end{tikzcd}$$

We will now prove that Eq. (D.7) holds for any **fully parametric profunctor** $P^{A,B}$, i.e., a profunctor built via the six type constructions from Table D.1. We will assume that all liftings such as $f^{\downarrow P^{\bullet,B}}$ or $g^{\uparrow P^{A,\bullet}}$ are implemented as shown in Chapter 6 for functors and contrafunctors. (Section D.1.2 showed that the code from Chapter 6 is the only possible lawful and fully parametric implementation of the liftings.)

The proof goes by induction on the type structure of $P^{A,B}$. The base case contains the first two constructions (the unit type and the type parameter), which gives two possibilities: $P^{A,B} \stackrel{\text{def}}{=} \mathbb{1}$ (constant profunctor) and $P^{A,B} \stackrel{\text{def}}{=} B$ (identity profunctor). The other possibility of using a type parameter, $P^{A,B} \stackrel{\text{def}}{=} A$, does not give a profunctor since $P^{A,B}$ must be contravariant in A.

The commutativity law holds for $P^{A,B} \stackrel{\text{def}}{=} \mathbb{1}$ because all liftings are identity functions: $\text{cmap}_{P^{\bullet,B}}(f) = \text{id}$ and $\text{fmap}_{P^{A,\bullet}}(g) = \text{id}$. For the same reason, the law will also hold for constant profunctors $P^{A,B} \stackrel{\text{def}}{=} Z$ where Z is a fixed type (or a type parameter other than A or B).

For the profunctor $P^{A,B} \stackrel{\text{def}}{=} B$, the law holds because $\text{cmap}_{P^{\bullet,B}}(f) = \text{id}$. So, the base case is proved.

The inductive step looks at the outer construction in the type expression of $P^{A,B}$, which must be one of product, co-product, function, or recursion. These constructions create larger type expressions out of smaller ones; for example, $P^{A,B} \stackrel{\text{def}}{=} (A \to B) \times B$ uses the product construction with sub-expressions $A \to B$ and B, which can be viewed as profunctors $R^{A,B} \stackrel{\text{def}}{=} A \to B$ and $S^{A,B} \stackrel{\text{def}}{=} B$. The inductive assumption is that the commutativity law is already proved for all sub-expression types (such as $R^{A,B}$ or $S^{A,B}$). We will then need to prove the law for the entire type expression $P^{A,B}$.

Product type If the outer level of $P^{A,B}$ is a product, we must have $P^{A,B} = R^{A,B} \times S^{A,B}$ where $R^{A,B}$ and $S^{A,B}$ are some profunctors. The code for liftings follows from Statement 6.2.3.3 and Exercise 6.3.1.2:

$$f^{\downarrow P^{\bullet,B}} \stackrel{\text{def}}{=} f^{\downarrow R^{\bullet,B}} \boxtimes f^{\downarrow S^{\bullet,B}} \quad , \qquad g^{\uparrow P^{A,\bullet}} \stackrel{\text{def}}{=} g^{\uparrow R^{A,\bullet}} \boxtimes g^{\uparrow S^{A,\bullet}} \quad .$$

By inductive assumption, R and S already satisfy the commutativity law (D.7). So, we compute:

$$\begin{aligned}
\text{expect to equal } g^{\uparrow P^{C,\bullet}} \fatsemi f^{\downarrow P^{\bullet,D}} : \quad & \underline{f^{\downarrow P^{\bullet,B}}} \fatsemi \underline{g^{\uparrow P^{A,\bullet}}} = \left(f^{\downarrow R^{\bullet,B}} \boxtimes f^{\downarrow S^{\bullet,B}}\right) \fatsemi \left(g^{\uparrow R^{A,\bullet}} \boxtimes g^{\uparrow S^{A,\bullet}}\right) \\
\text{composition law (7.2)} : \quad & = \left(\underline{f^{\downarrow R^{\bullet,B}} \fatsemi g^{\uparrow R^{A,\bullet}}}\right) \boxtimes \left(\underline{f^{\downarrow S^{\bullet,B}} \fatsemi g^{\uparrow S^{A,\bullet}}}\right) \\
\text{inductive assumption} : \quad & = \left(g^{\uparrow R^{C,\bullet}} \fatsemi f^{\downarrow R^{\bullet,D}}\right) \boxtimes \left(g^{\uparrow S^{C,\bullet}} \fatsemi f^{\downarrow S^{\bullet,D}}\right) \\
\text{composition law (7.2)} : \quad & = \left(\underline{g^{\uparrow R^{C,\bullet}} \boxtimes g^{\uparrow S^{C,\bullet}}}\right) \fatsemi \left(\underline{f^{\downarrow R^{\bullet,D}} \boxtimes f^{\downarrow S^{\bullet,D}}}\right) = g^{\uparrow P^{C,\bullet}} \fatsemi f^{\downarrow P^{\bullet,D}} \quad .
\end{aligned}$$

Co-product type In this case, we must have $P^{A,B} = R^{A,B} + S^{A,B}$ where $R^{A,B}$ and $S^{A,B}$ are some profunctors. The code for liftings follows from Statement 6.2.3.4 and Exercise 6.3.1.2:

$$f^{\downarrow P^{\bullet,B}} \stackrel{\text{def}}{=} f^{\downarrow R^{\bullet,B}} \boxplus f^{\downarrow S^{\bullet,B}} = \left\| \begin{array}{cc} f^{\downarrow R^{\bullet,B}} & \mathbb{0} \\ \mathbb{0} & f^{\downarrow S^{\bullet,B}} \end{array} \right| \quad , \qquad g^{\uparrow P^{A,\bullet}} \stackrel{\text{def}}{=} g^{\uparrow R^{A,\bullet}} \boxplus g^{\uparrow S^{A,\bullet}} = \left\| \begin{array}{cc} g^{\uparrow R^{A,\bullet}} & \mathbb{0} \\ \mathbb{0} & g^{\uparrow S^{A,\bullet}} \end{array} \right| \quad .$$

By inductive assumption, R and S already satisfy the commutativity law (D.7). So, we compute:

$$\begin{aligned}
\text{expect to equal } g^{\uparrow P^{C,\bullet}} \fatsemi f^{\downarrow P^{\bullet,D}} : \quad & f^{\downarrow P^{\bullet,B}} \fatsemi g^{\uparrow P^{A,\bullet}} = \left\| \begin{array}{cc} f^{\downarrow R^{\bullet,B}} & \mathbb{0} \\ \mathbb{0} & f^{\downarrow S^{\bullet,B}} \end{array} \right| \fatsemi \left\| \begin{array}{cc} g^{\uparrow R^{A,\bullet}} & \mathbb{0} \\ \mathbb{0} & g^{\uparrow S^{A,\bullet}} \end{array} \right| \\
\text{matrix composition} : \quad & = \left\| \begin{array}{cc} \underline{f^{\downarrow R^{\bullet,B}} \fatsemi g^{\uparrow R^{A,\bullet}}} & \mathbb{0} \\ \mathbb{0} & \underline{f^{\downarrow S^{\bullet,B}} \fatsemi g^{\uparrow S^{A,\bullet}}} \end{array} \right| \\
\text{inductive assumption} : \quad & = \left\| \begin{array}{cc} g^{\uparrow R^{C,\bullet}} \fatsemi f^{\downarrow R^{\bullet,D}} & \mathbb{0} \\ \mathbb{0} & g^{\uparrow S^{C,\bullet}} \fatsemi f^{\downarrow S^{\bullet,D}} \end{array} \right| \quad .
\end{aligned}$$

The right-hand side of the law, $g^{\uparrow P^{C,\bullet}} \fatsemi f^{\downarrow P^{\bullet,D}}$, can be rewritten as:

$$g^{\uparrow P^{C,\bullet}} \fatsemi f^{\downarrow P^{\bullet,D}} = \left\| \begin{matrix} g^{\uparrow R^{C,\bullet}} & \mathbb{0} \\ \mathbb{0} & g^{\uparrow S^{C,\bullet}} \end{matrix} \right| \fatsemi \left\| \begin{matrix} f^{\downarrow R^{\bullet,D}} & \mathbb{0} \\ \mathbb{0} & f^{\downarrow S^{\bullet,D}} \end{matrix} \right|$$

$$\text{matrix composition :} \quad = \left\| \begin{matrix} g^{\uparrow R^{C,\bullet}} \fatsemi f^{\downarrow R^{\bullet,D}} & \mathbb{0} \\ \mathbb{0} & g^{\uparrow S^{C,\bullet}} \fatsemi f^{\downarrow S^{\bullet,D}} \end{matrix} \right| \quad .$$

This coincides with the left-hand side of the law.

Function type The type expression $P^{A,B} \stackrel{\text{def}}{=} R^{B,A} \to S^{A,B}$ (note the swapped type parameters in $R^{B,A}$) is contravariant in A and covariant in B when $R^{A,B}$ and $S^{A,B}$ are profunctors. The lifting code follows from Statement 6.2.3.5 and Exercise 6.3.1.1:

$$(f^{:A\to C})^{\downarrow P^{\bullet,B}} \stackrel{\text{def}}{=} p^{:P^{C,B}} \to f^{\uparrow R^{B,\bullet}} \fatsemi p \fatsemi f^{\downarrow S^{\bullet,B}} \quad , \qquad (g^{:B\to D})^{\uparrow P^{A,\bullet}} \stackrel{\text{def}}{=} p^{:P^{A,B}} \to g^{\downarrow R^{\bullet,A}} \fatsemi p \fatsemi g^{\uparrow S^{A,\bullet}} \quad .$$

By the inductive assumptions, R and S obey the commutativity law (D.7). Omitting types, we write:

$$\begin{aligned} \text{left-hand side :} \quad & f^{\downarrow P} \fatsemi g^{\uparrow P} = \big(p \to f^{\uparrow R} \fatsemi p \fatsemi f^{\downarrow S}\big) \fatsemi \big(p \to g^{\downarrow R} \fatsemi p \fatsemi g^{\uparrow S}\big) \\ \text{compute composition :} \quad & = p \to \underline{g^{\downarrow R} \fatsemi f^{\uparrow R}} \fatsemi p \fatsemi \underline{f^{\downarrow S} \fatsemi g^{\uparrow S}} \\ \text{inductive assumption :} \quad & = p \to f^{\uparrow R} \fatsemi g^{\downarrow R} \fatsemi p \fatsemi g^{\uparrow S} \fatsemi f^{\downarrow S} \quad . \end{aligned}$$

The right-hand side of the law is then transformed to the same expression:

$$\begin{aligned} & g^{\uparrow P} \fatsemi f^{\downarrow P} = \big(p \to g^{\downarrow R} \fatsemi p \fatsemi g^{\uparrow S}\big) \fatsemi \big(p \to f^{\uparrow R} \fatsemi p \fatsemi f^{\downarrow S}\big) \\ \text{compute composition :} \quad & = p \to f^{\uparrow R} \fatsemi g^{\downarrow R} \fatsemi p \fatsemi g^{\uparrow S} \fatsemi f^{\downarrow S} \quad . \end{aligned}$$

Recursive type A recursive profunctor P is defined using a type equation of the form:

$$P^{A,B} \stackrel{\text{def}}{=} S^{A,B,P^{A,B}} \quad ,$$

where $S^{A,B,R}$ must be contravariant in A but covariant in B and R. The liftings are defined by:

$$(f^{:A\to C})^{\downarrow P^{\bullet,B}} \stackrel{\text{def}}{=} f^{\downarrow S^{\bullet,B,P^{C,B}}} \fatsemi \big(f^{\downarrow \overline{P^{\bullet,B}}}\big)^{\uparrow S^{A,B,\bullet}} \quad , \qquad (g^{:B\to D})^{\uparrow P^{A,\bullet}} \stackrel{\text{def}}{=} g^{\uparrow S^{A,\bullet,P^{A,B}}} \fatsemi \big(g^{\overline{\uparrow P^{A,\bullet}}}\big)^{\uparrow S^{A,D,\bullet}} \quad .$$

There are two inductive assumptions. First, the recursively used lifting to $P^{A,B}$ already satisfies the commutativity law. Second, the profunctor $S^{A,B,R}$ satisfies the commutativity law with respect to *any* two parameters (we will prove that law for covariant type parameters in Section D.2.2).

Both sides of the commutativity law are functions of type $P^{C,B} \to P^{A,D}$ or equivalently (if we expand the type recursion) of type $S^{C,B,P^{C,B}} \to S^{A,D,P^{A,D}}$. To avoid mistakes, we write out the type parameters in this computation:

$$\begin{aligned} \text{expect to equal } g^{\uparrow P^{C,\bullet}} \fatsemi f^{\downarrow P^{\bullet,D}} : \quad & f^{\downarrow P^{\bullet,B}} \fatsemi g^{\uparrow P^{A,\bullet}} = f^{\downarrow S^{\bullet,B,P^{C,B}}} \fatsemi \underline{\big(f^{\downarrow \overline{P^{\bullet,B}}}\big)^{\uparrow S^{A,B,\bullet}} \fatsemi g^{\uparrow S^{A,\bullet,P^{A,B}}}} \fatsemi \big(g^{\overline{\uparrow P^{A,\bullet}}}\big)^{\uparrow S^{A,D,\bullet}} \\ {}^{B,R}\text{-commutativity of } S^{A,B,R} : \quad & = \underline{f^{\downarrow S^{\bullet,B,P^{C,B}}} \fatsemi g^{\uparrow S^{A,\bullet,P^{C,B}}}} \fatsemi \big(f^{\downarrow \overline{P^{\bullet,B}}}\big)^{\uparrow S^{A,D,\bullet}} \fatsemi \big(g^{\overline{\uparrow P^{A,\bullet}}}\big)^{\uparrow S^{A,D,\bullet}} \\ {}^{A,B}\text{-commutativity of } S^{A,B,R} : \quad & = g^{\uparrow S^{C,\bullet,P^{C,B}}} \fatsemi f^{\downarrow S^{\bullet,D,P^{C,B}}} \fatsemi \big(\underline{f^{\downarrow \overline{P^{\bullet,B}}} \fatsemi g^{\overline{\uparrow P^{A,\bullet}}}}\big)^{\uparrow S^{A,D,\bullet}} \\ \text{inductive assumption :} \quad & = g^{\uparrow S^{C,\bullet,P^{C,B}}} \fatsemi f^{\downarrow S^{\bullet,D,P^{C,B}}} \fatsemi \big(g^{\overline{\uparrow P^{C,\bullet}}} \fatsemi \underline{f^{\downarrow \overline{P^{\bullet,D}}}}\big)^{\uparrow S^{A,D,\bullet}} \\ \text{composition under } {}^{\uparrow S^{A,D,\bullet}} : \quad & = g^{\uparrow S^{C,\bullet,P^{C,B}}} \fatsemi \underline{f^{\downarrow S^{\bullet,D,P^{C,B}}} \fatsemi \big(g^{\overline{\uparrow P^{C,\bullet}}}\big)^{\uparrow S^{A,D,\bullet}}} \fatsemi \big(f^{\overline{\downarrow P^{\bullet,D}}}\big)^{\uparrow S^{A,D,\bullet}} \\ {}^{A,R}\text{-commutativity of } S^{A,B,R} : \quad & = g^{\uparrow S^{C,\bullet,P^{C,B}}} \fatsemi \big(g^{\overline{\uparrow P^{C,\bullet}}}\big)^{\uparrow S^{C,D,\bullet}} \fatsemi f^{\downarrow S^{\bullet,D,P^{C,D}}} \fatsemi \big(f^{\overline{\downarrow P^{\bullet,D}}}\big)^{\uparrow S^{A,D,\bullet}} \\ \text{definitions of liftings :} \quad & = g^{\uparrow P^{C,\bullet}} \fatsemi f^{\downarrow P^{\bullet,D}} \quad . \end{aligned}$$

This concludes the proof of the profunctor commutativity law.

D.2.2 Commutativity laws for bifunctors and bi-contrafunctors

A bi-contrafunctor $P^{A,B}$ is a type constructor contravariant with respect to both A and B. The commutativity law for bi-contrafunctors is formulated as:

$$\begin{aligned} \text{commutativity law of } P: &\quad \text{cmap}_{P^{\bullet,D}}(f^{:A\to C}) \fatsemi \text{cmap}_{P^{A,\bullet}}(g^{:B\to D}) = \text{cmap}_{P^{C,\bullet}}(g) \fatsemi \text{cmap}_{P^{\bullet,B}}(f) \quad . \\ \text{shorter notation}: &\quad f^{\downarrow P^{\bullet,D}} \fatsemi g^{\downarrow P^{A,\bullet}} = g^{\downarrow P^{C,\bullet}} \fatsemi f^{\downarrow P^{\bullet,B}} \quad . \end{aligned} \tag{D.8}$$

$$\begin{tikzcd} P^{C,D} \arrow[r, "\text{cmap}_{P^{\bullet,D}}(f^{:A\to C})"] \arrow[d, "\text{cmap}_{P^{C,\bullet}}(g^{:B\to D})"'] & P^{A,D} \arrow[d, "\text{cmap}_{P^{A,\bullet}}(g)"] \\ P^{C,B} \arrow[r, "\text{cmap}_{P^{\bullet,B}}(f^{:A\to C})"'] & P^{A,B} \end{tikzcd}$$

Any bifunctor or bi-contrafunctor whose type expression is built up using the six type constructions (Table D.1) will satisfy its commutativity law. To prove that, we need to repeat the same calculations as for profunctors in Section D.2.1 except for changing `cmap` into `fmap` or back when needed. We only need to check that the proof will still work after such changes. Looking over the proof in Section D.2.1, we find that we never used the functor or contrafunctor *composition law* for P. The only usage of the composition law was with respect to the type parameter R in the structure functor $S^{A,B,R}$ of the recursive type construction. However, in all cases $S^{A,B,R}$ needs to be *covariant* in R because that is required by the recursive type equation. So, we are assured that the same proof with minor changes will work for bifunctors and bi-contrafunctors.

The same techniques and proofs apply to type constructors with more than two type parameters.

D.3 Relational formulation of parametricity

Naturality laws are formulated using arbitrary functions $f^{:A\to B}$ between arbitrary types A and B. Typically, a naturality law is an equation that involves the function f lifted to some functors. For instance, $f^{\uparrow G} \fatsemi t = t \fatsemi f^{\uparrow H}$ is the naturality law of a natural transformation $t : \forall A.\, G^A \to H^A$, where G and H are some functors.

To prove that naturality laws hold for any fully parametric transformation t, we need to use induction in the structure of the code of t. The proof will decompose t into smaller sub-expressions for which the naturality law should hold by the inductive assumption. Some of those sub-expressions will have types that are no longer of the form $\forall A.\, G^A \to H^A$. So, we need to generalize naturality laws to type signatures of the form $\forall A.\, P^{A,A}$, where $P^{X,Y}$ is an exponential-polynomial profunctor (contravariant in X and covariant in Y).

It is not possible to lift an arbitrary function $f^{:A\to B}$ to a function of type $P^{A,A} \to P^{B,B}$; for that, we would need also a function of type $B \to A$. To solve this problem, the Reynolds-Wadler approach replaces functions $f^{:A\to B}$ by arbitrary many-to-many relations (denoted by $r^{:A\leftrightarrow B}$) between values of types A and B. It turns out that any relation r of type $A \leftrightarrow B$ can be lifted to a relation (denoted by $r^{\updownarrow P}$) of type $P^{A,A} \leftrightarrow P^{B,B}$. The lifting operation can be defined for any exponential-polynomial profunctor $P^{\bullet,\bullet}$. Using this technique, we will prove the "relational parametricity theorem": any fully parametric code expression (not necessarily a function) of type $\forall A.\, P^{A,A}$ satisfies a specially formulated relational naturality law. That law will then allow us to derive the wedge law (D.3).

D.3.1 Relations between values of different types

Programmers are familiar with relations as tables in relational databases. The simplest relation is a table with two columns having values of some fixed types, say `INT` and `FLOAT`. As an example, consider a table called `R` containing this data:

$R =$

SAMPLE_COUNT: INT	MEAN_VALUE: FLOAT
150	0.92
150	0.95
180	0.95
180	1.02
200	0.95

Each row of the table R has a value of type INT and a value of type FLOAT (we disallow NULL values here). The existence of a row $(180, 0.95)$ means that the values 180 and 0.95 are "in the relation" R.

Two values $a^{:\text{Int}}$ and $b^{:\text{Float}}$ are in the relation R only if there exists a row (a, b) in the table R. All other pairs of values (a, b) are *not* in the relation R.

Note that the table R contains several values in the first column that correspond to the same value in the second column, and vice versa. So, the relation R is many-to-many and cannot be represented by a function of type `Int => Float` or of type `Float => Int`. Instead, we must view the relation R as a subset of the set of *all* possible pairs $(a^{:\text{Int}}, b^{:\text{Float}})$. The table R lists the pairs in that subset.

Instead of listing the pairs, we may describe a relation by implementing a function telling us whether two given values are in the relation. The Scala type signature for such a function could be:

```
def inRelationR(a: Int, b: Float): Boolean
```

In the short code notation, this function is written as $r : \text{Int} \times \text{Float} \to \mathbb{2}$. The function call $r(a \times b)$ returns `true` when the two given values a and b are in the relation r.

For proving the parametricity theorem, we will need relations between values of arbitrary types. Replacing `Int` and `Float` by type parameters A and B, we get the following definition:

Definition D.3.1.1 A **relation** of type $A \leftrightarrow B$ is a function $r : A \times B \to \mathbb{2}$. The type notation $A \leftrightarrow B$ indicates that relations are more general than functions of types $A \to B$ or $B \to A$. If values x and y are in a relation r, we write $(x, y) \in r$ or with full type annotations: $(x^{:A}, y^{:B}) \in r^{:A \leftrightarrow B}$. The condition $(x, y) \in r$ is equivalent to writing $r(x, y) = \text{true}$.

Defined in this sense, relations hold between values of given types. Certain values $x^{:A}$ and $y^{:B}$ will be in a given relation $r^{:A \leftrightarrow B}$, while other values of the same types will not be in the relation r. This should not be confused with *type relations* described in Section 8.5.5, where certain *types* will be in a given type relation while other types will not be in that relation. Since we will not be using any type relations in this Appendix, we will write simply "relations between types A and B" rather than "relations between *values of* types A and B". □

A simple example of a relation is the **identity relation**, denoted by $\text{id}^{:A \leftrightarrow A}$. The identity relation holds only when two values of type A are equal:

$$(x^{:A}, y^{:A}) \in \text{id}^{:A \leftrightarrow A} \quad \text{means} \quad x = y \quad .$$

To use the parametricity theorem in practice, we will need to convert functions into relations. The relation of type $A \leftrightarrow B$ obtained from a function $f^{:A \to B}$ is denoted by $\langle f \rangle$ and is defined as:

$$(a^{:A}, b^{:B}) \in \langle f \rangle \text{ means } b = f(a) \quad \text{or equivalently :} \quad a \triangleright f = b \quad .$$

Functions f of type $A \to B$ are converted to many-to-one relations $\langle f \rangle$ between types A and B: one or more values of type A are related to a single value of type B.

A function g of type $B \to A$ can be also converted to a relation of type $A \leftrightarrow B$. However, $\langle g \rangle$ will have type $B \leftrightarrow A$ in our notation. We note that the relation types $A \leftrightarrow B$ and $B \leftrightarrow A$ are equivalent (isomorphic as types) because we can swap $x^{:A}$ and $y^{:B}$ in the condition $(x, y) \in r^{:A \leftrightarrow B}$ and obtain the **reverse relation** of type $B \leftrightarrow A$, denoted $\text{rev}\,(r)$, such that $(y, x) \in \text{rev}\,(r)$. So, we will write $\text{rev}\,\langle g \rangle$ if we need to convert a function $g^{:B \to A}$ into a relation of type $A \leftrightarrow B$.

The operation `rev` it is its own inverse, and so it is an isomorphism between $A \leftrightarrow B$ and $B \leftrightarrow A$:

$$\text{rev}\,(\text{rev}\,(r)) = r \quad .$$

To prove this, substitute the definition of `rev` twice:

$$(x^{:A}, y^{:B}) \in \text{rev}\,\big(\text{rev}\,(r)\big) \quad \text{is the same as :} \quad (y, x) \in \text{rev}\,(r) \quad \text{and is the same as :} \quad (x, y) \in r \quad .$$

Here is an example of a relation $r^{:A \leftrightarrow B}$ that is *not* equivalent to a function:

$$(x^{:A}, y^{:B}) \in r \text{ means } p(x) = q(y) \text{ where } p^{:A \to C} \text{ and } q^{:B \to C} \text{ are some given functions} \quad .$$

We call this a **pullback relation** and denote it by $\text{pull}\,(p, q)$. For some choices of p and q, there will be many different values $x_1, x_2, y_1, y_2, ...$, such that $p(x_1) = p(x_2) = q(y_1) = q(y_2) = ...$ Then the pullback relation will be many-to-many and will not be equivalent to $\langle f \rangle$ or to $\text{rev}\,\langle f \rangle$ for any function f.

In the derivations below, we will often use relations of type $P^{A,A} \leftrightarrow P^{B,B}$, where $P^{X,Y}$ is a profunctor (contravariant in X and covariant in Y). For any function $f^{:A \to B}$, there is a special pullback relation of type $P^{A,A} \leftrightarrow P^{B,B}$:

Definition D.3.1.2 Given a profunctor $P^{X,Y}$ and a function $f^{:A \to B}$, two values $x^{:P^{A,A}}$ and $y^{:P^{B,B}}$ are **in a (P, f)-wedge relation** if $x \triangleright f^{\uparrow P^{A,\bullet}} = y \triangleright f^{\downarrow P^{\bullet,B}}$. This condition can be written more concisely as $(x, y) \in \text{pull}\,(f^{\uparrow P}, f^{\downarrow P})$.

The wedge relation comes up often in applications of the relational parametricity theorem. The (P, f)-wedge relation generalizes the wedge law (D.3) to two arbitrary values x and y. The wedge law of an expression $p : \forall A.\, P^{A,A}$ is equivalent to the requirement that p^A and p^B are in the (P, f)-wedge relation for all $f^{:A \to B}$ and for all types A, B.

D.3.2 Relational product, co-product, and pair mapper. Relational lifting

Lifting of relations to type constructors is necessary in order to formulate naturality laws via relations. To see why, consider ordinary naturality laws involving functions, such as Eq. (D.1). These laws are formulated using an arbitrary function $f^{:A \to B}$ lifted to some functors or contrafunctors. Lifting a function $f^{:A \to B}$ to a functor G yields a function $f^{\uparrow G}$ of type $G^A \to G^B$. Lifting f to a contrafunctor H yields a function $f^{\downarrow H} : H^B \to H^A$. As a first step towards writing naturality laws in terms of relations, we need to define a lifting of an arbitrary relation $r^{:A \leftrightarrow B}$ to any type constructor $G^\bullet$. The lifted relation will have type $G^A \leftrightarrow G^B$ and will be denoted by $r^{\updownarrow G}$.

As motivation, let us reformulate some known naturality laws in terms of relations. Consider a natural transformation $t : \forall A.\, G^A \to H^A$ with its naturality law $f^{\uparrow G} \fatsemi t = t \fatsemi f^{\uparrow H}$. To obtain a relational formulation of that law, we first apply both sides to an arbitrary value $p^{:G^A}$:

$$p \triangleright f^{\uparrow G} \triangleright t^B = p \triangleright t^A \triangleright f^{\uparrow H} \quad , \qquad \text{or equivalently :} \quad t^B(p \triangleright f^{\uparrow G}) = t^A(p) \triangleright f^{\uparrow H} \quad ,$$

and define for brevity the value $q^{:G^B} \stackrel{\text{def}}{=} p \triangleright f^{\uparrow G}$. This definition of q is equivalent to the condition $(p, q) \in \langle f^{\uparrow G} \rangle$. Now we can rewrite the naturality law of t as a statement about relations:

$$\text{for any } f^{:A \to B}, p^{:G^A}, q^{:G^B} \quad : \quad \text{when} \quad (p, q) \in \langle f^{\uparrow G} \rangle \quad \text{then} \quad (t^A(p), t^B(q)) \in \langle f^{\uparrow H} \rangle \quad . \tag{D.9}$$

We will define a concise notation for conditions of this form because they are found often when working with relations.

Definition D.3.2.1 The **pair mapper of relations** $r^{:A \leftrightarrow C}$ and $s^{:B \leftrightarrow D}$ is a relation $r \boxtimes s$ defined by:

$$\big(r^{:A \leftrightarrow C} \boxtimes s^{:B \leftrightarrow D}\big) : (A \to B) \leftrightarrow (C \to D) \quad ,$$
$$(f^{:A \to B}, g^{:C \to D}) \in r \boxtimes s \quad \text{means} \quad \forall x^{:A}, y^{:C} \quad : \quad \text{if } (x, y) \in r \quad \text{then} \quad (f(x), g(y)) \in s \quad .$$

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, leftrightarrow, "r"'] \arrow[dr, phantom, "r \boxtimes s"] & B \arrow[d, leftrightarrow, "s"] \\ C \arrow[r, "g"'] & D \end{tikzcd}$$

The operation $\boxtimes$ associates to the right, so that $r \boxtimes s \boxtimes t \stackrel{\text{def}}{=} r \boxtimes (s \boxtimes t)$. The type diagram at left illustrates the pair mapper construction. To read the diagram, we need to start with two arbitrary values, $x^{:A}$ at the top left and $y^{:C}$ at the bottom left. Given two relations $r^{:A \leftrightarrow C}$ and $s^{:B \leftrightarrow D}$, we define a new relation $r \boxtimes s$ between functions of types $A \to B$ and $C \to D$. Two functions (f, g) are in the relation $r \boxtimes s$ if the diagram commutes, which means that $(f(x), g(y)) \in s$ for any $(x, y) \in r$. □

Using Definition D.3.2.1, we can rewrite the naturality law (D.9) as:

$$\text{for all } f^{:A \to B} \quad : \quad (t^A, t^B) \in \langle f^{\uparrow G} \rangle \boxtimes \langle f^{\uparrow H} \rangle \quad .$$

It appears reasonable to expect that the relation $\langle f \rangle$ lifted to G is the same relation as $\langle f^{\uparrow G} \rangle$:

$$\langle f \rangle^{\updownarrow G} = \langle f^{\uparrow G} \rangle \quad .$$

With this property (proved below in Statement D.3.6.12), we rewrite Eq. (D.9) as:

$$\text{for all } f^{:A \to B} : \quad (t^A, t^B) \in \langle f \rangle^{\updownarrow G} \boxtimes \langle f \rangle^{\updownarrow H} \quad .$$

We now replace $\langle f \rangle$ by an arbitrary relation $r^{:A \leftrightarrow B}$ and obtain the **relational naturality law**:

$$\text{for any relation } r^{:A \leftrightarrow B} : \quad (t^A, t^B) \in r^{\updownarrow G} \boxtimes r^{\updownarrow H} \quad . \tag{D.10}$$

Note the similarity of the relation $r^{\updownarrow G} \boxtimes r^{\updownarrow H}$ and the type $G^A \to H^A$ of t. Denote the type signature of t by $P^A \stackrel{\text{def}}{=} G^A \to H^A$. If we *define* the lifting $r^{\updownarrow P}$ as $r^{\updownarrow G} \boxtimes r^{\updownarrow H}$, the naturality law (D.10) will be written as:

$$\text{for any relation } r^{:A \leftrightarrow B} : \quad (t^A, t^B) \in r^{\updownarrow P} \quad . \tag{D.11}$$

The generalization of the naturality law from functions to relations now has a concise form: The components t^A and t^B of a natural transformation t belong to *any* relation r lifted to t's type signature (the type constructor $P^\bullet$). This prepares us for the formulation of the parametricity theorem below. At the same time, this motivates using the pair mapper operation ($\boxtimes$) to define the lifting a relation to a function type constructor such as $G^A \to H^A$.

What about lifting a relation to product ($G^A \times H^A$) or co-product ($G^A + H^A$) type constructors? Recall that lifting a function to $G^A \times H^A$ involves the pair product of functions ($\boxtimes$):

$$\left(f^{:A \to C} \boxtimes g^{:B \to D}\right) : A \times B \to C \times D \quad , \qquad f \boxtimes g \stackrel{\text{def}}{=} a \times b \to f(a) \times g(b) \quad .$$

Namely, the lifting of a function $k^{:A \to B}$ to $G^A \times H^A$ is defined by $k^{\uparrow (G \times H)} \stackrel{\text{def}}{=} k^{\uparrow G} \boxtimes k^{\uparrow H}$.

The pair co-product of functions is defined by Exercise 7.2.5.1(b):

$$\begin{aligned} &\left(f^{:A \to C} \boxplus g^{:B \to D}\right) : A + B \to C + D \quad , \\ &(f \boxplus g)(a^{:A} + \mathbb{0}^{:B}) \stackrel{\text{def}}{=} f(a) + \mathbb{0}^{:D} \quad , \qquad (f \boxplus g)(\mathbb{0} + b^{:B}) \stackrel{\text{def}}{=} \mathbb{0}^{:C} + g(b) \quad , \\ &(k^{:A \to B})^{\uparrow (G+H)} \stackrel{\text{def}}{=} k^{\uparrow G} \boxplus k^{\uparrow H} \quad . \end{aligned}$$

Let us now define the analogous constructions for relations.

Definition D.3.2.2 Given two relations $r^{:A \leftrightarrow C}$ and $s^{:B \leftrightarrow D}$, the **pair product** $r \boxtimes s$ is a relation between values of the product types $A \times B$ and $C \times D$:

$$\left(r^{:A \leftrightarrow C} \boxtimes s^{:B \leftrightarrow D}\right) : A \times B \leftrightarrow C \times D \quad , \qquad (a \times b, c \times d) \in r \boxtimes s \quad \text{when} \quad (a, c) \in r \text{ and } (b, d) \in s \quad .$$

The second construction is the pair co-product of relations, which creates a relation between co-products of types. The definition resembles that of the pair co-product of functions:

Definition D.3.2.3 The **pair co-product** of relations $r^{:A\leftrightarrow C}$ and $s^{:B\leftrightarrow D}$ is a relation $r \boxplus s$ defined by:

$$\begin{aligned}
&\left(r^{:A\leftrightarrow C} \boxplus s^{:B\leftrightarrow D}\right) : A+B \leftrightarrow C+D \quad , \\
&\text{either} \qquad (a^{:A}+\mathbb{0}^{:B}, c^{:C}+\mathbb{0}^{:D}) \in r \boxplus s \quad \text{when} \quad (a,c) \in r \quad , \\
&\text{or} \qquad (\mathbb{0}^{:A}+b^{:B}, \mathbb{0}^{:C}+d^{:D}) \in r \boxplus s \quad \text{when} \quad (b,d) \in s \quad .
\end{aligned}$$

Any other combinations of values (such as $a+\mathbb{0}$ and $\mathbb{0}+d$) are not in the relation $r \boxplus s$. □

The following example illustrates some properties of the relational operations $\boxtimes$, $\boxplus$, and $\boxslash$ when applied to relations obtained from functions.

Example D.3.2.4 For arbitrary given functions $f^{:A\to B}$ and $g^{:C\to D}$, prove that:

(a) $\langle f\rangle \boxtimes \langle g\rangle = \langle f \boxtimes g\rangle$, both relations having type $A\times C \leftrightarrow B\times D$.
(b) $\langle f\rangle \boxplus \langle g\rangle = \langle f \boxplus g\rangle$, both relations having type $A+C \leftrightarrow B+D$.
(c) $\langle f\rangle \boxslash \mathrm{id}^{:C\leftrightarrow C} = \mathrm{rev}\,\langle l^{:B\to C} \to f \fatsemi l\rangle$, both relations having type $(A\to C)\leftrightarrow(B\to C)$.
(d) $\mathrm{id}^{:A\leftrightarrow A} \boxslash \langle g\rangle = \langle k^{:A\to C} \to k \fatsemi g\rangle$, both relations having type $(A\to C)\leftrightarrow(A\to D)$.
(e) $\langle f\rangle \boxslash \langle g\rangle = \mathrm{pull}\,(k^{:A\to C} \to k\fatsemi g, l^{:B\to C} \to f\fatsemi l)$, both relations having type $(A\to C)\leftrightarrow(B\to D)$.
(f) $\left(\mathrm{rev}\,\langle f\rangle\right) \boxslash \langle g\rangle = \langle l^{:B\to C} \to f\fatsemi l\fatsemi g\rangle$, both relations having type $(B\to C)\leftrightarrow(A\to D)$.

Solution We will apply various relations to arbitrary values $a^{:A}$, $b^{:B}$, $c^{:C}$, $d^{:D}$.

(a) The following conditions are equivalent:

$$\begin{aligned}
&(a\times c, b\times d) \in \langle f\rangle \boxtimes \langle g\rangle \quad \text{means} \quad (a,b)\in\langle f\rangle \quad \text{and} \quad (c,d)\in\langle g\rangle \quad \text{or}: \quad f(a)=b \text{ and } g(c)=b \quad , \\
&(a\times c, b\times d) \in \langle f \boxtimes g\rangle \quad \text{means} \quad (a\times c)\triangleright(f\boxtimes g) = b\times d \quad \text{or}: \quad f(a)\times g(c) = b\times d \quad .
\end{aligned}$$

(b) The following conditions are equivalent:

$$\begin{aligned}
&(a+\mathbb{0}, b+\mathbb{0}) \in \langle f\rangle \boxplus \langle g\rangle \quad \text{means} \quad (a,b)\in\langle f\rangle \quad \text{or equivalently}: \quad a\triangleright f = b \quad , \\
&(a+\mathbb{0}, b+\mathbb{0}) \in \langle f \boxplus g\rangle \quad \text{means} \quad (a+\mathbb{0})\triangleright(f\boxplus g) = b+\mathbb{0} \quad \text{or equivalently}: \quad a\triangleright f = b \quad .
\end{aligned}$$

Similarly we can show that the conditions $(\mathbb{0}+c, \mathbb{0}+d) \in \langle f\rangle \boxplus \langle g\rangle$ and $(\mathbb{0}+c, \mathbb{0}+d) \in \langle f\boxplus g\rangle$ are equivalent. Values from different parts of the disjunctions, such as $(a+\mathbb{0}, \mathbb{0}+d)$, are not in the relation $\langle f\rangle \boxplus \langle g\rangle$ by definition of $\boxplus$. They are also not in the relation $\langle f\boxplus g\rangle$ since the function $f\boxplus g$ preserves the left and right parts of the disjunctions.

(c) Two functions $k^{:A\to C}$ and $l^{:B\to C}$ are in the relation $\langle f\rangle \boxslash \mathrm{id}^{:C\leftrightarrow C}$ if:

$$\text{for all } a^{:A}, b^{:B} \quad : \quad \text{when} \quad (a,b)\in\langle f\rangle \quad \text{then} \quad (k(a), l(b)) \in \mathrm{id}^{:C\leftrightarrow C} \quad .$$

Simplifying the last condition, we get:

$$\text{for all } a^{:A} \quad : \quad k(a) = l(f(a)) \quad , \quad \text{or equivalently}: \quad k = f\fatsemi l \quad .$$

So, the relation between k and l may be expressed as $k=\psi(l)$ where the function ψ is defined as:

$$\psi : (B\to C)\to A\to C \quad , \qquad \psi \stackrel{\text{def}}{=} l^{:B\to C} \to f\fatsemi l \quad .$$

This relation is denoted by $\mathrm{rev}\,\langle\psi\rangle$. It follows that $\langle f\rangle \boxslash \mathrm{id}^{:C\leftrightarrow C} = \mathrm{rev}\,\langle\psi\rangle = \mathrm{rev}\langle l^{:B\to C} \to f\fatsemi l\rangle$.

(d) Two functions $k^{:A\to C}$ and $l^{:A\to D}$ are in the relation $\mathrm{id}^{:A\leftrightarrow A} \boxslash \langle g\rangle$ if:

$$\text{for all } x^{:A}, y^{:A} \quad : \quad \text{when} \quad (x,y)\in \mathrm{id}^{:A\leftrightarrow A} \quad \text{then} \quad (k(x), l(y)) \in \langle g\rangle \quad .$$

Simplifying the last condition, we get:

$$\text{for all } x^{:A} \quad : \quad g(k(x)) = l(x) \quad , \quad \text{or equivalently}: \quad l = k\fatsemi g \quad .$$

So, the relation between k and l may be expressed as $\phi(k)=l$ where:

$$\phi : (A\to C)\to A\to D \quad , \quad \phi \stackrel{\text{def}}{=} k^{:A\to C} \to k\fatsemi g \quad .$$

This relation is denoted by $\langle\phi\rangle$. It follows that we may write $\mathrm{id}^{:A\leftrightarrow A} \boxtimes \langle g\rangle = \langle\phi\rangle = \langle k^{:A\to C} \to k \fatsemi g\rangle$.

(e) Two functions $k^{:A\to C}$ and $l^{:B\to D}$ are in the relation $\langle f\rangle \boxtimes \langle g\rangle$ if:

$$\text{for all } a^{:A}, b^{:B} \quad : \quad \text{when} \quad (a,b) \in \langle f\rangle \quad \text{then} \quad (k(a), l(b)) \in \langle g\rangle \quad .$$

Simplifying the last condition, we get:

$$\text{for all } a^{:A} \quad : \quad g(k(a)) = l(f(a)) \quad , \quad \text{or equivalently :} \quad f \fatsemi l = k \fatsemi g \quad .$$

So, the relation between k and l may be expressed as $\phi(k) = \psi(l)$ where:

$$\phi : (A \to C) \to A \to D \quad , \quad \phi \stackrel{\text{def}}{=} k^{:A\to C} \to k \fatsemi g \quad , \quad \psi : (B \to D) \to A \to D \quad , \quad \psi \stackrel{\text{def}}{=} l^{:B\to C} \to f \fatsemi l \quad .$$

This is a pullback relation that we denote by $\mathrm{pull}\,(\phi,\psi) = \mathrm{pull}\left(k^{:A\to C} \to k \fatsemi g, l^{:B\to C} \to f \fatsemi l\right)$.

(f) Two functions $k^{:B\to C}$ and $l^{:A\to D}$ are in the relation $\mathrm{rev}\,\langle f\rangle \boxtimes \langle g\rangle$ if:

$$\text{for all } x^{:B}, y^{:A} \quad : \quad \text{when} \quad (x,y) \in \mathrm{rev}\,\langle f\rangle \quad \text{then} \quad (k(x), l(y)) \in \langle g\rangle \quad .$$

Simplifying the last condition, we get:

$$\text{for all } y^{:A} \quad : \quad g(k(f(y))) = l(y) \quad , \quad \text{or equivalently :} \quad f \fatsemi k \fatsemi g = l \quad .$$

So, the relation between k and l may be expressed as $\phi(k) = l$ where:

$$\phi : (B \to C) \to A \to D \quad , \quad \phi \stackrel{\text{def}}{=} k^{:B\to C} \to f \fatsemi k \fatsemi g \quad .$$

This relation is denoted by $\langle\phi\rangle$. We get $\mathrm{rev}\,\langle f\rangle \boxtimes \langle g\rangle = \langle\phi\rangle = \langle k^{:B\to C} \to f \fatsemi k \fatsemi g\rangle$. □

We now turn to the question of how to define the lifting $r^{\updownarrow G}$ to an arbitrary type constructor $G^\bullet$. It turns out that we will actually need to define a *simultaneous* lifting of several relations to a type constructor with several type parameters. For clarity, we postpone that definition and begin by lifting a single relation.

Definition D.3.2.5 (relational lifting) Given a relation $r^{:A\leftrightarrow B}$ and a type constructor $G^\bullet$, the relational lifting of r to G, denoted by $r^{\updownarrow G}$, is a new relation of type $G^A \leftrightarrow G^B$. The relation $r^{\updownarrow G}$ is defined by induction on the structure of G.

Below, the type constructors K^A, L^A, $H^{X,A}$, and $S^{A,R}$ are not assumed to be covariant or contravariant. They can be any exponential-polynomial, fully parametric type constructors.

(a) If $G^A \stackrel{\text{def}}{=} Z$ with a fixed type Z, we define $r^{\updownarrow G} \stackrel{\text{def}}{=} \mathrm{id}^{:Z\leftrightarrow Z}$.

(b) If $G = \mathrm{Id}$ (that is, $G^A \stackrel{\text{def}}{=} A$), we define $r^{\updownarrow \mathrm{Id}} \stackrel{\text{def}}{=} r$.

(c) If $G^A \stackrel{\text{def}}{=} K^A \times L^A$ then we define $r^{\updownarrow G} \stackrel{\text{def}}{=} r^{\updownarrow K} \boxtimes r^{\updownarrow L}$. The inductive assumption is that the relational liftings $r^{\updownarrow K}$ and $r^{\updownarrow L}$ are already defined.

(d) If $G^A \stackrel{\text{def}}{=} K^A + L^A$ then we define $r^{\updownarrow G} \stackrel{\text{def}}{=} r^{\updownarrow K} \boxplus r^{\updownarrow L}$ with the same inductive assumptions.

(e) If $G^A \stackrel{\text{def}}{=} K^A \to L^A$ then we define $r^{\updownarrow G} \stackrel{\text{def}}{=} r^{\updownarrow K} \boxtimes r^{\updownarrow L}$ with the same inductive assumptions.

(f) If $G^A \stackrel{\text{def}}{=} S^{A,G^A}$ is defined recursively via a recursion scheme $S^{\bullet,\bullet}$, we define $r^{\updownarrow G}$ by:

$$r^{\updownarrow G} \stackrel{\text{def}}{=} \left(r, \overline{r^{\uparrow G}}\right)^{\updownarrow S} \quad .$$

Here the notation $(r,s)^{\updownarrow S}$ means the simultaneous lifting of the two relations r, s to the type constructor $S^{\bullet,\bullet}$ (see Definition D.3.3.2 below). Also note that we use $\overline{r^{\updownarrow G}}$ recursively within the definition of $r^{\updownarrow G}$. This is allowed since we understand $r^{\updownarrow G}$ to be a function (of type $G^A \times G^B \to \mathbb{2}$, see Definition D.3.1.1), and it is permitted to define functions recursively. The inductive assumption is that simultaneous liftings to $S^{\bullet,\bullet}$ are already defined.

(g) If $G^A \stackrel{\text{def}}{=} \forall X.\, H^{X,A}$, we define $r^{\updownarrow G}$ of type $(\forall X.\, H^{X,A}) \leftrightarrow (\forall Y.\, H^{Y,B})$ by:

$$(p^{:\forall X.\, H^{X,A}}, q^{:\forall X.\, H^{X,B}}) \in r^{\updownarrow \forall X.\, H^{X,\bullet}} \quad \text{means} \quad \forall (X,Y).\, \forall s^{X\leftrightarrow Y}.\, (p^X, q^Y) \in (s,r)^{\updownarrow H} \quad .$$

Here $(s,r)^{\updownarrow H}$ denotes the simultaneous lifting of s and r to H (see Definition D.3.3.2 below). A shorter way of writing the formula above is by formulating a relation between p^X and q^Y directly:

$$\forall (X,Y): \quad \left(r^{\updownarrow \forall Z.\, H^{Z,\bullet}}\right)^{X,Y} \stackrel{\text{def}}{=} \forall s^{:X\leftrightarrow Y}.\, (s,r)^{\updownarrow H} \quad . \tag{D.12}$$

The inductive assumption is that simultaneous liftings to H are already defined. □

We can see that Definition D.3.2.5 translates the type structure of G^A into an analogous relational structure. Constant types (Z) are replaced by the corresponding identity relations $(\mathrm{id}^{:Z\leftrightarrow Z})$. Each occurrence of A in G^A is replaced by an occurrence of the relation r being lifted. Product types, coproduct types, and function types are replaced by the relational operations $\boxtimes$, $\boxplus$, and $\boxrightarrow$. Recursive usage of types are replaced by the recursive usage of the lifted relations. Universally quantified types $(\forall X)$ are replaced by universally quantified new relations between new types.

So, an expression for a relational lifting $r^{\updownarrow G}$ can be written immediately by looking at the structure of G. For instance, lifting a relation $r^{:A\leftrightarrow B}$ to a type constructor $G^A \stackrel{\text{def}}{=} \forall X.\ (A\to X)\times(X\to A+Z)$ is expressed as:

$$\left(r^{\updownarrow G}\right)^{X,Y} = \forall s^{:X\leftrightarrow Y}.\, (r \boxrightarrow s) \boxtimes \left(s \boxrightarrow (r \boxplus \mathrm{id}^{:Z\leftrightarrow Z})\right) \quad .$$

We will illustrate this technique further in Example D.3.2.6 below.

To get more intuition, let us consider relations r obtained from functions f as $r = \langle f\rangle$. In that case, one can sometimes (but not always) express the lifted relation $\langle f\rangle^{\updownarrow P}$ through ordinary function liftings $f^{\uparrow P}$ and $f^{\downarrow P}$ via the (P,f)-wedge condition.

Example D.3.2.6 A function $f^{:A\to B}$ gives rise to a relation $\langle f\rangle$ of type $A\leftrightarrow B$. Use Definition D.3.2.5 to compute the lifting $\langle f\rangle^{\updownarrow P}$ for the following type constructors $P^\bullet$:

(a) $P^A \stackrel{\text{def}}{=} A + A\times A$.

(b) $P^A \stackrel{\text{def}}{=} (R\to A)\times(A\to S)$ where R and S are fixed types.

(c) $P^A \stackrel{\text{def}}{=} A\to A$.

(d) $P^A \stackrel{\text{def}}{=} A\to(A\to R)\to R$ where R is a fixed type.

(e) $P^A \stackrel{\text{def}}{=} (A\to A)\to A$.

In each case **(a)–(e)**, express $P^A \stackrel{\text{def}}{=} N^{A,A}$ where $N^{X,Y}$ is a suitable profunctor and compare the condition $(p_1,p_2)\in\langle f\rangle^{\updownarrow P}$ with the (N,f)-wedge relation $(p_1 \triangleright f^{\uparrow N} = p_2 \triangleright f^{\downarrow N})$.

Solution In all cases, the lifted relation $\langle f\rangle^{\updownarrow P}$ has type $P^A\leftrightarrow P^B$.

(a) At the top level, P^A is a disjunction: $P^A = \mathrm{Id}^A + H^A$ where $H^A \stackrel{\text{def}}{=} A\times A = \mathrm{Id}^A\times\mathrm{Id}^A$. The lifting to the identity functor is given by $r^{\updownarrow\mathrm{Id}} \stackrel{\text{def}}{=} r$ according to Definition D.3.2.5(b). So, the lifting $r^{\updownarrow P}$ can be written as:

$$r^{\updownarrow P} = r^{\updownarrow\mathrm{Id}} \boxplus r^{\updownarrow H} = r^{\updownarrow\mathrm{Id}} \boxplus \left(r^{\updownarrow\mathrm{Id}} \boxtimes r^{\updownarrow\mathrm{Id}}\right) = r \boxplus \left(r\boxtimes r\right) \quad .$$

We see that the relational formula $r^{\updownarrow P} = r\boxplus\left(r\boxtimes r\right)$ repeats the type structure $P^A = A + A\times A$.

Setting now $r = \langle f\rangle$, we find:

$$\langle f\rangle^{\updownarrow P} = \langle f\rangle \boxplus \langle f\rangle^{\updownarrow H} = \langle f\rangle \boxplus \left(\langle f\rangle \boxtimes \langle f\rangle\right) \quad .$$

Use Definition D.3.2.5(d) to express the condition for some values $(x^{:P^A}, y^{:P^B})$ to be in the relation $\langle f\rangle \boxplus \langle f\rangle^{\updownarrow H}$:

$$(x,y)\in\langle f\rangle^{\updownarrow P} \text{ if } x = x_1 + \mathbb{0},\ y = y_1 + \mathbb{0},\ (x_1,y_1)\in\langle f\rangle \text{ or } x = \mathbb{0} + x_2,\ y = \mathbb{0} + y_2,\ (x_2,y_2)\in\langle f\rangle^{\updownarrow H} \quad .$$

The condition $(x_1,y_1)\in\langle f\rangle$ means $f(x_1) = y_1$. Finally, Definition D.3.2.5(c) gives:

$$(a_1^{:A}\times a_2^{:A}, b_1^{:B}\times b_2^{:B})\in\langle f\rangle\boxtimes\langle f\rangle \text{ if } (a_1,b_1)\in\langle f\rangle \text{ and } (a_2,b_2)\in\langle f\rangle \quad .$$

The last condition is simplified to:

$$(a_1^{:A}\times a_2^{:A}, b_1^{:B}\times b_2^{:B})\in\langle f\rangle^{\updownarrow H} \text{ if } f(a_1) = b_1 \text{ and } f(a_2) = b_2 \quad .$$

Putting the pieces together, we obtain the following definition of the relation $\langle f\rangle^{\updownarrow P}$:

$$(x^{:P^A}, y^{:P^B}) \in \langle f\rangle^{\updownarrow P} \text{ if } x = x_1 + \mathbb{0} \text{ and } y = f(x_1) + \mathbb{0}, \text{ or } x = \mathbb{0} + a_1 \times a_2 \text{ and } y = \mathbb{0} + f(a_1) \times f(a_2) \quad .$$

We note that this condition is equivalent to applying $f^{\uparrow P}$:

$$(x^{:P^A}, y^{:P^B}) \in \langle f\rangle^{\updownarrow P} \text{ if } x \triangleright f^{\uparrow P} = y \quad , \qquad \text{or more concisely :} \quad \langle f\rangle^{\updownarrow P} = \langle f^{\uparrow P}\rangle \quad .$$

We will prove below (Statement D.3.6.12) that this property holds for all functors P.

With $N^{X,Y} \stackrel{\text{def}}{=} P^Y$, the (N, f)-wedge relation is equivalent to the equation $x \triangleright f^{\uparrow P} = y$.

(b) At the top level, P^A is a product: $P^A = G^A \times H^A$ where $G^A \stackrel{\text{def}}{=} R \to A$ and $H^A \stackrel{\text{def}}{=} A \to S$. By Definition D.3.2.5(c), we get $r^{\updownarrow P} = r^{\updownarrow G} \boxtimes r^{\updownarrow H}$. To compute $r^{\updownarrow G}$, we note that G is an exponential functor construction, $G^A = K^A \to \mathrm{Id}^A$, that uses a constant functor $K^A \stackrel{\text{def}}{=} R$. Lifting to a constant functor gives $r^{\updownarrow K} = \mathrm{id}^{:R\leftrightarrow R}$ according to Definition D.3.2.5(a). So, we find:

$$r^{\updownarrow G} = \mathrm{id}^{:R\leftrightarrow R} \boxtimes r^{\updownarrow \mathrm{Id}} = \mathrm{id}^{:R\leftrightarrow R} \boxtimes r \quad .$$

We treat $r^{\updownarrow H}$ similarly and obtain $r^{\updownarrow H} = r \boxtimes \mathrm{id}^{:S\leftrightarrow S}$. Put the pieces together:

$$r^{\updownarrow P} = r^{\updownarrow G} \boxtimes r^{\updownarrow H} = (\mathrm{id}^{:R\leftrightarrow R} \boxtimes r) \boxtimes (r \boxtimes \mathrm{id}^{:S\leftrightarrow S}) \quad .$$

We see that $r^{\updownarrow P}$ repeats the type structure of $P^A = (R \to A) \times (A \to S)$. The occurrences of A in P^A are replaced by r, while constant types (R, S) are replaced by the corresponding identity relations ($\mathrm{id}^{:R\leftrightarrow R}$ and $\mathrm{id}^{:S\leftrightarrow S}$). We will see a similar pattern in the remaining parts of this example.

We now set $r = \langle f\rangle$ and transform $\langle f\rangle^{\updownarrow P}$ to a more specific formula for a relation of type $P^A \leftrightarrow P^B$ where A, B are arbitrary types. For arbitrary values $g_1^{:R\to A}$, $g_2^{:R\to B}$, $h_1^{:A\to S}$, $h_2^{:B\to S}$, we write:

$$(g_1 \times h_1, g_2 \times h_2) \in \langle f\rangle^{\updownarrow P} \text{ means } (g_1, g_2) \in \mathrm{id}^{:R\leftrightarrow R} \boxtimes \langle f\rangle \text{ and } (h_1, h_2) \in \langle f\rangle \boxtimes \mathrm{id}^{:S\leftrightarrow S} \quad .$$

The condition $(g_1, g_2) \in \mathrm{id} \boxtimes \langle f\rangle$ is rewritten using Definition D.3.2.5(e):

$$(g_1^{:R\to A}, g_2^{:R\to B}) \in \mathrm{id} \boxtimes \langle f\rangle \text{ means if } (r_1^{:R}, r_2^{:R}) \in \mathrm{id}^{:R\leftrightarrow R} \text{ then } (g_1(r_1), g_2(r_2)) \in \langle f\rangle \quad .$$

The identity relation between r_1 and r_2 holds only if $r_1 = r_2$. We find:

$$(g_1^{:R\to A}, g_2^{:R\to B}) \in \mathrm{id}^{:R\leftrightarrow R} \boxtimes \langle f\rangle \text{ means } \forall r^{:R}.\, f(g_1(r)) = g_2(r) \quad \text{or equivalently :} \quad g_1 \fatsemi f = g_2 \quad .$$

The condition $(h_1, h_2) \in \langle f\rangle^{\updownarrow H}$ is rewritten using the same definitions:

$$(h_1^{:A\to S}, h_2^{:B\to S}) \in \langle f\rangle \boxtimes \mathrm{id}^{:S\leftrightarrow S} \text{ means if } (a^{:A}, b^{:B}) \in \langle f\rangle \text{ then } (h_1(a), h_2(b)) \in \mathrm{id} \quad .$$

We simplify this to:

$$(h_1^{:A\to S}, h_2^{:B\to S}) \in \langle f\rangle \boxtimes \mathrm{id}^{:S\leftrightarrow S} \text{ means } h_1(a) = h_2(f(a)) \quad \text{or equivalently :} \quad h_1 = f \fatsemi h_2 \quad .$$

Finally, the lifted relation $\langle f\rangle^{\updownarrow P}$ is expressed as:

$$(g_1 \times h_1, g_2 \times h_2) \in \langle f\rangle^{\updownarrow P} \text{ if } g_1 \fatsemi f = g_2 \text{ and } h_1 = f \fatsemi h_2 \quad .$$

Defining the profunctor $N^{X,Y} \stackrel{\text{def}}{=} (R \to Y) \times (X \to S)$, we find that $g_1 \times h_1$ has type $N^{A,A}$ and $g_2 \times h_2$ has type $N^{B,B}$. Then the condition $(g_1 \times h_1, g_2 \times h_2) \in \langle f\rangle^{\updownarrow P}$ is *equivalent* to the (N, f)-wedge relation:

$$(g_1 \times h_1) \triangleright f^{\uparrow N^{A,\bullet}} = (g_2 \times h_2) \triangleright f^{\downarrow N^{\bullet,B}} \quad .$$

(c) Using Definition D.3.2.5(b) and (e), we get:

$$r^{\updownarrow P} = r^{\updownarrow \mathrm{Id}} \boxtimes r^{\updownarrow \mathrm{Id}} = r \boxtimes r \quad , \qquad \text{so} \quad \langle f \rangle^{\updownarrow P} = \langle f \rangle \boxtimes \langle f \rangle \quad ;$$
$$(p_1^{:A\to A}, p_2^{:B\to B}) \in \langle f \rangle^{\updownarrow P} \quad \text{means} \quad \text{if } (a^{:A}, b^{:B}) \in \langle f \rangle \text{ then } (p_1(a), p_2(b)) \in \langle f \rangle \quad .$$

The condition can be rewritten as a single equation:

$$(p_1^{:A\to A}, p_2^{:B\to B}) \in \langle f \rangle^{\updownarrow P} \quad \text{means} \quad f(p_1(a)) = p_2(f(a)) \quad \text{or equivalently :} \quad p_1 \fatsemi f = f \fatsemi p_2 \quad .$$

This is the same as the (N, f)-wedge relation ($p_1 \triangleright f^{\uparrow N} = p_2 \triangleright f^{\downarrow N}$) with the profunctor $N^{X,Y} \stackrel{\text{def}}{=} X \to Y$.

(d) Since $P^A = \mathrm{Id}^A \to Q^A$ is a function type (with $Q^A \stackrel{\text{def}}{=} (A \to R) \to R$), we use Definition D.3.2.5(e):

$$r^{\updownarrow P} = r^{\updownarrow \mathrm{Id}} \boxtimes r^{\updownarrow Q} = r \boxtimes (r \boxtimes \mathrm{id}^{:R\leftrightarrow R}) \boxtimes \mathrm{id}^{:R\leftrightarrow R} \quad .$$

Let us now substitute $r = \langle f \rangle$ and simplify the conditions:

$$(p_1^{:P^A}, p_2^{:P^B}) \in \langle f \rangle \boxtimes \langle f \rangle^{\updownarrow Q} \quad \text{means} \quad \text{if } (a^{:A}, b^{:B}) \in \langle f \rangle \text{ then } (p_1(a), p_2(b)) \in \langle f \rangle^{\updownarrow Q} \quad .$$

The definition of $\langle f \rangle^{\updownarrow Q}$ is obtained if we define $K^A \stackrel{\text{def}}{=} A \to R$:

$$(q_1^{:Q^A}, q_2^{:Q^B}) \in \langle f \rangle^{\updownarrow Q} \quad \text{means} \quad \text{if } (k_1^{:A\to R}, k_2^{:B\to R}) \in \langle f \rangle^{\updownarrow K} \text{ then } (q_1(k_1), q_2(k_2)) \in \mathrm{id} \quad .$$

Expanding the definition of $\langle f \rangle^{\updownarrow K}$ as we did in part **(b)** of this example, we get:

$$(q_1, q_2) \in \langle f \rangle^{\updownarrow Q} \quad \text{means} \quad \text{if } b = f(a) \text{ and } k_1(a) = k_2(b) \text{ then } q_1(k_1) = q_2(k_2) \quad .$$

This simplifies to $k_1(a) = k_2(f(a))$, or $k_1 = f \fatsemi k_2$. So, we finally express $\langle f \rangle^{\updownarrow Q}$ and $\langle f \rangle^{\updownarrow P}$ as:

$$(q_1^{:Q^A}, q_2^{:Q^B}) \in \langle f \rangle^{\updownarrow Q} \quad \text{means} \quad (k \to f \fatsemi k) \fatsemi q_1 = q_2 \quad ,$$
$$(p_1^{:P^A}, p_2^{:P^B}) \in \langle f \rangle^{\updownarrow P} \quad \text{means} \quad a \to \big((k \to f \fatsemi k) \fatsemi p_1(a)\big) = f \fatsemi p_2 \quad .$$

Defining $N^{X,Y} \stackrel{\text{def}}{=} X \to (Y \to R) \to R$, we find that the last formula is equivalent to the (N, f)-wedge relation for p_1 and p_2:

$$p_1 \triangleright f^{\uparrow N} = p_1 \fatsemi (q \to (k \to f \fatsemi k) \fatsemi q) \quad , \qquad p_2 \triangleright f^{\downarrow N} = f \fatsemi p_2 \quad .$$

(e) We may write $P^A = Q^A \to \mathrm{Id}^A$ with $Q^A \stackrel{\text{def}}{=} A \to A$ and use Definition D.3.2.5(e):

$$r^{\updownarrow P} = r^{\updownarrow Q} \boxtimes r^{\updownarrow \mathrm{Id}} = (r \boxtimes r) \boxtimes r \quad , \qquad \langle f \rangle^{\updownarrow P} = (\langle f \rangle \boxtimes \langle f \rangle) \boxtimes \langle f \rangle \quad ,$$
$$(p_1^{:P^A}, p_2^{:P^B}) \in (\langle f \rangle \boxtimes \langle f \rangle) \boxtimes \langle f \rangle \quad \text{means} \quad \text{if } (q_1^{:A\to A}, q_2^{:B\to B}) \in \langle f \rangle \boxtimes \langle f \rangle \text{ then } (p_1(q_1), p_2(q_2)) \in \langle f \rangle \quad .$$
$$(q_1^{:A\to A}, q_2^{:B\to B}) \in \langle f \rangle \boxtimes \langle f \rangle \quad \text{means} \quad \text{if } (a^{:A}, b^{:B}) \in \langle f \rangle \text{ then } (q_1(a), q_2(b)) \in \langle f \rangle \quad .$$

Simplifying these conditions, we get:

$$(q_1, q_2) \in \langle f \rangle^{\updownarrow Q} \quad \text{means} \quad q_1 \fatsemi f = f \fatsemi q_2 \quad ,$$
$$(p_1^{:P^A}, p_2^{:P^B}) \in \langle f \rangle^{\updownarrow P} \quad \text{means} \quad \text{if } q_1 \fatsemi f = f \fatsemi q_2 \text{ then } f(p_1(q_1)) = p_2(q_2) \quad . \tag{D.13}$$

The condition for $(p_1, p_2) \in \langle f \rangle^{\updownarrow P}$ *cannot* be written as a single equation involving p_1 and p_2 because we cannot express q_1 through q_2 or q_2 through q_1. The relation between q_1 and q_2 is many-to-many and is not equivalent to a function.

Let us define $N^{X,Y} \stackrel{\text{def}}{=} (Y \to X) \to Y$ and write the (N, f)-wedge relation for $p_1^{:N^{A,A}}$ and $p_2^{:N^{B,B}}$:

$$k^{:B\to A} \triangleright (p_1 \triangleright f^{\uparrow N}) = k \triangleright (p_2 \triangleright f^{\downarrow N}) \quad \text{or equivalently :} \quad f(p_1(f \fatsemi k)) = p_2(k \fatsemi f) \quad . \tag{D.14}$$

We notice that if we set $q_1 \stackrel{\text{def}}{=} f \fatsemi k$ and $q_2 \stackrel{\text{def}}{=} k \fatsemi f$ then the precondition $q_1 \fatsemi f = f \fatsemi q_2$ in Eq. (D.13) will be satisfied, and it will follow that $f(p_1(q_1)) = p_2(q_2)$. So, the (N, f)-wedge relation is a *consequence* of the relation $(p_1, p_2) \in \langle f \rangle^{\updownarrow P}$. However, if we are given some functions f, q_1, and q_2 such that the precondition $q_1 \fatsemi f = f \fatsemi q_2$ holds, it does not follow that there exists a function k such that $q_1 = f \fatsemi k$ and $q_2 = k \fatsemi f$. A simple counterexample is found when f is a constant function that ignores its argument and always returns a fixed value:

$$f^{:A\to B} \stackrel{\text{def}}{=} _^{:A} \to b_0 \quad , \qquad \text{where } b_0 \text{ is a fixed value of type } B \quad .$$

Then the precondition $q_1 \fatsemi f = f \fatsemi q_2$ is reduced to the condition $q_2(b_0) = b_0$ with *no* restrictions on q_1. But if $q_1^{:A\to A}$ were expressed as $q_1 = f \fatsemi k$ with some $k^{:B\to A}$, we would have $q_1 = _ \to k(b_0)$. So, functions q_1 expressed as $f \fatsemi k$ are constant functions that ignore their argument. This is a stronger restriction than $q_1 \fatsemi f = f \fatsemi q_2$ because more pairs (q_1, q_2) satisfy $q_1 \fatsemi f = f \fatsemi q_2$. Accordingly, the (N, f)-wedge relation (D.14) constrains the functions (p_1, p_2) weaker and is satisfied by more pairs (p_1, p_2) than the relation $\langle f \rangle^{\updownarrow P}$. So, the (N, f)-wedge relation is *weaker* than the relation $\langle f \rangle^{\updownarrow P}$. □

Below we will prove (Statement D.3.5.1) that the (N, f)-wedge relation is always a consequence of the corresponding lifted relation $\langle f \rangle^{\updownarrow P}$. Example D.3.2.6 shows that the (N, f)-wedge relation is equivalent to $\langle f \rangle^{\updownarrow P}$ only when the type structure of P is sufficiently simple.

D.3.3 Properties of relational lifting. Simultaneous lifting

Definition D.3.2.5 requires some additional work to be fully usable. This section proves the properties of relational lifting that will be needed later in this Appendix.

We begin by pointing out an ambiguity in applying Definition D.3.2.5(g) to $G^A \stackrel{\text{def}}{=} \forall X. H^{X,A}$ when $H^{X,A}$ does *not* actually depend on the type parameter A; that is, when $H^{X,A} = K^X$ with some $K^\bullet$. In that case, the type constructor G is a constant functor: $G^A = Z \stackrel{\text{def}}{=} \forall X. K^X$. We may lift a relation $r^{:A\leftrightarrow B}$ to $G^A \stackrel{\text{def}}{=} Z$ using Definition D.3.2.5(a) and obtain $r^{\updownarrow G} = \text{id}^{:Z\leftrightarrow Z}$. To show that Definition D.3.2.5 is consistent, we need to prove that Definitions D.3.2.5(a) and (g) define the same lifted relation $r^{\updownarrow G}$. This will be shown in Statement D.3.3.1 below by using the relational naturality law (D.18). That law is the result of the relational parametricity theorem (Section D.3.4). So, the relational parametricity theorem turns out to be a necessary requirement for the consistency of Definition D.3.2.5.

Statement D.3.3.1 For a quantified type $Z \stackrel{\text{def}}{=} \forall X. H^X$, the relation $\text{id}^{:Z\leftrightarrow Z}$ has two equivalent forms:

$$(p^{:\forall X. H^X}, q^{:\forall Y. H^Y}) \in \text{id} \quad \text{is equivalent to :} \quad \forall (X, Y). \forall s^{:X\leftrightarrow Y}. (p^X, q^Y) \in s^{\updownarrow H} \quad .$$

Here we assume that p and q are implemented via fully parametric code.

Proof We need to prove the equivalence in both directions.

If some p and q satisfy $(p^X, q^Y) \in s^{\updownarrow H}$ where we may choose s arbitrarily, let us choose s to be the identity relation and force X and Y to be the same type:

$$(x^{:X}, y^{:Y}) \in s \text{ only when } X = Y \text{ and } x = y \quad .$$

Lifting an identity relation will produce an identity relation (Statement D.3.5.2), so:

$$\forall (X, Y). (p^X, q^Y) \in s^{\updownarrow H} \text{ with } s^{\updownarrow H} = \text{id} \quad \text{means} \quad X = Y \text{ and } p^X = q^X \quad .$$

This is the same as the identity relation between p and q.

It remains to show that $(p^X, q^Y) \in s^{\updownarrow H}$ when $p = q$. In other words, $(p^X, p^Y) \in s^{\updownarrow H}$ for any $s^{:X\leftrightarrow Y}$. Since p is fully parametric, it obeys the naturality law (D.18), which we will prove below without using this statement. We set $Q^X \stackrel{\text{def}}{=} H^X$, $r \stackrel{\text{def}}{=} s$, and $t \stackrel{\text{def}}{=} p$ in the naturality law (D.18) and obtain:

$$\forall (X, Y). \forall s^{:X\leftrightarrow Y}. (p^X, p^Y) \in s^{\updownarrow H} \quad .$$

□

Another construction required for Definition D.3.2.5 is a simultaneous lifting of two relations to a type constructor with two type parameters. To gain intuition, consider a bifunctor $S^{\bullet,\bullet}$ and two functions $f^{:A\to B}$ and $g^{:X\to Y}$. We can lift f and g simultaneously with respect to the two parameters of $S^{\bullet,\bullet}$ using the `bimap` method of the bifunctor S. The result is a function of type $S^{A,X} \to S^{B,Y}$:

$$\text{bimap}_S(f)(g) : S^{A,X} \to S^{B,Y} \quad , \quad \text{bimap}_S(f)(g) = f^{\uparrow S^{\bullet,X}} \fatsemi g^{\uparrow S^{B,\bullet}} = g^{\uparrow S^{A,\bullet}} \fatsemi f^{\uparrow S^{\bullet,Y}} \quad .$$

The resulting function is independent of the order of liftings due to the bifunctor commutativity law (Section D.2). However, it turns out that the analog of `bimap` for relations is an operation that *cannot* be expressed via separate liftings with respect to each type parameter. For this reason, we need to define the simultaneous relational lifting as a special new operation.

Definition D.3.3.2 (simultaneous relational lifting) Given any fully parametric type constructor with two type parameters ($G^{\bullet,\bullet}$), we lift two relations $r^{:A\leftrightarrow B}$ and $s^{:X\leftrightarrow Y}$ simultaneously to G. This creates a new relation of type $G^{A,X} \leftrightarrow G^{B,Y}$, denoted by $(r,s)^{\updownarrow G}$ and defined by induction on the structure of G:

(a) If $G^{A,X} \stackrel{\text{def}}{=} Z$ with a fixed type Z, we define $(r,s)^{\updownarrow G} \stackrel{\text{def}}{=} \text{id}^{:Z\leftrightarrow Z}$.

(b) If $G^{A,X} = A$, we define $(r,s)^{\updownarrow G} \stackrel{\text{def}}{=} r$. If $G^{A,X} = X$, we define $(r,s)^{\updownarrow G} \stackrel{\text{def}}{=} s$.

(c) If $K^{\bullet,\bullet}$ and $L^{\bullet,\bullet}$ are any fully parametric type constructors, we define:

$$\begin{aligned}
\text{for} \quad G^{A,X} \stackrel{\text{def}}{=} K^{A,X} \times L^{A,X} \quad &: \quad (r,s)^{\updownarrow G} \stackrel{\text{def}}{=} (r,s)^{\updownarrow K} \boxtimes (r,s)^{\updownarrow L} \quad ; \\
\text{for} \quad G^{A,X} \stackrel{\text{def}}{=} K^{A,X} + L^{A,X} \quad &: \quad (r,s)^{\updownarrow G} \stackrel{\text{def}}{=} (r,s)^{\updownarrow K} \boxplus (r,s)^{\updownarrow L} \quad ; \\
\text{for} \quad G^{A,X} \stackrel{\text{def}}{=} K^{A,X} \to L^{A,X} \quad &: \quad (r,s)^{\updownarrow G} \stackrel{\text{def}}{=} (r,s)^{\updownarrow K} \boxrightarrow (r,s)^{\updownarrow L} \quad .
\end{aligned}$$

The last relation is between functions of types $K^{A,X} \to L^{A,X}$ and $K^{B,Y} \to L^{B,Y}$.

The inductive assumption is that simultaneous liftings to K and L are already defined.

(d) If $G^{A,X} \stackrel{\text{def}}{=} S^{A,X,G^{A,X}}$ is defined recursively via a recursion scheme $S^{\bullet,\bullet,\bullet}$, we define:

$$(r,s)^{\updownarrow G} \stackrel{\text{def}}{=} \left(r, s, \overline{(r,s)^{\updownarrow G}}\right)^{\updownarrow S} \quad .$$

Here we use $\overline{(r,s)^{\updownarrow G}}$ recursively within the definition of $(r,s)^{\updownarrow G}$. This is allowed since we understand $(r,s)^{\updownarrow G}$ to be a function, and it is permitted to define functions recursively. The inductive assumption is that simultaneous liftings of any *three* relations to $S^{\bullet,\bullet,\bullet}$ are already defined.

(e) If $G^{A,X} \stackrel{\text{def}}{=} \forall Z.\, H^{Z,A,X}$, we define $(r,s)^{\updownarrow G}$ of type $(\forall U.\, H^{U,A,X}) \leftrightarrow (\forall V.\, H^{V,B,Y})$ by:

$$(p^{:\forall U.\, H^{U,A,X}}, q^{:\forall V.\, H^{V,B,Y}}) \in (r,s)^{\updownarrow \forall Z.\, H^{Z,\bullet,\bullet}} \quad \text{means} \quad \forall (U,V).\, \forall w^{U\leftrightarrow V}.\, (p^U, q^V) \in (w,r,s)^{\updownarrow H^{\bullet,\bullet,\bullet}} \quad .$$

A shorter way of writing this definition is by formulating a relation between p^U and q^V directly:

$$\forall (U,V): \quad \left((r,s)^{\updownarrow \forall Z.\, H^{Z,\bullet,\bullet}}\right)^{U,V} \stackrel{\text{def}}{=} \forall w^{:U\leftrightarrow V}.\, (w,r,s)^{\updownarrow H^{\bullet,\bullet,\bullet}} \quad . \tag{D.15}$$

The inductive assumption is that simultaneous liftings to $H^{\bullet,\bullet,\bullet}$ are already defined. □

Parts (d) and (e) of Definition D.3.3.2 use a simultaneous lifting of *three* relations. Comparing Definitions D.3.2.5 and D.3.3.2, we see that a similar inductive definition can be given for simultaneous liftings of n relations (with $n = 1, 2, ...$) to a type constructor with n type parameters. We omit the details. This book will only need simultaneous liftings of two relations.

When working with dinaturality laws (Section D.1.1), we used profunctors $P^{X,Y}$ whose type parameters are set to the same type (e.g., $G^A \stackrel{\text{def}}{=} P^{A,A}$). In that case, there is an ambiguity in lifting a relation $r^{:A\leftrightarrow B}$ to G: First, Definition D.3.2.5 defines $r^{\updownarrow G}$ as a relation of type $G^A \leftrightarrow G^B$, which is the same type as $P^{A,A} \leftrightarrow P^{B,B}$. Second, we may lift the pair of two relations (r,r) simultaneously to $P^{\bullet,\bullet}$ according to Definition D.3.3.2 and obtain another relation $(r,r)^{\updownarrow P}$ of the same type $P^{A,A} \leftrightarrow P^{B,B}$. The following statement shows that the relations $r^{\updownarrow G}$ and $(r,r)^{\updownarrow P}$ are equivalent.

Statement D.3.3.3 Given any fully parametric type constructor $P^{X,Y}$, define $G^A \stackrel{\text{def}}{=} P^{A,A}$. For any relation $r^{:A\leftrightarrow B}$, the liftings $r^{\updownarrow G}$ and $(r,r)^{\updownarrow P}$ will produce the same relation of type $P^{A,A} \leftrightarrow P^{B,B}$.

Proof We need to enumerate all cases of Definition D.3.3.2 for the type constructor $P^{\bullet,\bullet}$ and the corresponding cases of Definition D.3.2.5 for $G^\bullet$. In each case, we will show that $(r,r)^{\updownarrow H} = r^{\updownarrow G}$.

If $P^{A,X} \stackrel{\text{def}}{=} Z$ with a fixed type Z, we have also $G^A = Z$. Then $(r,r)^{\updownarrow P} = \text{id}^{:Z\leftrightarrow Z}$ and $r^{\updownarrow G} = \text{id}^{:Z\leftrightarrow Z}$.

If $P^{A,X} \stackrel{\text{def}}{=} A$ or $P^{A,X} = X$, we have $G^A = A$. In both cases $(r,r)^{\updownarrow H} \stackrel{\text{def}}{=} r$ and $r^{\updownarrow G} = r$.

If $P^{\bullet,\bullet} \stackrel{\text{def}}{=} K^{\bullet,\bullet} \times L^{\bullet,\bullet}$ then we have $G^A = K^{A,A} \times L^{A,A}$. Denote $M^A \stackrel{\text{def}}{=} K^{A,A}$ and $N^A \stackrel{\text{def}}{=} L^{A,A}$, so that $G^A = M^A \times N^A$. The inductive assumptions are $(r,r)^{\updownarrow K} = r^{\updownarrow M}$ and $(r,r)^{\updownarrow L} = r^{\updownarrow N}$. We find:

$$(r,r)^{\updownarrow P} = (r,r)^{\updownarrow K} \boxtimes (r,r)^{\updownarrow L} = r^{\updownarrow M} \boxtimes r^{\updownarrow N} = r^{\updownarrow (M\times N)} = r^{\updownarrow G} \quad .$$

If $P^{\bullet,\bullet} \stackrel{\text{def}}{=} K^{\bullet,\bullet} + L^{\bullet,\bullet}$ with $G^A = M^A + N^A$ and the same inductive assumptions, we get:

$$(r,r)^{\updownarrow P} = (r,r)^{\updownarrow K} \boxplus (r,r)^{\updownarrow L} = r^{\updownarrow M} \boxplus r^{\updownarrow N} = r^{\updownarrow (M+N)} = r^{\updownarrow G} \quad .$$

If $P^{\bullet,\bullet} \stackrel{\text{def}}{=} K^{\bullet,\bullet} \to L^{\bullet,\bullet}$ with $G^A = M^A \to N^A$ and the same inductive assumptions, we find:

$$(r,r)^{\updownarrow P} = (r,r)^{\updownarrow K} \boxtimes (r,r)^{\updownarrow L} = r^{\updownarrow M} \boxtimes r^{\updownarrow N} = r^{\updownarrow (M^\bullet \to N^\bullet)} = r^{\updownarrow G} \quad .$$

If $P^{A,X} \stackrel{\text{def}}{=} S^{A,X,P^{A,X}}$ with a recursion scheme $S^{\bullet,\bullet,\bullet}$, the type constructor $G^\bullet$ is defined by:

$$G^A \stackrel{\text{def}}{=} P^{A,A} = S^{A,A,P^{A,A}} = S^{A,A,G^A} \quad .$$

Denote $Q^{A,X} \stackrel{\text{def}}{=} S^{A,A,X}$ and obtain $G^A = Q^{A,G^A}$. So, the lifting to G is given by $r^{\updownarrow G} = \left(r, \overline{r^{\updownarrow G}}\right)^{\updownarrow Q}$. Now write the definition of lifting to P and simplify:

$$\begin{aligned}
& (r,r)^{\updownarrow P} = \left(r, r, \overline{(r,r)^{\updownarrow P}}\right)^{\updownarrow S} \\
\text{inductive assumption } \overline{(r,r)^{\updownarrow P}} = \overline{r^{\updownarrow G}} : \quad & = \left(r, r, \overline{r^{\updownarrow G}}\right)^{\updownarrow S} \\
\text{inductive assumption } \left(r, \overline{r^{\updownarrow G}}\right)^{\updownarrow Q} = \left(r, r, \overline{r^{\updownarrow G}}\right)^{\updownarrow S} : \quad & = \left(r, \overline{r^{\updownarrow G}}\right)^{\updownarrow Q} \\
\text{definition of lifting to } G : \quad & = r^{\updownarrow G} \quad .
\end{aligned}$$

If $P^{A,X} \stackrel{\text{def}}{=} \forall Z.\, S^{Z,A,X}$, the type constructor $G^\bullet$ is $G^A = \forall Z.\, S^{Z,A,A}$. Denote $Q^{Z,A} \stackrel{\text{def}}{=} S^{Z,A,A}$ and obtain $G^A = \forall Z.\, Q^{Z,A}$. The lifting $(r,r)^{\updownarrow P}$ is a relation of type $\forall U.\, S^{U,A,A} \leftrightarrow \forall V.\, S^{V,A,A}$ written as:

$$\begin{aligned}
\forall (U,V) : \quad & \left((r,r)^{\updownarrow \forall Z.\, S^{Z,\bullet,\bullet}}\right)^{U,V} = \forall w^{:U\leftrightarrow V}.\, (w,r,r)^{\updownarrow S^{\bullet,\bullet,\bullet}} \\
\text{inductive assumption about lifting to } S : \quad & = \forall w^{:U\leftrightarrow V}.\, (w,r)^{\updownarrow Q} \\
\text{definition of lifting to } G : \quad & = \left(r^{\updownarrow G}\right)^{U,V} \quad .
\end{aligned}$$

D.3.4 Relational parametricity theorem

Our goal is to prove that any fully parametric function $t : \forall A.\, P^A \to Q^A$ obeys the law (D.10):

$$\forall A, B.\, \forall r^{:A\leftrightarrow B}.\, (t^A, t^B) \in r^{\updownarrow P} \boxtimes r^{\updownarrow Q} \quad .$$

The proof will need to go by induction on the structure of the code of the function t, which is built from smaller sub-expressions using the nine code constructions of Table D.2. The inductive assumption is that all sub-expressions already satisfy the relational naturality law. An immediate difficulty with this approach is that the sub-expressions of t may not have the type signature of a function and may also contain free variables. To illustrate, write the code of t as $t = z^{:P^A} \to \text{expr}(z)$, or in Scala:

```
def t[A] = { z: P[A] => expr(z) }
```

where "`expr(z)`" is the function's body. That function body *itself* does not necessarily have a type signature of the form $K^A \to L^A$. Also, "`expr(z)`" may contain z as a free variable defined outside the scope of "`expr(z)`", and the type of z may depend on the type parameter A. So, we cannot directly apply the relational naturality law (D.10) to the subexpressions, which prevents us from using induction. The relational naturality law needs to be reformulated to describe function *bodies*, i.e., arbitrary expressions that may contain externally defined variables. A suitable formulation of the relational naturality law is given in the next definition and will be the goal of Statement D.3.4.3.

Definition D.3.4.1 (relational naturality law) Consider any expression $t : \forall A.\, Q^A$ containing a single free variable $x^{:P^A}$, where $P^\bullet$ and $Q^\bullet$ are any type constructors. Define the **binding function** $t' : \forall A.\, P^A \to Q^A$ such that $t = \tilde{t}(x)$ and $\tilde{t}$ has no free variables. (The binding function describes how the expression t depends on its free variable x.) Then the **relational naturality law** of t is written as:

$$\forall (A, B).\, \forall r^{:A \leftrightarrow B}.\, (\tilde{t}^A, \tilde{t}^B) \in r^{\updownarrow P} \boxtimes r^{\updownarrow Q} \quad . \tag{D.16}$$

If t contains several free variables ($x_1^{:P_1^A}$, $x_2^{:P_2^A}$, etc.), we define the binding function $\tilde{t}$ as a curried function of all the free variables. For example, with *two* free variables we will have $t = \tilde{t}(x_1)(x_2)$, so that $\tilde{t}^A$ will have type $P_1^A \to P_2^A \to Q^A$. Then the relational naturality law is written as:

$$\forall (A, B).\, \forall r^{:A \leftrightarrow B}.\, (\tilde{t}^A, \tilde{t}^B) \in r^{\updownarrow P_1} \boxtimes r^{\updownarrow P_2} \boxtimes r^{\updownarrow Q} \quad . \tag{D.17}$$

The generalization to any number of free variables is straightforward. □

If t contains no free variables, we may still write t as $\tilde{t}(x)$ where $\tilde{t}$ does not depend on x, setting $P^A \stackrel{\text{def}}{=} \mathbb{1}$ for simplicity. In that case, Eq. (D.16) is simplified to:

$$\forall (A, B).\, \forall r^{:A \leftrightarrow B}.\, \forall x_1^{:\mathbb{1}}, x_2^{:\mathbb{1}}.\quad \text{if } x_1 = x_2 \text{ then } (\tilde{t}^A(x_1), \tilde{t}^B(x_2)) \in r^{\updownarrow Q} \quad ,$$

$$\text{or equivalently :} \quad \forall (A, B).\, \forall r^{:A \leftrightarrow B}.\, (t^A, t^B) \in r^{\updownarrow Q} \quad . \tag{D.18}$$

Even in that case, to save time, we will keep using Eq. (D.16) and writing $t = \tilde{t}(x)$.

When an expression contains more than one free variable, we can gather all the free variables into a tuple. This creates an equivalent expression with just one free variable:

Statement D.3.4.2 The relational naturality law for an expression $t : \forall A.\, Q^A$ containing two free variables $z_1^{:K^A}$ and $z_2^{:L^A}$ and the binding function $\tilde{t}(z_1)(z_2) = t$ is equivalent to the relational naturality law for the expression u with *one* free variable h defined as:

$$h^{:K^A \times L^A} \stackrel{\text{def}}{=} z_1 \times z_2 \quad , \qquad u \stackrel{\text{def}}{=} \tilde{u}(h) \stackrel{\text{def}}{=} \tilde{t}(h \triangleright \pi_1)(h \triangleright \pi_2) \quad .$$

Proof The relational naturality laws for t and u say that, for all types A, B, and relations $r^{:A \leftrightarrow B}$:

$$(\tilde{t}^A, \tilde{t}^B) \in r^{\updownarrow K} \boxtimes r^{\updownarrow L} \boxtimes r^{\updownarrow Q} \quad , \qquad (\tilde{u}^A, \tilde{u}^B) \in r^{\updownarrow (K \times L)} \boxtimes r^{\updownarrow Q} \quad .$$

We need to show that these two relations are equivalent given the definition of $\tilde{u}$ via $\tilde{t}$. This is similar to the equivalence of curried and uncurried function types: $A \to (B \to C) \cong A \times B \to C$. We write out the definition of the pair mapper operation ($\boxtimes$) and obtain:

$$(\tilde{t}^A, \tilde{t}^B) \in r^{\updownarrow K} \boxtimes r^{\updownarrow L} \boxtimes r^{\updownarrow Q} \quad \text{means} \quad : \quad \forall x_1^{:K^A}, x_2^{:L^A}, y_1^{:K^B}, y_2^{:L^B} : \text{ if } (x_1, y_1) \in r^{\updownarrow K} \text{ and } (x_2, y_2) \in r^{\updownarrow L}$$
$$\text{then } (\tilde{t}(x_1)(x_2), \tilde{t}(y_1)(y_2)) \in r^{\updownarrow Q} \quad ;$$
$$(\tilde{u}^A, \tilde{u}^B) \in r^{\updownarrow (K \times L)} \boxtimes r^{\updownarrow Q} \quad \text{means} \quad : \quad \forall h^{:K^A \times L^A}, w^{:K^B \times L^B} : \text{ if } (h, w) \in r^{\updownarrow (K \times L)}$$
$$\text{then } (\tilde{u}(h), \tilde{u}(w)) \in r^{\updownarrow Q} \quad .$$

By Definition D.3.2.5(c) we have $r^{\updownarrow (K \times L)} = r^{\updownarrow K} \boxtimes r^{\updownarrow L}$, and so we may write:

$$(h, w) \in r^{\updownarrow (K \times L)} \quad \text{if} \quad h = x_1 \times x_2 \quad , \quad w = y_1 \times y_2 \quad , \quad (x_1, x_2) \in r^{\updownarrow K} \quad , \quad \text{and} \quad (y_1, y_2) \in r^{\updownarrow L} \quad .$$

When $h = x_1 \times x_2$ and $w = y_1 \times y_2$, we will have $\tilde{u}(h) = \tilde{t}(x_1, x_2)$ and $\tilde{u}(w) = \tilde{t}(y_1, y_2)$. It follows that the relational naturality laws for t and u are equivalent. □

Due to Statement D.3.4.2, we are allowed to assume that the expression t always has a *single* free variable. This simplifies the formulation of the main theorem:

Statement D.3.4.3 (relational parametricity theorem) Let H and Q be any fully parametric type constructors. Any fully parametric expression $t : \forall A.\, Q^A$ of the form $t = \forall A.\, \tilde{t}^A(h)$ containing a single free variable $h^{:H^A}$ satisfies the relational naturality law (D.16).

Proof By assumption, t is built up from the nine constructions of Table D.2. So, one of these nine constructions is at the top level in the syntax tree of t. For each of those constructions, we will prove that t satisfies Eq. (D.16) as long as all its sub-expressions do. Throughout the proof, all relational naturality laws will involve an arbitrary relation $r^{:A\leftrightarrow B}$ between arbitrary types A, B. For brevity, we will not write the quantifiers $\forall A, B, r^{:A\leftrightarrow B}$ in front of all formulas.

To shorten the proof further, we note that in every inductive case the expression t and all its sub-expressions will contain the same free variable h. So, all relational naturality laws will follow the pattern "for all $h_1^{:H^A}$ and $h_2^{:H^B}$ satisfying the relation $(h_1, h_2) \in r^{\updownarrow H}$, some other values are in some other relation". Let us choose arbitrary but fixed values $h_1^{:H^A}$ and $h_2^{:H^B}$ satisfying $(h_1, h_2) \in r^{\updownarrow H}$, denote $t_1 \stackrel{\text{def}}{=} \tilde{t}^A(h_1)$ and $t_2 \stackrel{\text{def}}{=} \tilde{t}^B(h_2)$, and simplify the naturality law (D.16) to:

$$(t_1, t_2) \in r^{\updownarrow Q} \quad \text{where} \quad t_1 \stackrel{\text{def}}{=} \tilde{t}^A(h_1) \text{ and } t_2 \stackrel{\text{def}}{=} \tilde{t}^B(h_2) \quad . \tag{D.19}$$

We will now prove this form of the law. Inductive assumptions will always begin with "for all h_1 and h_2 ...", so we are allowed to substitute the fixed values h_1, h_2 into each of the inductive assumptions and write those assumptions also in the form of Eq. (D.19).

Use unit value In this case, $t \stackrel{\text{def}}{=} 1$ and has the unit type ($Q^A \stackrel{\text{def}}{=} \mathbb{1}$). Since t contains no free variables, the law (D.19) becomes:

$$(1, 1) \in r^{\updownarrow Q} \quad .$$

This holds because by Definition D.3.2.5(a) we have $r^{\updownarrow Q} = \text{id}$ regardless of r.

The same proof applies for $t \stackrel{\text{def}}{=} c$ where $c^{:C}$ is a fixed value of a fixed type C.

Use argument In this case, $t \stackrel{\text{def}}{=} h$ where $h^{:H^A}$ is the free variable (say, the argument of the function whose body is t). So, we must have $Q = H$ and $\tilde{t} = \forall A.\, \text{id}^{:H^A \to H^A}$. The law (D.19) then becomes:

$$\begin{aligned} & (t_1, t_2) \in r^{\updownarrow H} \quad \text{where} \quad t_1 \stackrel{\text{def}}{=} h_1 \text{ and } t_2 \stackrel{\text{def}}{=} h_2 \quad , \\ \text{or equivalently :} \quad & (h_1, h_2) \in r^{\updownarrow H} \quad . \end{aligned}$$

The last condition holds trivially, since $(h_1, h_2) \in r^{\updownarrow H}$ is already assumed.

Create function In this case, $t \stackrel{\text{def}}{=} \forall A.\, p^{:P^A} \to \tilde{g}^A(h)(p)$ where the sub-expression $\tilde{g}^A(h)(p) : G^A$ contains two free variables ($h^{:H^A}$ and $p^{:P^A}$). Because $\tilde{g}(h)(p)$ is defined with curried arguments, the binding function $\tilde{t}$ satisfies:

$$\tilde{t}(h) = t = p \to \tilde{g}(h)(p) = \tilde{g}(h) \quad , \qquad \text{or equivalently :} \quad \tilde{t} = \tilde{g}.$$

The type of $\tilde{g}$ is $H^A \to Q^A$, where we set $Q^A \stackrel{\text{def}}{=} P^A \to G^A$. Definition D.3.2.5(e) gives $r^{\updownarrow Q} = r^{\updownarrow P} \boxtimes r^{\updownarrow G}$. So, the inductive assumption for g may be written as:

$$(g_1, g_2) \in r^{\updownarrow P} \boxtimes r^{\updownarrow G} \quad \text{where} \quad g_1 \stackrel{\text{def}}{=} \tilde{g}^A(h_1) = t_1 \text{ and } g_2 \stackrel{\text{def}}{=} \tilde{g}^B(h_2) = t_2 \quad .$$

This is equivalent to:

$$(t_1, t_2) \in r^{\updownarrow P} \boxtimes r^{\updownarrow G} = r^{\updownarrow Q} \quad ,$$

which is the law (D.19) we needed to prove.

Use function In this case, $t = \tilde{t}(h) \stackrel{\text{def}}{=} k(p)$ where $k^{:\forall A.\, P^A \to Q^A}$ and $p^{:\forall A.\, P^A}$ are some sub-expressions. Both k and p may contain $h^{:H^A}$ as a free variable: $k \stackrel{\text{def}}{=} \tilde{k}(h)$ and $p \stackrel{\text{def}}{=} \tilde{p}(h)$. Denote for brevity $K^A \stackrel{\text{def}}{=} P^A \to Q^A$. The inductive assumptions say that the laws (D.19) already hold for both k and p:

$$\begin{aligned} \text{for } k : \quad & (k_1, k_2) \in r^{\updownarrow K} = r^{\updownarrow P} \boxtimes r^{\updownarrow Q} \quad \text{where} \quad k_1 \stackrel{\text{def}}{=} \tilde{k}^A(h_1) \text{ and } k_2 \stackrel{\text{def}}{=} \tilde{k}^B(h_2) \quad , \\ \text{for } p : \quad & (p_1, p_2) \in r^{\updownarrow P} \quad \text{where} \quad p_1 \stackrel{\text{def}}{=} \tilde{p}^A(h_1) \text{ and } p_2 \stackrel{\text{def}}{=} \tilde{p}^B(h_2) \quad . \end{aligned}$$

We need to show that:

$$(t_1, t_2) \in r^{\updownarrow Q} \qquad \text{where} \quad t_1 \stackrel{\text{def}}{=} \tilde{t}^A(h_1) = k_1(p_1) \text{ and } t_2 \stackrel{\text{def}}{=} \tilde{t}^B(h_2) = k_2(p_2) \quad .$$

By definition of $\boxtimes$, the inductive assumption $(k_1, k_2) \in r^{\updownarrow P} \boxtimes r^{\updownarrow Q}$ means:

$$(k_1(p_1), k_2(p_2)) \in r^{\updownarrow Q} \quad \text{whenever} \quad (p_1, p_2) \in r^{\updownarrow P} \quad .$$

Since $(p_1, p_2) \in r^{\updownarrow P}$ already holds by the other inductive assumption, we obtain $(t_1, t_2) \in r^{\updownarrow Q}$.

Create tuple In this case, $t = \tilde{t}(h) \stackrel{\text{def}}{=} k^{:K^A} \times l^{:L^A}$, where the sub-expressions k and l contain the free variable $h^{:H^A}$ via $k \stackrel{\text{def}}{=} \tilde{k}(h)$ and $l \stackrel{\text{def}}{=} \tilde{l}(h)$. We have $Q^A \stackrel{\text{def}}{=} K^A \times L^A$, so we use Definition D.3.2.5(c) for the lifting $r^{\updownarrow Q}$ to get $r^{\updownarrow Q} = r^{\updownarrow K} \boxtimes r^{\updownarrow L}$. We need to prove that:

$$(t_1, t_2) \in r^{\updownarrow Q} \quad \text{or equivalently :} \quad (k_1 \times l_1, k_2 \times l_2) \in r^{\updownarrow K} \boxtimes r^{\updownarrow L} \quad ,$$
$$\text{where} \quad t_1 \stackrel{\text{def}}{=} k_1 \times l_1 \stackrel{\text{def}}{=} \tilde{k}^A(h_1) \times \tilde{l}^A(h_1) \quad , \qquad t_2 \stackrel{\text{def}}{=} k_2 \times l_2 \stackrel{\text{def}}{=} \tilde{k}^A(h_2) \times \tilde{l}^A(h_2) \quad .$$

The inductive assumptions are that the relational naturality law (D.19) holds for k and l:

$$(k_1, k_2) \in r^{\updownarrow K} \quad , \qquad (l_1, l_2) \in r^{\updownarrow L} \quad .$$

By definition of $\boxtimes$ for relations, we obtain the required property: $(k_1 \times l_1, k_2 \times l_2) \in r^{\updownarrow K} \boxtimes r^{\updownarrow L}$.

Use tuple It is sufficient to consider the case $t \stackrel{\text{def}}{=} \pi_1(g)$ where $g^{:\forall A.\, Q^A \times L^A}$ is a sub-expression that contains the free variable $h^{:H^A}$. The proof for $t = \pi_2(g)$ is analogous.

We need to prove that t satisfies the relational naturality law if g does. Define $g = \tilde{g}(h)$ and:

$$g_1 \stackrel{\text{def}}{=} \tilde{g}^A(h_1) \quad , \qquad g_2 \stackrel{\text{def}}{=} \tilde{g}^B(h_2) \quad , \qquad t_1 \stackrel{\text{def}}{=} \pi_1(g_1) \quad , \qquad t_2 \stackrel{\text{def}}{=} \pi_1(g_2) \quad .$$

The relational naturality law of g, which holds by the inductive assumption, is:

$$(g_1, g_2) \in r^{\updownarrow (Q \times L)} \quad .$$

By Definition D.3.2.5(c) for the lifting $r^{\updownarrow (Q \times L)}$, we have:

$$(g_1, g_2) \in r^{\updownarrow (Q \times L)} \quad \text{means} \quad (\pi_1(g_1), \pi_1(g_2)) \in r^{\updownarrow Q} \text{ and } (\pi_2(g_1), \pi_2(g_2)) \in r^{\updownarrow L} \quad .$$

The condition $(\pi_1(g_1), \pi_1(g_2)) \in r^{\updownarrow Q}$ is the same as the relational naturality law of t.

Create disjunction We consider the case $t \stackrel{\text{def}}{=} g + \mathbb{0}$ where $Q^A \stackrel{\text{def}}{=} K^A + L^A$ and $g^{:\forall A.\, K^A}$ is a sub-expression that contains the free variable $h^{:H^A}$ via $g \stackrel{\text{def}}{=} \tilde{g}(h)$. Define for convenience:

$$g_1 \stackrel{\text{def}}{=} \tilde{g}^A(h_1) \quad , \qquad g_2 \stackrel{\text{def}}{=} \tilde{g}^B(h_2) \quad , \qquad t_1 = g_1 + \mathbb{0} \quad , \qquad t_2 = g_2 + \mathbb{0} \quad .$$

The inductive assumption is that $g(h)$ satisfies its relational naturality law, which is $(g_1, g_2) \in r^{\updownarrow K}$. By Definition D.3.2.5(d) for the lifting $r^{\updownarrow (K+L)}$, we have:

$$(t_1, t_2) = (g_1 + \mathbb{0}, g_2 + \mathbb{0}) \in r^{\updownarrow (K+L)} = r^{\updownarrow K} \boxplus r^{\updownarrow L} \text{ when } (g_1, g_2) \in r^{\updownarrow K} \quad .$$

So, the relational naturality law of t holds. The proof for $t \stackrel{\text{def}}{=} \mathbb{0} + g$ is analogous.

Use disjunction In this case, $t \stackrel{\text{def}}{=} \forall A. \begin{array}{c|c} & G^A \\ \hline K^A & p \\ L^A & q \end{array}$ is a pattern-matching function of type $\forall A.\, Q^A$ with $Q^A \stackrel{\text{def}}{=} K^A + L^A \to G^A$. The sub-expressions $p^{:\forall A.\, K^A \to G^A}$ and $q^{:\forall A.\, L^A \to G^A}$ contain the free variable $h^{:H^A}$. We define:

$$p \stackrel{\text{def}}{=} \tilde{p}(h) \quad , \quad p_1 \stackrel{\text{def}}{=} \tilde{p}^A(h_1) \quad , \quad p_2 \stackrel{\text{def}}{=} \tilde{p}^B(h_2) \quad , \qquad q \stackrel{\text{def}}{=} \tilde{q}(h) \quad , \quad q_1 \stackrel{\text{def}}{=} \tilde{q}^A(h_1) \quad , \quad q_2 \stackrel{\text{def}}{=} \tilde{q}^B(h_2) \quad .$$

Then we have:

$$t_1 = \begin{array}{|c||c|} \hline & G^A \\ \hline K^A & p_1 \\ L^A & q_1 \\ \hline \end{array} \quad , \qquad t_2 = \begin{array}{|c||c|} \hline & G^B \\ \hline K^B & p_2 \\ L^B & q_2 \\ \hline \end{array} \quad .$$

By the inductive assumption, the relational naturality law already holds for p and q:

$$\text{if } (k_1^{:K^A}, k_2^{:K^B}) \in r^{\updownarrow K} \text{ then } \big(p_1(k_1), p_2(k_2)\big) \in r^{\updownarrow G} \quad ,$$
$$\text{if } (l_1^{:L^A}, l_2^{:L^B}) \in r^{\updownarrow L} \text{ then } \big(q_1(l_1), q_2(l_2)\big) \in r^{\updownarrow G} \quad .$$

To derive the specific form of the law for t, we use Definition D.3.2.5(e) for the lifting $r^{\updownarrow Q}$:

$$\text{if } (x_1^{:K^A+L^A}, x_2^{:K^B+L^B}) \in r^{\updownarrow (K+L)} \text{ then } \big(t_1(x_1), t_2(x_2)\big) \in r^{\updownarrow G} \quad .$$

By Definition D.3.2.5(d), the values x_1 and x_2 are in relation $r^{\updownarrow (K+L)}$ only if both x_1 and x_2 are in the same part of the disjunction $(K^\bullet + L^\bullet)$. We consider separately the case when they are in the left part or in the right part.

If both x_1 and x_2 in the left part, we can write $x_1 = k_1^{:K^A} + \mathbb{0}$ and $x_2 = k_2^{:K^B} + \mathbb{0}$ with some k_1 and k_2. Then the condition $(x_1, x_2) \in r^{\updownarrow (K+L)}$ is equivalent to $(k_1, k_2) \in r^{\updownarrow K}$, while the values $t_1(x_1)$ and $t_2(x_2)$ are expressed as:

$$t_1(x_1) = (k_1 + \mathbb{0}) \triangleright \begin{array}{|c||c|} \hline & G^A \\ \hline K^A & p_1 \\ L^A & q_1 \\ \hline \end{array} = p_1(k_1) \quad , \qquad t_2(x_2) = (k_2 + \mathbb{0}) \triangleright \begin{array}{|c||c|} \hline & G^B \\ \hline K^B & p_2 \\ L^B & q_2 \\ \hline \end{array} = p_2(k_2) \quad .$$

So, the conclusion of the relational naturality law of t holds due to the relational law of p:

$$\big(t_1(x_1), t_2(x_2)\big) = \big(p_1(k_1), p_2(k_2)\big) \in r^{\updownarrow G} \quad .$$

A similar argument proves the law for the case when both x_1 and x_2 in the right part. We write $x_1 = \mathbb{0} + l_1^{:L^A}$ and $x_2 = \mathbb{0} + l_2^{:L^B}$ and reduce the relational naturality law of t to that of q.

Recursive call In this case, $t \stackrel{\text{def}}{=} g$ where g is a recursive call to a function (defined outside that expression). When proving a law of a recursively defined function, we may assume that the law holds for recursive calls to that function. So, the inductive assumption says that g is some expression for which the relational naturality law already holds. It then holds for t since $t = g$.

This completes the proof of the relational parametricity theorem. □

The relational formulation of parametricity is powerful but hard to use for deriving laws as equations rather than as relations. To convert the relational parametricity law such as Eq. (D.18) into an equation with functions, we first choose the relation r as $r \stackrel{\text{def}}{=} \langle f \rangle$ with a suitable function f. Then we need to express the lifted relation $r^{\updownarrow Q}$ via an equation and derive the corresponding equation for t. However, Example D.3.2.6(e) shows that for sufficiently complicated type constructors Q the lifted relation $\langle f \rangle^{\updownarrow Q}$ will *not* be expressible as an equation.

Nevertheless, this problem does not arise for most type signatures found in practice, and the relational naturality law can be reduced to the dinaturality law (D.4) via the wedge law (D.3). In a few cases, a more powerful property called the "strong" dinaturality law is required. In this book, strong dinaturality is only used to prove Statements 12.3.1.3, 13.7.3.1, and 13.7.3.2, while in all other places the wedge law is sufficient. The following sections will prove that the wedge law and the strong dinaturality law are consequences of the relational parametricity theorem.

D.3.5 Deriving the wedge law from relational parametricity

This section derives the wedge law from the relational naturality law of `xmap`.[5]

Statement D.3.5.1 **(a)** For any fully parametric profunctor $P^{X,Y}$, define $T^A \stackrel{\text{def}}{=} P^{A,A}$. Then:

$$\text{for all } X, Y, f^{:X\to Y}, x^{:P^{X,X}}, y^{:P^{Y,Y}} \quad : \quad \text{if} \quad (x,y) \in \langle f \rangle^{\updownarrow T} \quad \text{then} \quad x \triangleright f^{\uparrow P^{X,\bullet}} = y \triangleright f^{\downarrow P^{\bullet,Y}} \quad .$$

In other words, the (P, f)-wedge relation is a consequence of the lifted relation $\langle f \rangle$.

(b) Any fully parametric value t of type $\forall A.\, P^{A,A}$ satisfies the wedge law (D.3):

$$\text{for all } X, Y, f^{:X\to Y} \quad : \quad t^X \triangleright f^{\uparrow P^{X,\bullet}} = t^Y \triangleright f^{\downarrow P^{\bullet,Y}} \quad .$$

In other words, t^X and t^Y are always in the (P, f)-wedge relation for any function $f^{:X\to Y}$.

Proof **(a)** All profunctors P have a `xmap` method with the type signature:

$$\text{xmap}_P : \forall A, B, C, D.\ (B \to A) \to (C \to D) \to P^{A,C} \to P^{B,D} \quad .$$

Since $P^{X,Y}$ is a fully parametric profunctor, its `xmap` satisfies the commutativity law (Section D.2.1):

$$\text{xmap}_P(f^{:B\to A})(g^{:C\to D}) = f^{\downarrow P^{\bullet,C}} \fatsemi g^{\uparrow P^{A,\bullet}} = g^{\uparrow P^{B,\bullet}} \fatsemi f^{\downarrow P^{\bullet,D}} \quad .$$

Then the property we need to prove can be written in terms of `xmap` as:

$$\text{if} \quad (x,y) \in \langle f \rangle^{\updownarrow T} \quad \text{then} \quad \text{xmap}_P(\text{id})(f)(x) = \text{xmap}_P(f)(\text{id})(y) \quad . \tag{D.20}$$

Statement D.3.4.3 shows that `xmap` also satisfies a relational naturality law. Since `xmap` has four type parameters (A, B, C, D), its relational naturality law involves 8 arbitrary types (denote them by A, B, C, D, A', B', C', D') and 4 arbitrary relations (denote them by $a^{:A\leftrightarrow A'}$, $b^{:B\leftrightarrow B'}$, $c^{:C\leftrightarrow C'}$, and $d^{:D\leftrightarrow D'}$). With these notations, we write the relational naturality law of `xmap` as:

$$\forall (a,b,c,d) \quad : \quad (\text{xmap}_P^{A,B,C,D}, \text{xmap}_P^{A',B',C',D'}) \in (b \boxtimes a) \boxtimes (c \boxtimes d) \boxtimes (a,c)^{\updownarrow P} \boxtimes (b,d)^{\updownarrow P} \quad , \tag{D.21}$$

where $(a,c)^{\updownarrow P}$ and $(b,d)^{\updownarrow P}$ are simultaneous liftings to the profunctor P (see Definition D.3.3.2). Writing out the definition of the pair mapper ($\boxtimes$), we express this law as:

$$\begin{aligned} &\forall \big(a^{:A\leftrightarrow A'}, b^{:B\leftrightarrow B'}, c^{:C\leftrightarrow C'}, d^{:D\leftrightarrow D'}, g^{:B\to A}, h^{:B'\to A'}, k^{:C\to D}, l^{:C'\to D'}, x^{:P^{A,C}}, y^{:P^{A',C'}}\big) \quad : \\ &\quad \text{if} \quad (g,h) \in b \boxtimes a \quad , \quad (k,l) \in c \boxtimes d \quad , \quad \text{and} \quad (x,y) \in (a,c)^{\updownarrow P} \quad , \\ &\quad \text{then} \quad \big(\text{xmap}_P(g)(k)(x), \text{xmap}_P(h)(l)(y)\big) \in (b,d)^{\updownarrow P} \quad . \end{aligned} \tag{D.22}$$

The naturality law of `xmap` now has the form of a condition that holds for arbitrary choices of 8 types, 4 relations, and 4 functions, in addition to the values x and y. The plan of the proof is to substitute certain carefully chosen functions and relations into Eq. (D.22) in order to derive Eq. (D.20).

We note that Eq. (D.20) says that two `xmap(...)` values must be equal (both values having type $P^{X,Y}$). This can be obtained from Eq. (D.22) only if the relation $(b,d)^{\updownarrow P}$ is an identity relation of type $P^{X,Y} \leftrightarrow P^{X,Y}$. We will prove in Statement D.3.5.2 below that a simultaneous lifting of two identity relations gives again an identity relation. This suggests choosing $b = \text{id}$ and $d = \text{id}$, which also requires us to set the types as $B = B' = X$ and $D = D' = Y$.

The relations a and c need to be chosen such that the precondition $(x,y) \in (a,c)^{\updownarrow P}$ in Eq. (D.22) reproduces the precondition $(x,y) \in \langle f \rangle^{\updownarrow T}$ in Eq. (D.20). Statement D.3.3.3 shows that $r^{\updownarrow T} = (r,r)^{\updownarrow P}$

[5] The proof is based on the blog post by B. Milewski (https://bartoszmilewski.com/2017/04/11/). Another derivation of the dinaturality law from the relational parametricity theorem was given by J. Voigtländer in the paper https://arxiv.org/pdf/1908.07776.pdf. A proof of the dinaturality law based on obtaining the syntactic normal form of dinatural transformations (and *without* using relational parametricity) was shown by J. de Lataillade in the paper "Dinatural terms in System F": see https://www.irif.fr/~delatail/dinat.pdf

for any relation r. In particular, $\langle f\rangle^{\updownarrow T} = (\langle f\rangle, \langle f\rangle)^{\updownarrow P}$. So, we need to set $a = c = \langle f\rangle$ and the corresponding types $A = C = X$ and $A' = C' = Y$.

The functions g, h, k, and l in Eq. (D.22) are chosen such that:

$$\mathrm{xmap}_P(g)(k)(x) = \mathrm{xmap}_P(\mathrm{id})(f)(x) \quad , \qquad \mathrm{xmap}_P(h)(l)(x) = \mathrm{xmap}_P(f)(\mathrm{id})(x) \quad .$$

This implies $g = \mathrm{id}$, $h = k = f$, and $l = \mathrm{id}$. With these choices, the preconditions $(g, h) \in b \boxtimes a$ and $(k, l) \in c \boxtimes d$ will automatically hold in Eq. (D.22). So, the conclusion of Eq. (D.22) also holds, which proves Eq. (D.20).

(b) The relational parametricity theorem (Statement D.3.4.3) gives $(t^X, t^Y) \in c^{\updownarrow T}$ for any relation $c^{:X\leftrightarrow Y}$. We can now choose $c \stackrel{\text{def}}{=} \langle f\rangle$ and use the result of part **(a)** where we set $x \stackrel{\text{def}}{=} t^X$ and $y \stackrel{\text{def}}{=} t^Y$. Then we obtain $t^X \triangleright f^{\uparrow P^{X,\bullet}} = t^Y \triangleright f^{\downarrow P^{\bullet,Y}}$, which is the wedge law of t. □

While proving Statement D.3.5.1, we used the **identity laws** of the relational lifting:

Statement D.3.5.2 **(a)** An identity relation can be removed from a simultaneous lifting. For instance, given any type constructor $H^{\bullet,\bullet}$, a fixed type T, and any relation $r^{:A\leftrightarrow B}$, we have:

$$(r, \mathrm{id}^{:T\leftrightarrow T})^{\updownarrow H} = r^{\updownarrow G} \quad , \qquad \text{where we defined :} \quad G^A \stackrel{\text{def}}{=} H^{A,T} \quad .$$

(b) Lifting one or more identity relations produces again an identity relation:

$$\text{for any } G^\bullet, H^{\bullet,\bullet}, \text{etc. :} \quad (\mathrm{id}^{:A\leftrightarrow A})^{\updownarrow G} = \mathrm{id}^{:G^A\leftrightarrow G^A} \quad , \qquad (\mathrm{id}^{:A\leftrightarrow A}, \mathrm{id}^{:X\leftrightarrow X})^{\updownarrow H} = \mathrm{id}^{:H^{A,X}\leftrightarrow H^{A,X}} \quad , \quad \text{etc.}$$

Proof **(a)** Enumerate all cases of Definition D.3.3.2 for $H^{\bullet,\bullet}$.

If $H^{A,X} \stackrel{\text{def}}{=} Z$ with a fixed type Z, we have also $G^A = Z$. Then $(r, s)^{\updownarrow H} \stackrel{\text{def}}{=} \mathrm{id}^{:Z\leftrightarrow Z}$ and $r^{\updownarrow G} = \mathrm{id}^{:Z\leftrightarrow Z}$ for any relations r, s. So, $(r, \mathrm{id})^{\updownarrow H} = r^{\updownarrow G}$.

If $H^{A,X} \stackrel{\text{def}}{=} A$, we have $G^A = A$. Then $(r, s)^{\updownarrow H} \stackrel{\text{def}}{=} r$ and $r^{\updownarrow G} = r$ for any relation r, so we obtain $(r, \mathrm{id})^{\updownarrow H} = r^{\updownarrow G}$.

The next three similar cases use the inductive assumptions $(r, \mathrm{id})^{\updownarrow K} = r^{\updownarrow K^{\bullet,T}}$ and $(r, \mathrm{id})^{\updownarrow L} = r^{\updownarrow L^{\bullet,T}}$:

$$\begin{aligned}
&\text{if } H^{A,X} \stackrel{\text{def}}{=} K^{A,X} \times L^{A,X} : && (r, \mathrm{id})^{\updownarrow H} = (r, \mathrm{id})^{\updownarrow K} \boxtimes (r, \mathrm{id})^{\updownarrow L} = r^{\updownarrow K^{\bullet,T}} \boxtimes r^{\updownarrow L^{\bullet,T}} = r^{\updownarrow (K^{\bullet,T} \times L^{\bullet,T})} = r^{\updownarrow G} \quad ; \\
&\text{if } H^{A,X} \stackrel{\text{def}}{=} K^{A,X} + L^{A,X} : && (r, \mathrm{id})^{\updownarrow H} = (r, \mathrm{id})^{\updownarrow K} \boxplus (r, \mathrm{id})^{\updownarrow L} = r^{\updownarrow K^{\bullet,T}} \boxplus r^{\updownarrow L^{\bullet,T}} = r^{\updownarrow (K^{\bullet,T} + L^{\bullet,T})} = r^{\updownarrow G} \quad ; \\
&\text{if } H^{A,X} \stackrel{\text{def}}{=} K^{A,X} \to L^{A,X} : && (r, \mathrm{id})^{\updownarrow H} = (r, \mathrm{id})^{\updownarrow K} \boxtimes (r, \mathrm{id})^{\updownarrow L} = r^{\updownarrow K^{\bullet,T}} \boxtimes r^{\updownarrow L^{\bullet,T}} = r^{\updownarrow (K^{\bullet,T} \to L^{\bullet,T})} = r^{\updownarrow G} \quad .
\end{aligned}$$

If $H^{A,X} \stackrel{\text{def}}{=} S^{A,X,H^{A,X}}$ with a recursion scheme $S^{\bullet,\bullet,\bullet}$, the type constructor $G^\bullet$ is defined by:

$$G^A \stackrel{\text{def}}{=} H^{A,T} = S^{A,T,H^{A,T}} = S^{A,T,G^A} \quad .$$

So, the lifting to G is given by $r^{\updownarrow G} = \left(r, \overline{r^{\updownarrow G}}\right)^{\updownarrow S^{\bullet,T,\bullet}}$. Write the definition of lifting to H and simplify:

$$\begin{aligned}
& && (r, \mathrm{id})^{\updownarrow H} = \left(r, \mathrm{id}, \overline{(r, \mathrm{id})^{\updownarrow H}}\right)^{\updownarrow S} \\
&\text{inductive assumption } \overline{(r, \mathrm{id})^{\updownarrow H}} = \overline{r^{\updownarrow G}} : && = \left(r, \underline{\mathrm{id}}, \overline{r^{\updownarrow G}}\right)^{\updownarrow S} \\
&\text{inductive assumption about lifting id to } S^{\bullet,\bullet,\bullet} : && = \left(r, \overline{r^{\updownarrow G}}\right)^{\updownarrow S^{\bullet,T,\bullet}} \\
&\text{definition of lifting to } G : && = r^{\updownarrow G} \quad .
\end{aligned}$$

If $H^{A,X} \stackrel{\text{def}}{=} \forall Z.\, S^{Z,A,X}$, the type constructor $G^\bullet$ is $G^A = \forall Z.\, S^{Z,A,T}$. The lifting $(r, \mathrm{id})^{\updownarrow H}$ is a relation of type $\forall U.\, S^{U,A,X} \leftrightarrow \forall V.\, S^{V,A,X}$ written as:

$$\begin{aligned}
& && \forall (U, V) : \quad \left((r, \mathrm{id})^{\updownarrow \forall Z.\, S^{Z,\bullet,\bullet}}\right)^{U,V} = \forall w^{:U\leftrightarrow V}.\, (w, r, \underline{\mathrm{id}})^{\updownarrow S^{\bullet,\bullet,\bullet}} \\
&\text{omitting id from lifting to } S : && = \forall w^{:U\leftrightarrow V}.\, (w, r)^{\updownarrow S^{\bullet,\bullet,T}} \\
&\text{definition of lifting to } G : && = (r^{\updownarrow G})^{U,V} \quad .
\end{aligned}$$

Here we used the inductive assumption that identity relations may be omitted from liftings to $S^{\bullet,\bullet,\bullet}$.

(b) Enumerate all cases of Definition D.3.2.5 for $G^\bullet$ or Definition D.3.3.2 for $H^{\bullet,\bullet}$. The proofs are similar, so we will only prove that $(\mathrm{id},\mathrm{id})^{\updownarrow H} = \mathrm{id}$.

If $H^{A,X} \stackrel{\text{def}}{=} Z$ with a fixed type Z, we have $(r,s)^{\updownarrow H} \stackrel{\text{def}}{=} \mathrm{id}^{:Z\leftrightarrow Z}$ for any relations r, s.

If $H^{A,X} \stackrel{\text{def}}{=} A$, we have $(r,s)^{\updownarrow H} \stackrel{\text{def}}{=} r$ for any relation r, so we obtain $(\mathrm{id},\mathrm{id})^{\updownarrow H} = \mathrm{id}$. Similarly if $H^{A,X} \stackrel{\text{def}}{=} X$ then we obtain $(\mathrm{id},\mathrm{id})^{\updownarrow H} = \mathrm{id}$.

The next three similar cases use the inductive assumptions $(\mathrm{id},\mathrm{id})^{\updownarrow K} = \mathrm{id}$ and $(\mathrm{id},\mathrm{id})^{\updownarrow L} = \mathrm{id}$:

$$\begin{aligned}
\text{if } H^{\bullet,\bullet} \stackrel{\text{def}}{=} K^{\bullet,\bullet} \times L^{\bullet,\bullet} :&\quad (\mathrm{id},\mathrm{id})^{\updownarrow H} = (\mathrm{id},\mathrm{id})^{\updownarrow K} \boxtimes (\mathrm{id},\mathrm{id})^{\updownarrow L} = \mathrm{id} \boxtimes \mathrm{id} \quad ;\\
\text{if } H^{\bullet,\bullet} \stackrel{\text{def}}{=} K^{\bullet,\bullet} + L^{\bullet,\bullet} :&\quad (\mathrm{id},\mathrm{id})^{\updownarrow H} = (\mathrm{id},\mathrm{id})^{\updownarrow K} \boxplus (\mathrm{id},\mathrm{id})^{\updownarrow L} = \mathrm{id} \boxplus \mathrm{id} \quad ;\\
\text{if } H^{\bullet,\bullet} \stackrel{\text{def}}{=} K^{\bullet,\bullet} \to L^{\bullet,\bullet} :&\quad (\mathrm{id},\mathrm{id})^{\updownarrow H} = (\mathrm{id},\mathrm{id})^{\updownarrow K} \boxtimes (\mathrm{id},\mathrm{id})^{\updownarrow L} = \mathrm{id} \boxtimes \mathrm{id} \quad .
\end{aligned}$$

It follows from Example D.3.2.4 with $f = \mathrm{id}$ and $g = \mathrm{id}$ that

$$\mathrm{id} \boxtimes \mathrm{id} = \mathrm{id} \quad , \qquad \mathrm{id} \boxplus \mathrm{id} = \mathrm{id} \quad , \qquad \text{and} \qquad \mathrm{id} \boxtimes \mathrm{id} = \mathrm{id} \quad .$$

So, in all three cases we obtain: $(\mathrm{id},\mathrm{id})^{\updownarrow H} = \mathrm{id}$.

If $H^{A,X} \stackrel{\text{def}}{=} S^{A,X,H^{A,X}}$ with a recursion scheme $S^{\bullet,\bullet,\bullet}$, we have:

$$(\mathrm{id},\mathrm{id})^{\updownarrow H} = \left(\mathrm{id},\mathrm{id},\overline{(\mathrm{id},\mathrm{id})^{\updownarrow H}}\right)^{\updownarrow S} = (\mathrm{id},\mathrm{id},\mathrm{id})^{\updownarrow S} \quad ,$$

because by the inductive assumption the identity law holds for the recursive call: $\overline{(\mathrm{id},\mathrm{id})^{\updownarrow H}} = \mathrm{id}$. Another inductive assumption is that the identity law holds for the liftings to $S^{\bullet,\bullet,\bullet}$. So, we get:

$$(\mathrm{id},\mathrm{id})^{\updownarrow H} = (\mathrm{id},\mathrm{id},\mathrm{id})^{\updownarrow S} = \mathrm{id} \quad .$$

If $H^{A,X} \stackrel{\text{def}}{=} \forall Z.\, S^{Z,A,X}$, the lifting $(\mathrm{id},\mathrm{id})^{\updownarrow H}$ is a relation of type $\forall U.\, S^{U,A,X} \leftrightarrow \forall V.\, S^{V,A,X}$ written as:

$$\forall (U,V) : \quad \left((\mathrm{id},\mathrm{id})^{\updownarrow \forall Z.\, S^{Z,\bullet,\bullet}}\right)^{U,V} = \forall w^{:U\leftrightarrow V}.\, (w,\mathrm{id},\mathrm{id})^{\updownarrow H^{\bullet,\bullet,\bullet}} \quad .$$

As we showed in part **(a)**, any identity relation may be omitted from the lifting:

$$\forall w^{:U\leftrightarrow V}.\, (w,\underline{\mathrm{id}^{:A\leftrightarrow A}},\underline{\mathrm{id}^{:X\leftrightarrow X}})^{\updownarrow H^{\bullet,\bullet,\bullet}} = \forall w^{:U\leftrightarrow V}.\, w^{\updownarrow H^{\bullet,A,X}} \quad .$$

By Statement D.3.3.1 used for $H^{\bullet,A,X}$ as a type constructor with one type parameter, the relation $\forall w.\, w^{\updownarrow H^{\bullet,A,X}}$ is the identity relation for the type $\forall Z.\, H^{Z,A,X}$. We find that $(\mathrm{id},\mathrm{id})^{\updownarrow \forall Z.\, S^{Z,\bullet,\bullet}} = \mathrm{id}$. □

D.3.6 Strong dinaturality: definition and general properties

While the dinaturality law is often sufficient for proofs involving parametricity, some of the more complicated cases require a stronger property that we will study in the rest of this Appendix. As motivation, we begin by looking at the wedge law (D.3) in more detail.

We have seen in Statement D.3.5.1 that fully parametric values automatically satisfy the wedge law. The proof showed that the (P,f)-wedge relation is a consequence of the relation $(x,y) \in \langle f\rangle^{\updownarrow T}$ where $T^A \stackrel{\text{def}}{=} P^{A,A}$. Note that the type diagrams for the wedge law (D.3), the dinaturality law (D.4), and the commutativity law (D.7), involve three relations between values $x^{:P^{A,A}}$ and $y^{:P^{B,B}}$:

$$\begin{aligned}
\text{a } (P,f)\text{-wedge relation}: &\quad x \triangleright f^{\uparrow P^{A,\bullet}} = y \triangleright f^{\downarrow P^{\bullet,B}} \quad , \quad \text{or}: \quad (x,y) \in \mathrm{pull}\,(f^{\uparrow P}, f^{\downarrow P}) \quad ,\\
\text{the left part of a dinaturality diagram}: &\quad \exists z^{:P^{B,A}} \quad \text{such that} \quad x = z \triangleright f^{\downarrow P^{\bullet,A}} \text{ and } y = z \triangleright f^{\uparrow P^{B,\bullet}} \quad ,\\
\text{a relation } \langle f\rangle \text{ lifted to } T: &\quad (x,y) \in \langle f\rangle^{\updownarrow T} \quad .
\end{aligned}$$

To make the notation shorter, we define the **pushout relation** denoted by $\mathrm{push}\,(p,q)$:

$$(x^{:A}, y^{:B}) \in \mathrm{push}\,(p,q) \text{ means } \exists z^{:C} \text{ such that } x = p(z) \text{ and } y = q(z) \quad .$$

Here A, B, C are any types and $p^{:C\to A}$ and $q^{:C\to A}$ are any given functions. The left fragment of the dinaturality diagram is then written as $(x,y) \in \text{push}\,(f^{\downarrow P}, f^{\uparrow P})$.

The pullback and pushout relations may be used to reformulate the commutativity law (D.7) as:

$$\text{for any } x^{:P^{A,A}}, y^{:P^{B,B}} \quad : \quad \text{if} \quad (x,y) \in \text{push}\,(f^{\downarrow P}, f^{\uparrow P}) \quad \text{then} \quad (x,y) \in \text{pull}\,(f^{\uparrow P}, f^{\downarrow P}) \quad .$$

We find that the (P,f)-wedge relation, denoted by $\text{pull}\,(f^{\uparrow P}, f^{\downarrow P})$, is a consequence of both the lifted relation $\langle f\rangle^{\updownarrow T}$ and of the relation $\text{push}\,(f^{\downarrow P}, f^{\uparrow P})$, which we may call the (P,f)-pushout relation.

The (P,f)-pushout relation is generally stronger than the lifted relation $\langle f\rangle^{\updownarrow T}$:

Statement D.3.6.1 We assume a fully parametric profunctor $P^{\bullet,\bullet}$, arbitrary types X, Y, and arbitrary values $x^{:P^{X,X}}$, $y^{:P^{Y,Y}}$, and $f^{:X\to Y}$. If $(x,y) \in \text{push}\,(f^{\downarrow P}, f^{\uparrow P})$ then $(x,y) \in \langle f\rangle^{\updownarrow T}$ where $T^A \stackrel{\text{def}}{=} P^{A,A}$.

Proof The relation $(x,y) \in \text{push}\,(f^{\downarrow P}, f^{\uparrow P})$ means that there exists $z^{:P^{Y,X}}$ such that $x = z \triangleright f^{\downarrow P}$ and $y = z \triangleright f^{\uparrow P}$. Rewrite the expressions for x and y using the `xmap` method of P:

$$x = \text{xmap}_P(f)(\text{id})(z) \quad , \qquad y = \text{xmap}_P(\text{id})(f)(z) \quad .$$

We need to show that $(x,y) \in \langle f\rangle^{\updownarrow T}$, which we can write in terms of `xmap` like this:

$$\forall z^{:P^{Y,X}} : \quad \big(\text{xmap}_P(f)(\text{id})(z), \text{xmap}_P(\text{id})(f)(z)\big) \in \langle f\rangle^{\updownarrow T} \quad . \tag{D.23}$$

This equation is similar to the relational naturality law (D.21) of `xmap` that was used in the proof of Statement D.3.5.1:

$$\forall (a,b,c,d) \quad : \quad (\text{xmap}_P^{A,B,C,D}, \text{xmap}_P^{A',B',C',D'}) \in (b \boxtimes a) \boxtimes (c \boxtimes d) \boxtimes (a,c)^{\updownarrow P} \boxtimes (b,d)^{\updownarrow P} \quad .$$

As we did the proof of that statement, we will now choose specific types and relations in this law in order to derive Eq. (D.23). We begin by writing out the definition of $\boxtimes$:

$$\begin{aligned}
&\forall \big(a^{:A\leftrightarrow A'}, b^{:B\leftrightarrow B'}, c^{:C\leftrightarrow C'}, d^{:D\leftrightarrow D'}, g^{:B\to A}, h^{:B'\to A'}, k^{:C\to D}, l^{:C'\to D'}, x^{:P^{A,C}}, y^{:P^{A',C'}}\big) \quad : \\
&\quad \text{if} \quad (g,h) \in b \boxtimes a \quad , \quad (k,l) \in c \boxtimes d \quad , \quad \text{and} \quad (x,y) \in (a,c)^{\updownarrow P} \quad , \\
&\quad \text{then} \quad \big(\text{xmap}_P(g)(k)(x), \text{xmap}_P(h)(l)(y)\big) \in (b,d)^{\updownarrow P} \quad .
\end{aligned} \tag{D.24}$$

The conclusion of Eq. (D.23) is that two `xmap(...)` values are in the relation $\langle f\rangle^{\updownarrow T}$. We can reproduce that conclusion using Eq. (D.24) if we set $b = d = \langle f\rangle$ since, by Statement D.3.3.3, we will then have $(b,d)^{\updownarrow P} = (\langle f\rangle, \langle f\rangle)^{\updownarrow P} = \langle f\rangle^{\updownarrow T}$. The corresponding types need to be set as $B = D = X$ and $B' = D' = Y$.

Next, we need to find the values g, h, k, l, x, and y such that the `xmap(...)` expressions in Eq. (D.24) reproduce those in Eq. (D.23):

$$\text{xmap}_P(g)(k)(x) = \text{xmap}_P(f)(\text{id})(z) \quad , \qquad \text{xmap}_P(h)(l)(y) = \text{xmap}_P(\text{id})(f)(z) \quad .$$

It is clear that we need to set $g = l = f$, $h = \text{id}$, $k = \text{id}$, and $x = y = z$. This also implies that we set the types as $B = C' = X$, $A = D' = Y$, $A' = B' = Y$, and $C = D = X$.

Since we have $x = y$, the precondition $(x,y) \in (a,c)^{\updownarrow P}$ will hold if $(a,c)^{\updownarrow P}$ is an identity relation. To achieve that, we choose $a = \text{id}$ and $c = \text{id}$, which also means setting $A = A' = Y$ and $C = C' = X$.

Finally, we can check that the preconditions $(g,h) \in b \boxtimes a$ and $(k,l) \in c \boxtimes d$ in Eq. (D.24) will then hold automatically. So, we may use the conclusion of Eq. (D.24), which proves Eq. (D.23). □

It turns out that for some (but not all) profunctors P, the (P,f)-wedge relation is *equivalent* to the (P,f)-pushout relation:

Definition D.3.6.2 A profunctor $P^{X,Y}$ has the **pushout property** if the relation $\text{push}\,(f^{\downarrow P}, f^{\uparrow P})$ is a

$$\begin{tikzcd} & P^{A,A} \arrow[d, "f^{\uparrow P^{A,\bullet}}"'] \\ P^{B,B} \arrow[r, "f^{\downarrow P^{\bullet,B}}"] & P^{A,B} \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} P^{B,A} \arrow[r, "f^{\downarrow P^{\bullet,A}}"] \arrow[d, "f^{\uparrow P^{B,\bullet}}"'] & P^{A,A} \arrow[d, "f^{\uparrow P^{A,\bullet}}"] \\ P^{B,B} \arrow[r, "f^{\downarrow P^{\bullet,B}}"] & P^{A,B} \end{tikzcd}$$

consequence of pull $(f^{\uparrow P}, f^{\downarrow P})$. So, for any $x^{:P^{A,A}}$, $y^{:P^{B,B}}$:

$$\text{if } (x, y) \in \text{pull}\,(f^{\uparrow P}, f^{\downarrow P}) \text{ then } (x, y) \in \text{push}\,(f^{\downarrow P}, f^{\uparrow P}) \quad .$$

In other words, for any $f^{:A\to B}$ and any values $x^{:P^{A,A}}$ and $y^{:P^{B,B}}$ in the (P, f)-wedge relation $(x \triangleright f^{\uparrow P^{A,\bullet}} = y \triangleright f^{\downarrow P^{\bullet,B}})$, we can compute a value $p^{:P^{B,A}}$ such that $x = p \triangleright f^{\downarrow P^{\bullet,A}}$ and $y = p \triangleright f^{\uparrow P^{B,\bullet}}$. The type diagram means that a wedge starting with arbitrary $x^{:P^{A,A}}$ and $y^{:P^{B,B}}$ can be always completed to a square that represents the profunctor commutativity law (D.7). □

Below we will perform structural analysis for profunctors P with a pushout property and give some examples. The significance of those P is that we can obtain a property that is stronger than ordinary dinaturality:

Definition D.3.6.3 A function $t^A : P^{A,A} \to Q^{A,A}$ is **strongly dinatural** if for any $f^{:A\to B}$, $x^{:P^{A,A}}$, and $y^{:P^{B,B}}$ the following property holds:

$$\text{when} \quad x \triangleright f^{\uparrow P^{A,\bullet}} = y \triangleright f^{\downarrow P^{\bullet,B}} \quad \text{then} \quad x \triangleright t^A \triangleright f^{\uparrow Q^{A,\bullet}} = y \triangleright t^B \triangleright f^{\downarrow Q^{\bullet,B}} \quad . \tag{D.25}$$

$$\begin{tikzcd} P^{A,A} \arrow[rr, "t^A"] \arrow[dr, "f^{\uparrow P^{A,\bullet}}"'] & & Q^{A,A} \arrow[dr, "f^{\uparrow Q^{A,\bullet}}"] & \\ & P^{A,B} & \Rightarrow & Q^{A,B} \\ P^{B,B} \arrow[rr, "t^B"] \arrow[ur, "f^{\downarrow P^{\bullet,B}}"] & & Q^{B,B} \arrow[ur, "f^{\downarrow Q^{\bullet,B}}"'] & \end{tikzcd}$$

The strong dinaturality law is an equation (the "conclusion") that is required to hold only when some values satisfy another equation (the law's "precondition"). The type diagram for that law, shown at left, is a truncated form of the type diagram for Eq. (D.4). The diagram starts with two arbitrary values of types $P^{A,A}$ and $P^{B,B}$. The implication symbol ($\Rightarrow$) here means that the left part of the diagram is an assumption used by the right part.

In terms of the wedge relations, strong dinaturality's "precondition" is the (P, f)-wedge relation for (x, y), and the "conclusion" is the (Q, f)-wedge relation for $(t^A(x), t^B(y))$. So, the strong dinaturality condition (D.25) can be expressed as a relation using the pair mapper ($\boxtimes$):

$$(t^A, t^B) \in \text{pull}(f^{\uparrow P^{A,\bullet}}, f^{\downarrow P^{\bullet,B}}) \boxtimes \text{pull}(f^{\uparrow Q^{A,\bullet}}, f^{\downarrow Q^{\bullet,B}}) \quad .$$

For comparison, the ordinary dinaturality property of t is written in a similar way as:

$$(t^A, t^B) \in \text{push}\,(f^{\downarrow P^{\bullet,A}}, f^{\uparrow P^{B,\bullet}}) \boxtimes \text{pull}(f^{\uparrow Q^{A,\bullet}}, f^{\downarrow Q^{\bullet,B}}) \quad .$$

Statement D.3.6.4 If a profunctor P has the pushout property and Q is any profunctor then any fully parametric function $t : \forall A.\, P^{A,A} \to Q^{A,A}$ is strongly dinatural.

Proof By Statements D.1.1.3 and D.3.5.1, the function t satisfies the ordinary dinaturality law:

$$\text{if} \quad (x^{:P^{A,A}}, y^{:P^{B,B}}) \in \text{push}\,(f^{\downarrow P}, f^{\uparrow P}) \quad \text{then} \quad (t^A(x), t^B(y)) \in \text{pull}\,(f^{\uparrow Q}, f^{\downarrow Q}) \quad .$$

The pushout property of P means:

$$\text{if} \quad (x^{:P^{A,A}}, y^{:P^{B,B}}) \in \text{pull}\,(f^{\uparrow P}, f^{\downarrow P}) \quad \text{then} \quad (x, y) \in \text{push}\,(f^{\downarrow P}, f^{\uparrow P}) \quad .$$

Chaining the implications, we find:

$$\text{if} \quad (x^{:P^{A,A}}, y^{:P^{B,B}}) \in \text{pull}\,(f^{\uparrow P}, f^{\downarrow P}) \quad \text{then} \quad (t^A(x), t^B(y)) \in \text{pull}\,(f^{\uparrow Q}, f^{\downarrow Q}) \quad .$$

This is the strong dinaturality law of t. □

The name "strong dinaturality" suggests that this property is stronger than the ordinary dinaturality. Indeed, strongly dinatural transformations are always dinatural:

Statement D.3.6.5 **(a)** Any function $t^A : P^{A,A} \to Q^{A,A}$ satisfying Eq. (D.25) will also satisfy Eq. (D.4).

(b) A natural transformation $t^A : F^A \to G^A$ (where F and G are both functors or both contrafunctors) is strongly dinatural.

(c) A natural transformation $t^{X,Y} : P^{X,Y} \to Q^{X,Y}$ between profunctors P and Q gives a strongly dinatural transformation $t^{A,A} : P^{A,A} \to Q^{A,A}$ if we set $X \stackrel{\text{def}}{=} A$ and $Y \stackrel{\text{def}}{=} A$ in $t^{X,Y}$.

Proof **(a)** The law (D.4) is an equality of functions of type $P^{B,A} \to Q^{A,B}$. We will now show that those functions will give equal results when applied to an arbitrary value $p : P^{B,A}$. Choose $x \stackrel{\text{def}}{=} p \triangleright f^{\downarrow P^{\bullet,A}}$ and $y \stackrel{\text{def}}{=} p \triangleright f^{\uparrow P^{B,\bullet}}$. The precondition in Eq. (D.25) is satisfied with these x and y:

$$\begin{aligned} \text{expect to equal } (y \triangleright f^{\downarrow P^{\bullet,B}}) : \quad & \underline{x} \triangleright f^{\uparrow P^{A,\bullet}} = p \triangleright \underline{f^{\downarrow P^{\bullet,A}} \triangleright f^{\uparrow P^{A,\bullet}}} \\ \text{profunctor commutativity law of } P : \quad & = p \triangleright f^{\uparrow P^{B,\bullet}} \triangleright f^{\downarrow P^{\bullet,B}} = y \triangleright f^{\downarrow P^{\bullet,B}} \quad . \end{aligned}$$

So, we can use the law (D.25)'s conclusion and obtain Eq. (D.4) applied to p, completing the proof:

$$\begin{aligned} & \underline{x} \triangleright t^A \triangleright f^{\uparrow Q^{A,\bullet}} \stackrel{!}{=} \underline{y} \triangleright t^B \triangleright f^{\downarrow Q^{\bullet,B}} \quad , \\ \text{definitions of } x, y : \quad & p \triangleright f^{\downarrow P^{\bullet,A}} \fatsemi t^A \fatsemi f^{\uparrow Q^{A,\bullet}} \stackrel{!}{=} p \triangleright f^{\uparrow P^{B,\bullet}} \fatsemi t^B \fatsemi f^{\downarrow Q^{\bullet,B}} \quad . \end{aligned}$$

(b) Consider the case where F and G are both functors. We may view $t : F^A \to G^A$ as a function with the type signature $t : P^{A,A} \to Q^{A,A}$ if we define the profunctors $P^{X,Y} \stackrel{\text{def}}{=} F^Y$ and $Q^{X,Y} \stackrel{\text{def}}{=} G^Y$. Since $P^{X,Y}$ and $Q^{X,Y}$ ignore the type parameter X, we have the liftings $f^{\uparrow P} = f^{\uparrow F}$, $f^{\downarrow P} = \text{id}$, $f^{\uparrow Q} = f^{\uparrow G}$, and $f^{\downarrow Q} = \text{id}$. The strong dinaturality law of t is then written as:

$$\text{when} \quad x^{:F^A} \triangleright f^{\uparrow F} = y^{:F^B} \quad \text{then} \quad x \triangleright t \triangleright f^{\uparrow G} \stackrel{?}{=} y \triangleright t \quad .$$

This is equivalent to $x \triangleright t \fatsemi f^{\uparrow G} = x \triangleright f^{\uparrow F} \fatsemi t$, which holds by the assumed naturality of t.

A similar proof works when F and G are both contrafunctors.

We may also derive part **(b)** from part **(c)**.

(c) A natural transformation $t^{X,Y} : P^{X,Y} \to Q^{X,Y}$ satisfies naturality laws separately with respect to the type parameters X and Y. So, for all $f^{:A \to B}$ we have:

$$f^{\uparrow P^{X,\bullet}} \fatsemi t^{X,B} = t^{X,A} \fatsemi f^{\uparrow Q^{X,\bullet}} \quad , \qquad f^{\downarrow P^{\bullet,Y}} \fatsemi t^{A,Y} = t^{B,Y} \fatsemi f^{\downarrow Q^{\bullet,Y}} \quad .$$

We need to verify the strong dinaturality law of $t^{A,A}$:

$$\text{when} \quad x^{:P^{A,A}} \triangleright f^{\uparrow P^{A,\bullet}} = y^{:P^{B,B}} \triangleright f^{\downarrow P^{\bullet,B}} \quad \text{then} \quad x \triangleright t^{A,A} \fatsemi f^{\uparrow Q^{A,\bullet}} \stackrel{?}{=} y \triangleright t^{B,B} \fatsemi f^{\downarrow Q^{\bullet,B}} \quad .$$

Using the precondition of that law and the naturality laws of $t^{X,Y}$, rewrite the conclusion of that law:

$$\begin{aligned} & x \triangleright \underline{t^{A,A} \fatsemi f^{\uparrow Q^{A,\bullet}}} = \underline{x \triangleright f^{\uparrow P^{A,\bullet}}} \fatsemi t^{A,B} = y \triangleright f^{\downarrow P^{\bullet,B}} \fatsemi t^{A,B} \\ & \stackrel{?}{=} y \triangleright \underline{t^{B,B} \fatsemi f^{\downarrow Q^{\bullet,B}}} = y \triangleright f^{\downarrow P^{\bullet,B}} \fatsemi t^{A,B} \quad . \end{aligned}$$

The two sides of the strong dinaturality law are now equal. □

Let us look at some examples of profunctors to see which ones have the pushout property.

Example D.3.6.6 **(a)** Suppose a profunctor $P^{X,Y}$ does not depend on the type parameter X (so, $P^{X,Y} \stackrel{\text{def}}{=} G^Y$ with some functor G). Then P has the pushout property.

(b) The pushout property holds for profunctors $P^{X,Y} \stackrel{\text{def}}{=} H^X$, where H is a contrafunctor.

Proof **(a)** With the choice $P^{X,Y} \stackrel{\text{def}}{=} G^Y$, we have the liftings $f^{\uparrow P} = f^{\uparrow G}$ and $f^{\downarrow P} = f$. The pushout property of $P^{X,Y}$ says that, for any $f^{:A \to B}$, $x^{:G^A}$, $y^{:G^B}$:

$$\text{when} \quad x \triangleright f^{\uparrow G} = y \quad \text{then} \quad \exists p^{:G^A} \text{ such that } x = p \text{ and } y = p \triangleright f^{\uparrow G} \quad .$$

This property is satisfied by choosing $p \stackrel{\text{def}}{=} x$.

(b) With the choice $P^{X,Y} \stackrel{\text{def}}{=} H^X$, we have the liftings $f^{\uparrow P} = f$ and $f^{\downarrow P} = f^{\downarrow H}$. The pushout property of $P^{X,Y}$ says that, for any $f^{:A \to B}$, $x^{:H^A}$, $y^{:H^B}$:

$$\text{when} \quad x = y \triangleright f^{\downarrow H} \quad \text{then} \quad \exists p^{:H^A} \text{ such that } x = p \triangleright f^{\downarrow H} \text{ and } y = p \quad .$$

This property is satisfied by choosing $p \stackrel{\text{def}}{=} y$.

Example D.3.6.7 The profunctor $P^{X,Y} \stackrel{\text{def}}{=} X \to Y$ does *not* have the pushout property.

Proof With $P^{X,Y} \stackrel{\text{def}}{=} X \to Y$, we rewrite the (P, f)-wedge relation $(x \triangleright f^{\uparrow P^{A,\bullet}} = y \triangleright f^{\downarrow P^{\bullet,B}})$ for arbitrary $f^{:A\to B}$, $x^{:A\to A}$, and $y^{:B\to B}$ as:

$$x \fatsemi f = f \fatsemi y \quad .$$

The pushout property requires us to find a value $p^{:B\to A}$ such that $x = f \fatsemi p$ and $y = p \fatsemi f$. Choose f as a constant function ($f \stackrel{\text{def}}{=} _ \to b_0$ with a fixed value $b_0^{:B}$). The (P, f)-wedge relation applied to an arbitrary value $a^{:A}$ gives:

$$a \triangleright x \fatsemi f = b_0 \stackrel{!}{=} a \triangleright f \fatsemi y = y(b_0) \quad .$$

So, the wedge relation will hold for any $x^{:A\to A}$ and for any $y^{:B\to B}$ such that $y(b_0) = b_0$. In particular, y is not necessarily a constant function. But the condition $y = p \fatsemi f = _ \to b_0$ can be satisfied only if y is a constant function. So, there exists no suitable value $p^{:P^{B,A}}$. □

Structural analysis allows us to discover profunctors that have the pushout property:

Statement D.3.6.8 A profunctor $P^{X,Y}$ has the pushout property if:

(a) $P^{X,Y} \stackrel{\text{def}}{=} F^Y$ with some (covariant) functor F.
(b) $P^{X,Y} \stackrel{\text{def}}{=} G^X$ with some contrafunctor G.
(c) $P^{X,Y} \stackrel{\text{def}}{=} Z \to Q^{X,Y}$ with a fixed type Z and a profunctor Q that has the pushout property.
(d) $P^{X,Y} \stackrel{\text{def}}{=} Q^{X,Y} \times R^{X,Y}$ with profunctors Q and R that both have the pushout property.
(e) $P^{X,Y} \stackrel{\text{def}}{=} Q^{X,Y} + R^{X,Y}$ with profunctors Q and R that both have the pushout property.

Proof Consider some values $f^{:A\to B}$, $x^{:P^{A,A}}$, and $y^{:P^{B,B}}$, for which the (P, f)-wedge relation holds:

$$x \triangleright f^{\uparrow P^{A,\bullet}} = y \triangleright f^{\downarrow P^{\bullet,B}} \quad .$$

(a) If $P^{X,Y} = F^Y$ with a functor F, the wedge relation is simplified to $x \triangleright f^{\uparrow F} = y$ with $x^{:F^A}$ and $y^{:F^B}$. We need to find $p^{:F^A}$ such that $x = p$ and $y = x \triangleright f^{\uparrow F}$. So, we define $p \stackrel{\text{def}}{=} x$ and complete the commuting square.

(b) If $P^{X,Y} = G^X$ with a contrafunctor G, the wedge relation is simplified to $x = y \triangleright f^{\downarrow G}$ with $x^{:G^A}$ and $y^{:G^B}$. We need to find $p^{:G^B}$ such that $x = p \triangleright f^{\downarrow G}$ and $y = p$. So, we define $p \stackrel{\text{def}}{=} y$ and complete the commuting square.

(c) Write the (P, f)-wedge relation for $x^{:Z\to Q^{A,A}}$ and $y^{:Z\to Q^{B,B}}$, applied to an arbitrary $z^{:Z}$:

$$z \triangleright x \triangleright f^{\uparrow Q^{A,\bullet}} = z \triangleright y \triangleright f^{\downarrow Q^{\bullet,B}} \quad .$$

Note that this is the same as the (Q, f)-wedge relation for the values $z \triangleright x$ and $z \triangleright y$. By assumption, $Q^{X,Y}$ has the pushout property. So, we may apply that property to $z \triangleright x$ and $z \triangleright y$ and obtain a value $q^{:Q^{B,A}}$ that satisfies the following two equations:

$$q \triangleright f^{\uparrow Q^{B,\bullet}} = z \triangleright y \quad , \qquad q \triangleright f^{\downarrow Q^{\bullet,A}} = z \triangleright x \quad . \tag{D.26}$$

We need to find $p^{:Z\to Q^{B,A}}$ such that $x = p \fatsemi f^{\downarrow Q^{\bullet,A}}$ and $y = p \fatsemi f^{\uparrow Q^{B,\bullet}}$. These equations are the same as Eq. (D.26) if we define $p(z) \stackrel{\text{def}}{=} q$. We have found a value p that completes the square diagram.

(d) Write the wedge relation for $x^{:Q^{A,A}\times R^{A,A}} = x_1^{:Q^{A,A}} \times x_2^{:R^{A,A}}$ and $y^{:Q^{B,B}\times R^{B,B}} = y_1^{:Q^{B,B}} \times y_2^{:R^{B,B}}$ as:

$$(x_1 \times x_2) \triangleright f^{\uparrow(Q\times R)} = (x_1 \triangleright f^{\uparrow Q}) \times (x_2 \triangleright f^{\uparrow R}) \stackrel{!}{=} (y_1 \times y_2) \triangleright f^{\downarrow(Q\times R)} = (y_1 \triangleright f^{\downarrow Q}) \times (y_2 \triangleright f^{\downarrow R}) \quad .$$

This equation is equivalent to the two equations:

$$x_1 \triangleright f^{\uparrow Q} \stackrel{!}{=} y_1 \triangleright f^{\downarrow Q} \quad \text{and} \quad x_2 \triangleright f^{\uparrow R} \stackrel{!}{=} y_2 \triangleright f^{\downarrow R} \quad ,$$

which are the (Q, f)- and (R, f)-wedge relations. By assumption, Q and R have the pushout property. So, we may compute values $p_1^{:Q^{B,A}}$ and $p_2^{:R^{B,A}}$ such that:

$$p_1 \triangleright f^{\downarrow Q} = x_1 \quad , \qquad p_1 \triangleright f^{\uparrow Q} = y_1 \quad , \qquad p_2 \triangleright f^{\downarrow R} = x_2 \quad , \qquad p_2 \triangleright f^{\uparrow R} = y_2 \quad .$$

If we define $p \stackrel{\text{def}}{=} p_1 \times p_2$, we complete the square diagram for $(Q \times R, f)$-wedge relation because:

$$p \triangleright f^{\downarrow(Q\times R)} = (p_1 \triangleright f^{\downarrow Q}) \times (p_2 \triangleright f^{\downarrow R}) = x_1 \times x_2 \quad , \qquad p \triangleright f^{\uparrow(Q\times R)} = (p_1 \triangleright f^{\uparrow Q}) \times (p_2 \triangleright f^{\uparrow R}) = y_1 \times y_2 \quad .$$

(e) Write the wedge relation for $x^{:Q^{A,A}+R^{A,A}}$ and $y^{:Q^{B,B}+R^{B,B}}$ as:

$$x \triangleright \begin{array}{c|c||cc|} & & Q^{A,B} & R^{A,B} \\ \hline & Q^{A,A} & f^{\uparrow Q^{A,\bullet}} & \mathbb{0} \\ & R^{A,A} & \mathbb{0} & f^{\uparrow R^{A,\bullet}} \end{array} = y \triangleright \begin{array}{c|c||cc|} & & Q^{A,B} & R^{A,B} \\ \hline & Q^{B,B} & f^{\downarrow Q^{\bullet,B}} & \mathbb{0} \\ & R^{B,B} & \mathbb{0} & f^{\downarrow R^{\bullet,B}} \end{array} \quad .$$

This equation can be satisfied only if both sides are either of type $Q^{A,B} + \mathbb{0}$ or of type $\mathbb{0} + R^{A,B}$. Since the lifting matrices are diagonal, this can happen only if x has type $Q^{A,A} + \mathbb{0}$ and y has type $Q^{B,B} + \mathbb{0}$, of if x has type $\mathbb{0} + R^{A,A}$ and y has type $\mathbb{0} + R^{B,B}$. In the first case, the property becomes equivalent to that of the profunctor Q; in the second case, to that of the profunctor R.

To see this in detail, consider two cases: $x \stackrel{\text{def}}{=} x_1^{:Q^{A,A}} + \mathbb{0}$ and $x \stackrel{\text{def}}{=} \mathbb{0} + x_2^{:R^{A,A}}$. In the first case, $x \triangleright f^{\uparrow P} = (x_1 \triangleright f^{\uparrow Q}) + \mathbb{0}$. This can be equal to $y \triangleright f^{\downarrow P}$ only if $y \triangleright f^{\downarrow P}$ is in the left part of the disjunction type $Q^{A,B} + R^{A,B}$. This happens only when $y = y_1^{:Q^{B,B}} + \mathbb{0}$ with some y_1. So, the (P, f)-wedge relation implies $x_1 \triangleright f^{\uparrow Q} = y_1 \triangleright f^{\downarrow Q}$. This is the (Q, f)-wedge relation for (x_1, y_1). By assumption, Q has the pushout property. So, we can compute some $p_1^{:Q^{B,A}}$ such that $x_1 = p_1 \triangleright f^{\downarrow Q}$ and $y_1 = p_1 \triangleright f^{\uparrow Q}$. If we now define $p \stackrel{\text{def}}{=} p_1 + \mathbb{0}$, we will have $p \triangleright f^{\downarrow P} = x$ and $p \triangleright f^{\uparrow P} = y$, so the pushout property of P holds.

The case $x \stackrel{\text{def}}{=} \mathbb{0} + x_2^{:R^{A,A}}$ is proved similarly by using the pushout property of R. □

The list of constructions in Statement D.3.6.8 does not include the recursive type construction. It remains an open question whether, say, a suitably limited form of the recursive type construction would produce new profunctors having the pushout property.

However, profunctors P with the pushout property are not the only ones that produce strongly dinatural transformations. This is because the pushout property is too restrictive. It is sufficient if the lifted relation $\langle f \rangle^{\updownarrow P}$ is a *consequence* of the (P, f)-wedge relation. We call this the "post-wedge" property of the profunctor P.

Here and below, we will frequently need to use liftings $r^{\updownarrow T}$ to a type constructor $T^\bullet$ defined as $T^A \stackrel{\text{def}}{=} P^{A,A}$ with some profunctor $P^{X,Y}$. Statement D.3.3.3 shows that $r^{\updownarrow T} = (r, r)^{\updownarrow P}$. So, we will write the simultaneous lifting $(r, r)^{\updownarrow P}$ simply as $r^{\updownarrow P}$ when this does not cause confusion.

Definition D.3.6.9 A profunctor P has the **post-wedge property** if for any $f^{:A\to B}$, $x^{:P^{A,A}}$, and $y^{:P^{B,B}}$:

$$\text{if} \quad x \triangleright f^{\uparrow P} = x \triangleright f^{\downarrow P} \quad \text{then} \quad (x, y) \in \langle f \rangle^{\updownarrow P} \quad .$$

In other words, the lifted relation $\langle f \rangle^{\updownarrow P}$ always follows from the (P, f)-wedge relation. □

By Statement D.3.5.1(a), the (P, f)-wedge relation is always a consequence of the lifted relation $\langle f \rangle^{\updownarrow P}$. So, the post-wedge property of P means that $\langle f \rangle^{\updownarrow P}$ is *equivalent* to the (P, f)-wedge relation.

Statement D.3.6.10 If P is a profunctor with the post-wedge property and Q is any profunctor then any fully parametric function $t : \forall A. P^{A,A} \to Q^{A,A}$ is strongly dinatural.

Proof The post-wedge property of P means:

$$\text{if} \quad (x^{:P^{A,A}}, y^{:P^{B,B}}) \in \text{pull}\,(f^{\uparrow P}, f^{\downarrow P}) \quad \text{then} \quad (x, y) \in \langle f \rangle^{\updownarrow P} \quad .$$

Since t is fully parametric, it satisfies the relational naturality law:

$$\text{if} \quad (x, y) \in \langle f \rangle^{\updownarrow P} \quad \text{then} \quad (t^A(x), t^B(y)) \in \langle f \rangle^{\updownarrow Q} \quad .$$

By Statement D.3.5.1(a), the wedge law follows from the relational naturality law of t:

$$\text{if} \quad (t^A(x), t^B(y)) \in \langle f \rangle^{\updownarrow Q} \quad \text{then} \quad (t^A(x), t^B(y)) \in \text{pull}\,(f^{\uparrow Q}, f^{\downarrow Q}) \quad .$$

Chaining the implications, we find:

$$\text{if} \quad (x^{:P^{A,A}}, y^{:P^{B,B}}) \in \text{pull}\,(f^{\uparrow P}, f^{\downarrow P}) \quad \text{then} \quad (t^A(x), t^B(y)) \in \text{pull}\,(f^{\uparrow Q}, f^{\downarrow Q}) \quad .$$

This is the strong dinaturality law of t. □

We will now do structural analysis to describe the profunctors with the post-wedge property.[6]

Statement D.3.6.11 A profunctor P will have the post-wedge property if:

(a) The type expression $P^{X,Y}$ does not depend either on X or on Y. That is, either $P^{X,Y} \stackrel{\text{def}}{=} Q^Y$ where Q is a functor or $P^{X,Y} \stackrel{\text{def}}{=} R^X$ where R is a contrafunctor.

(b) We have $P^{X,Y} \stackrel{\text{def}}{=} K^{X,Y} \times L^{X,Y}$, where the profunctors K and L have the post-wedge property.

(c) We have $P^{X,Y} \stackrel{\text{def}}{=} K^{X,Y} + L^{X,Y}$, where the profunctors K and L have the post-wedge property.

(d) We have $P^{X,Y} \stackrel{\text{def}}{=} K^{Y,X} \to L^{X,Y}$, where the profunctor K has the pushout property and the profunctor L has the post-wedge property.

Proof In each case, assuming the (P, f)-wedge relation for (x, y), we will show that $(x, y) \in \langle f \rangle^{\updownarrow P}$.

(a) If $P^{X,Y}$ does not depend on one of its type parameters then we have either $P^{X,Y} \stackrel{\text{def}}{=} Q^Y$ or $P^{X,Y} \stackrel{\text{def}}{=} R^X$. We know from Statement D.3.6.8(a, b) that P will then have the pushout property. So, if any values x and y are in the (P, f)-wedge relation then $(x, y) \in \text{push}\,(f^{\downarrow P}, f^{\uparrow P})$. By Statement D.3.6.1, we will also have $(x, y) \in \langle f \rangle^{\uparrow P}$.

(b) We need to show that P has the post-wedge property:

$$\text{if} \quad (k_1^{:K^{A,A}} \times l_1^{:L^{A,A}}) \triangleright f^{\uparrow P} = (k_2^{:K^{B,B}} \times l_2^{:L^{B,B}}) \triangleright f^{\downarrow P} \quad \text{then} \quad (k_1 \times l_1, k_2 \times l_2) \in \langle f \rangle^{\updownarrow P} \quad .$$

Using the definitions of the liftings $^{\updownarrow P}$, $^{\uparrow P}$, and $^{\downarrow P}$, we rewrite the above condition as:

$$\text{if} \quad k_1 \triangleright f^{\uparrow K} = k_2 \triangleright f^{\downarrow K} \text{ and } l_1 \triangleright f^{\uparrow L} = l_2 \triangleright f^{\downarrow L} \quad \text{then} \quad (k_1, k_2) \in \langle f \rangle^{\updownarrow K} \text{ and } (l_1, l_2) \in \langle f \rangle^{\updownarrow L} \quad .$$

This is the same as the conjunction of the post-wedge properties of K and L.

(c) We need to show that P has the post-wedge property:

$$\text{if} \quad p_1^{:K^{A,A}+L^{A,A}} \triangleright f^{\uparrow P} = p_2^{:K^{B,B}+L^{B,B}} \triangleright f^{\downarrow P} \quad \text{then} \quad (p_1, p_2) \in \langle f \rangle^{\updownarrow P} \quad .$$

The liftings $f^{\uparrow P}$ and $f^{\downarrow P}$ are defined via the standard pattern-matching code for disjunctive types:

$$f^{\uparrow P^{A,\bullet}} \stackrel{\text{def}}{=} \begin{array}{|c||cc|} & K^{A,B} & L^{A,B} \\ \hline K^{A,A} & f^{\uparrow K^{A,\bullet}} & \mathbb{0} \\ L^{A,A} & \mathbb{0} & f^{\uparrow L^{A,\bullet}} \end{array} \quad , \qquad f^{\downarrow P^{\bullet,B}} \stackrel{\text{def}}{=} \begin{array}{|c||cc|} & K^{A,B} & L^{A,B} \\ \hline K^{B,B} & f^{\downarrow K^{\bullet,B}} & \mathbb{0} \\ L^{B,B} & \mathbb{0} & f^{\downarrow L^{\bullet,B}} \end{array} \quad .$$

Since both code matrices are diagonal, the two parts of the disjunctive type $K + L$ do not mix. It follows that $p_1 \triangleright f^{\uparrow P} = p_2 \triangleright f^{\downarrow P}$ only if both p_1 and p_2 are in the same part of the disjunction $K + L$. If both p_1 and p_2 are in the left part of the disjunction, we have $p_1 \stackrel{\text{def}}{=} k_1^{:K^{A,A}} + \mathbb{0}$ and $p_2 \stackrel{\text{def}}{=} k_2^{:K^{B,B}} + \mathbb{0}$, and the (P, f)-wedge relation for p_1 and p_2 reduces to the (K, f)-wedge relation for k_1 and k_2:

$$k_1 \triangleright f^{\uparrow K} = k_2 \triangleright f^{\downarrow K} \quad .$$

By Definition D.3.2.5(d), the values p_1 and p_2 will be in the relation $\langle f \rangle^{\updownarrow P}$ only if $(k_1, k_2) \in \langle f \rangle^{\updownarrow K}$. Then the post-wedge property of P becomes:

$$\text{if} \quad k_1 \triangleright f^{\uparrow K} = k_2 \triangleright f^{\downarrow K} \quad \text{then} \quad (k_1, k_2) \in \langle f \rangle^{\updownarrow K} \quad .$$

This holds because it is just the post-wedge property of K. Similarly, we prove that if both p_1 and p_2 are in the right part of the disjunction then the post-wedge property of P is reduced to the post-wedge property of L.

[6] The following derivations are based on the talk slides: https://www.ioc.ee/~tarmo/tday-voore/vene-slides.pdf

(d) Express the (P, f)-wedge relation and the relation $\langle f\rangle^{\updownarrow P}$ for (x, y) through liftings to K and L:

$$x{:}K^{A,A}\to L^{A,A} \triangleright f^{\uparrow P^{A,\bullet}} = y{:}K^{B,B}\to L^{B,B} \triangleright f^{\downarrow P^{\bullet,B}} \quad \text{or equivalently :} \quad f^{\downarrow K^{\bullet,A}} \fatsemi x \fatsemi f^{\uparrow L^{A,\bullet}} = f^{\uparrow K^{B,\bullet}} \fatsemi y \fatsemi f^{\downarrow L^{\bullet,B}} \quad , \tag{D.27}$$

$$(x{:}K^{A,A}\to L^{A,A}, y{:}K^{B,B}\to L^{B,B}) \in \langle f\rangle^{\updownarrow P} \text{ means if } (k_1{:}K^{A,A}, k_2{:}K^{B,B}) \in \langle f\rangle^{\updownarrow K} \text{ then } (x(k_1), y(k_2)) \in \langle f\rangle^{\updownarrow L} \quad .$$

Fix any $k_1{:}K^{A,A}$ and $k_2{:}K^{B,B}$ such that $(k_1, k_2) \in \langle f\rangle^{\updownarrow K}$. We need to prove that $(x(k_1), y(k_2)) \in \langle f\rangle^{\updownarrow L}$ assuming Eq. (D.27).

The pushout property of K gives:

$$\text{if} \quad k_1{:}K^{A,A} \triangleright f^{\uparrow K} = k_2{:}K^{B,B} \triangleright f^{\downarrow K} \quad \text{then} \quad \exists k_0{:}K^{B,A} \text{ such that } k_1 = k_0 \triangleright f^{\downarrow K^{\bullet,A}} \text{ and } k_2 = k_0 \triangleright f^{\uparrow K^{B,\bullet}} \quad .$$

So, there exists a suitable value $k_0{:}K^{B,A}$. Apply both sides of Eq. (D.27) to that k_0:

$$k_0 \triangleright f^{\downarrow K^{\bullet,A}} \fatsemi x \fatsemi f^{\uparrow L^{A,\bullet}} = k_0 \triangleright f^{\uparrow K^{B,\bullet}} \fatsemi y \fatsemi f^{\downarrow L^{\bullet,B}} \quad \text{or equivalently :} \quad k_1 \triangleright x \triangleright f^{\uparrow L} = k_2 \triangleright y \triangleright f^{\downarrow L} \quad .$$

It is also given that L has the post-wedge property:

$$\text{if} \quad l_1{:}L^{A,A} \triangleright f^{\uparrow L} = l_2{:}L^{B,B} \triangleright f^{\downarrow L} \quad \text{then} \quad (l_1, l_2) \in \langle f\rangle^{\updownarrow L} \quad .$$

We now set $l_1 \stackrel{\text{def}}{=} k_1 \triangleright x$ and $l_2 \stackrel{\text{def}}{=} k_2 \triangleright y$ to obtain $(x(k_1), y(k_2)) \in \langle f\rangle^{\updownarrow L}$ as required. □

As a consequence of Statement D.3.6.11, we can quickly prove (without structural analysis) that a function relation lifted to functors or contrafunctors will yield another function relation:

Statement D.3.6.12 Lifting a function $f{:}A\to B$ to a fully parametric functor or a contrafunctor G agrees with lifting the relation $\langle f\rangle$ to G. In detail: **(a)** $\langle f\rangle^{\updownarrow G} = \langle f^{\uparrow G}\rangle$ if G is a functor. **(b)** $\langle f\rangle^{\updownarrow G} = \text{rev}\langle f^{\downarrow G}\rangle$ if G is a contrafunctor.

Proof Whether G is a functor or a contrafunctor, Statement D.3.6.11(a) shows that it has the post-wedge property. So, the relation $(\langle f\rangle, \langle f\rangle)^{\updownarrow P}$ is *equivalent* to the (P, f)-wedge relation:

$$(x, y) \in (\langle f\rangle, \langle f\rangle)^{\updownarrow P} \quad \text{is equivalent to :} \quad x \triangleright f^{\uparrow P} = y \triangleright f^{\downarrow P} \quad .$$

(a) If G is a functor, define $P^{X,Y} \stackrel{\text{def}}{=} G^Y$. Simplifying $(\langle f\rangle, \langle f\rangle)^{\updownarrow P} = \langle f\rangle^{\updownarrow G}$ and using the liftings $f^{\uparrow P} = f^{\uparrow G}$ and $f^{\downarrow P} = \text{id}$, we get:

$$(x, y) \in \langle f\rangle^{\updownarrow G} \quad \text{is equivalent to :} \quad x \triangleright f^{\uparrow G} = y \quad \text{or equivalently :} \quad (x, y) \in \langle f^{\uparrow G}\rangle \quad .$$

So, we find $\langle f\rangle^{\updownarrow G} = \langle f^{\uparrow G}\rangle$ as required.

(b) If G is a contrafunctor, define $P^{X,Y} \stackrel{\text{def}}{=} G^X$. Simplifying $(\langle f\rangle, \langle f\rangle)^{\updownarrow P} = \langle f\rangle^{\updownarrow G}$ and using the liftings $f^{\uparrow P} = \text{id}$ and $f^{\downarrow P} = f^{\downarrow G}$, we get:

$$(x, y) \in \langle f\rangle^{\updownarrow G} \quad \text{is equivalent to :} \quad x = y \triangleright f^{\downarrow G} \quad \text{or equivalently :} \quad (x, y) \in \text{rev}\langle f^{\downarrow G}\rangle \quad .$$

So, we find $\langle f\rangle^{\updownarrow G} = \text{rev}\langle f^{\downarrow G}\rangle$ as required. □

Strong dinaturality holds only for type signatures $\forall A.\, K^{A,A} \to L^{A,A}$ where the profunctor K has a certain structure. Nevertheless, a broad range of practically encountered functions have type signatures of this form. Here are some examples.

Example D.3.6.13 Use Statement D.3.6.10 to show that strong dinaturality holds for all fully parametric functions with the following type signatures:

(a) $\forall A.\, F^A \to H^A$.

(b) $\forall A.\, F^A \to G^A \to H^A$.

(c) $\forall A.\, (F^A \to G^A) \to H^A$.

Here each of $F^\bullet$, $G^\bullet$, $H^\bullet$ is independently either a functor or a contrafunctor.

Solution **(a)** and **(b)** We can represent these type signatures as $\forall A.\, K^{A,A} \to L^{A,A}$ with some profunctors $K^{X,Y}$ and $L^{X,Y}$. Then we will need to consider two cases where F is either covariant or contravariant. If F is covariant, we define $K^{X,Y} \stackrel{\text{def}}{=} F^Y$. If F is contravariant, we define $K^{X,Y} \stackrel{\text{def}}{=} F^X$. In every case, K depends on only one of its type parameters. We define $L^{X,Y}$ appropriately for each case. By Statement D.3.6.11(a), K will have the post-wedge property. Strong dinaturality will then follow from Statement D.3.6.10.

(c) The type signature $\forall A.\, (F^A \to G^A) \to H^A$ needs to be expressed as $\forall A.\, K^{A,A} \to L^{A,A}$ with some profunctors K and L. We define $K^{X,Y} \stackrel{\text{def}}{=} M^{Y,X} \to N^{X,Y}$ with suitably chosen profunctors M and N. Reasoning as before, we find that the profunctors M and N will depend only on one of their type parameters. So, they will have the post-wedge property. In addition, Example D.3.6.6(a) shows that M has the pushout property. The post-wedge property of K is then established via Statement D.3.6.11(d). So, the profunctor K satisfies the conditions of Statement D.3.6.10. □

Example D.3.6.13(c) is used in Statement 13.7.3.1 for the Church encoding of recursive types.

D.3.7 Strong dinaturality of `foldFn`

This section will show that the function `foldFn` used in Statement 12.3.1.2 is strongly dinatural. The type signature of `foldFn`,

$$\text{foldFn}_L : \forall B.\, L^{B\to B} \to B \to B \quad ,$$

is not immediately covered by Statement D.3.6.10 because it contains the composition of an arbitrary functor L and a profunctor. To handle `foldFn`, we will first need to prove some additional properties.

Statement D.3.7.1 If $L^\bullet$ is a polynomial functor and $K^{\bullet,\bullet}$ is a profunctor with the post-wedge property then the profunctor $P^{X,Y} \stackrel{\text{def}}{=} L^{K^{X,Y}}$ also has the post-wedge property.

Proof Assume any two values $x^{:L^{K^{A,A}}}$ and $y^{:L^{K^{B,B}}}$ that are in the (P, f)-wedge relation:

$$(x, y) \in \text{pull}\,(f^{\uparrow K^{A,\bullet}\uparrow L}, f^{\downarrow K^{\bullet,B}\uparrow L}) \quad \text{or equivalently}: \quad x \triangleright f^{\uparrow K^{A,\bullet}\uparrow L} = y \triangleright f^{\downarrow K^{\bullet,B}\uparrow L} \quad .$$

We need to show that $(x, y) \in \langle f\rangle^{\updownarrow P}$. As shown in Statement D.3.7.2 below, lifting a relation to P means first lifting to K and then to L, so $\langle f\rangle^{\updownarrow P} = \langle f\rangle^{\updownarrow K\updownarrow L}$. Since K has the post-wedge property, the (K, f)-wedge relation is equivalent to the relation $\langle f\rangle^{\updownarrow K}$:

$$(k_1, k_2) \in \langle f\rangle^{\updownarrow K} \quad \text{is equivalent to}: \quad (k_1, k_2) \in \text{pull}\,(f^{\uparrow K^{A,\bullet}}, f^{\downarrow K^{B,\bullet}}) \quad .$$

By Statement D.3.7.3 proved below, lifting a pullback relation to L gives again a pullback relation:

$$(x, y) \in \langle f\rangle^{\updownarrow P} = \big(\text{pull}\,(f^{\uparrow K^{A,\bullet}}, f^{\downarrow K^{B,\bullet}})\big)^{\updownarrow L} \quad \text{means} \quad (x, y) \in \text{pull}\,(f^{\uparrow K^{A,\bullet}\uparrow L}, f^{\downarrow K^{\bullet,B}\uparrow L}) \quad .$$

So, (x, y) are in the relation $\langle f\rangle^{\updownarrow P}$.

Statement D.3.7.2 For any type constructors $G^\bullet$ and $H^\bullet$, define $F \stackrel{\text{def}}{=} G \circ H$ or equivalently $F^A \stackrel{\text{def}}{=} G^{H^A}$. Then the lifting of any relation $r^{:A\leftrightarrow B}$ to $F^\bullet$ can be expressed as $r^{\updownarrow F} = (r^{\updownarrow H})^{\updownarrow G}$, which we may write more concisely as $r^{\updownarrow F} = r^{\updownarrow H\updownarrow G}$.

Proof For each case of Definition D.3.2.5 for the type constructor $G^\bullet$, we show that $r^{\updownarrow F} = (r^{\updownarrow H})^{\updownarrow G}$.

Constant type With $G^A \stackrel{\text{def}}{=} Z$ where Z is a fixed type, we have $F^A = Z$, so we write:

$$r^{\updownarrow F} = \text{id}^{:Z\leftrightarrow Z} \quad \text{and} \quad (r^{\updownarrow H})^{\updownarrow G} = \text{id}^{:Z\leftrightarrow Z} \quad .$$

Type parameter With $G^A \stackrel{\text{def}}{=} A$, we have $F^A = H^A$, so we write:

$$r^{\updownarrow F} = r^{\updownarrow H} \quad \text{and} \quad (r^{\updownarrow H})^{\updownarrow G} = (r^{\updownarrow H})^{\updownarrow \text{Id}} = r^{\updownarrow H} \quad .$$

Products With $G^A \stackrel{\text{def}}{=} K^A \times L^A$, we have $F^A = K^{H^A} \times L^{H^A}$. Assuming that Statement D.3.7.2 already holds for $K \circ H$ and $L \circ H$, we write:

$$r^{\updownarrow F} = r^{\updownarrow(K\circ H)} \boxtimes r^{\updownarrow(L\circ H)} = r^{\updownarrow H \updownarrow K} \boxtimes r^{\updownarrow H \updownarrow L} = (r^{\updownarrow H})^{\updownarrow K} \boxtimes (r^{\updownarrow H})^{\updownarrow L} \quad ,$$
$$(r^{\updownarrow H})^{\updownarrow G} = (r^{\updownarrow H})^{\updownarrow(K\times L)} = (r^{\updownarrow H})^{\updownarrow K} \boxtimes (r^{\updownarrow H})^{\updownarrow L} \quad .$$

The two relations are now equal.

Co-products and function types The proofs are similar to that for products if we replace the operation $\boxtimes$ by $\boxplus$ or by $\boxslash$ everywhere as appropriate.

Recursive types With $G^A \stackrel{\text{def}}{=} S^{A,G^A}$, we have $F^A = S^{H^A,G^{H^A}} = S^{H^A,F^A}$. Denoting $Q^{A,B} \stackrel{\text{def}}{=} S^{H^A,B}$, we can write $F^A = Q^{A,F^A}$. One inductive assumption is that Statement D.3.7.2 already holds separately with respect to each type parameter of $S^{\bullet,\bullet}$ and, in particular, a simultaneous lifting to Q satisfies:

$$(r,s)^{\updownarrow P} = (r,s)^{\updownarrow S^{H^\bullet,\bullet}} = (r^{\updownarrow H}, s)^{\updownarrow S^{\bullet,\bullet}} \quad .$$

Then we can finish the derivation:

$$\begin{aligned} \text{expect to equal } (r^{\updownarrow H})^{\updownarrow G} : \quad & r^{\updownarrow F} = (r, \overline{r^{\updownarrow F}})^{\updownarrow P} = (r^{\updownarrow H}, \overline{r^{\updownarrow F}})^{\updownarrow S} \\ \text{inductive assumption } \overline{r^{\updownarrow F}} = \overline{r^{\updownarrow H \updownarrow G}} : \quad & = (r^{\updownarrow H}, \overline{r^{\updownarrow H \updownarrow G}})^{\updownarrow S} = (r^{\updownarrow H})^{\updownarrow G} \quad . \end{aligned}$$

Quantified types With $G^A \stackrel{\text{def}}{=} \forall X. P^{X,A}$, we have $F^A = \forall X. P^{X,H^A}$. We may assume that Statement D.3.7.2 already holds separately for liftings with respect to each type parameter of $P^{\bullet,\bullet}$. In particular, if we denote $Q^{X,B} \stackrel{\text{def}}{=} P^{X,H^B}$ then:

$$(s,r)^{\updownarrow Q} = (s,r)^{\updownarrow P^{\bullet,H^\bullet}} = (s, r^{\updownarrow H})^{\updownarrow P} \quad .$$

For any types A, B, X, Y and for any values $p^{:F^A}$ and $q^{:F^B}$, we write:

$$\begin{aligned} (p,q) \in r^{\updownarrow F} \quad & \text{means} \quad \forall s^{:X\leftrightarrow Y} : (p^X, q^Y) \in (s,r)^{\updownarrow P^{\bullet,H^\bullet}} = (s, r^{\updownarrow H})^{\updownarrow P} \quad , \\ (p,q) \in (r^{\updownarrow H})^{\updownarrow G} \quad & \text{means} \quad \forall s^{:X\leftrightarrow Y} : (p^X, q^Y) \in (s, r^{\updownarrow H})^{\updownarrow P^{\bullet,\bullet}} \quad . \end{aligned}$$

The two relations are now equal.

Statement D.3.7.3 The lifting of a pullback relation to any polynomial functor F is equivalent to a pullback relation with lifted functions:

$$\text{for all } f^{:A\to C}, g^{:B\to C} \quad : \quad \big(\text{pull}\,(f,g)\big)^{\updownarrow F} = \text{pull}\,(f^{\uparrow F}, g^{\uparrow F}) \quad .$$

Proof We enumerate the type constructions that F is built from. Denote $r^{:A\leftrightarrow B} \stackrel{\text{def}}{=} \text{pull}\,(f,g)$.

Constant types If $F^A \stackrel{\text{def}}{=} Z$ where Z is a fixed type then $r^{\updownarrow F} = \text{id}^{:Z\leftrightarrow Z}$. We write:

$$\text{pull}\,(f^{\uparrow F}, g^{\uparrow F}) = \text{pull}\,(\text{id}^{:Z\leftrightarrow Z}, \text{id}^{:Z\leftrightarrow Z}) = \text{id}^{:Z\leftrightarrow Z} = r^{\updownarrow F} \quad .$$

Type parameter If $F^A \stackrel{\text{def}}{=} A$ then $r^{\updownarrow F} = r$, $f^{\uparrow F} = f$, and $g^{\uparrow F} = g$. We write:

$$\text{pull}\,(f^{\uparrow F}, g^{\uparrow F}) = \text{pull}\,(f,g) = r = r^{\updownarrow F} \quad .$$

Products If $F^A \stackrel{\text{def}}{=} K^A \times L^A$ then $r^{\updownarrow F} = r^{\updownarrow K} \boxtimes r^{\updownarrow L}$, $f^{\uparrow F} = f^{\uparrow K} \boxtimes f^{\uparrow L}$, and $g^{\uparrow F} = g^{\uparrow K} \boxtimes g^{\uparrow L}$. For any values $k_1^{:K^A}$, $l_1^{:L^A}$, $k_2^{:K^B}$, and $l_2^{:L^B}$, we write out the relation $r^{\updownarrow F}$:

$$(k_1 \times l_1, k_2 \times l_2) \in r^{\updownarrow F} = r^{\updownarrow K} \boxtimes r^{\updownarrow L} \quad \text{means} \quad (k_1, k_2) \in r^{\updownarrow K} \text{ and } (l_1, l_2) \in r^{\updownarrow L} \quad .$$

The inductive assumptions for K and L allow us to rewrite the last conditions as:

$$(k_1, k_2) \in \text{pull}\,(f^{\uparrow K}, g^{\uparrow K}) \quad \text{and} \quad (l_1, l_2) \in \text{pull}\,(f^{\uparrow L}, g^{\uparrow L}) \quad .$$

Turning now to the pullback relation, pull $(f^{\uparrow F}, g^{\uparrow F})$, we write:

$$\begin{aligned}
&(k_1^{:K^A} \times l_1^{:L^A}, k_2^{:K^B} \times l_2^{:L^B}) \in \text{pull}\,(f^{\uparrow F}, g^{\uparrow F}) = \text{pull}\,(f^{\uparrow K} \boxtimes f^{\uparrow L}, g^{\uparrow K} \boxtimes g^{\uparrow L}) \\
&\quad \text{means} \quad (k_1 \times l_1) \triangleright (f^{\uparrow K} \boxtimes f^{\uparrow L}) = (k_2 \times l_2) \triangleright (g^{\uparrow K} \boxtimes g^{\uparrow L}) \quad , \\
&\text{or equivalently :} \quad k_1 \triangleright f^{\uparrow K} = k_2 \triangleright g^{\uparrow K} \quad \text{and} \quad l_1 \triangleright f^{\uparrow L} = l_2 \triangleright g^{\uparrow L} \quad .
\end{aligned}$$

The last conditions are now the same as for the relation $r^{\updownarrow F}$.

Co-products If $F^A \stackrel{\text{def}}{=} K^A + L^A$ then $r^{\updownarrow F} = r^{\updownarrow K} \boxplus r^{\updownarrow L}$, $f^{\uparrow F} = f^{\uparrow K} \boxplus f^{\uparrow L}$, and $g^{\uparrow F} = g^{\uparrow K} \boxplus g^{\uparrow L}$. For any values $k_1^{:K^A}$, $l_1^{:L^A}$, $k_2^{:K^B}$, and $l_2^{:L^B}$, we write out the relation $r^{\updownarrow F}$:

$$\begin{aligned}
&\text{either :} \quad (k_1 + \mathbb{0}, k_2 + \mathbb{0}) \in r^{\updownarrow K} \boxplus r^{\updownarrow L} \quad \text{when} \quad (k_1, k_2) \in r^{\updownarrow K} \quad , \\
&\text{or :} \quad (\mathbb{0} + l_1, \mathbb{0} + l_2) \in r^{\updownarrow K} \boxplus r^{\updownarrow L} \quad \text{when} \quad (l_1, l_2) \in r^{\updownarrow L} \quad .
\end{aligned}$$

The inductive assumptions for K and L allow us to rewrite the last conditions as:

$$\begin{aligned}
&(k_1, k_2) \in r^{\updownarrow K} = \text{pull}\,(f^{\uparrow K}, g^{\uparrow K}) \quad \text{or equivalently :} \quad k_1 \triangleright f^{\uparrow K} = k_2 \triangleright g^{\uparrow K} \quad , \\
&(l_1, l_2) \in r^{\updownarrow L} = \text{pull}\,(f^{\uparrow L}, g^{\uparrow L}) \quad \text{or equivalently :} \quad l_1 \triangleright f^{\uparrow L} = l_2 \triangleright g^{\uparrow L} \quad .
\end{aligned}$$

Turning now to the pullback relation, pull $(f^{\uparrow F}, g^{\uparrow F})$, we write:

$$\begin{aligned}
&(p_1^{:K^A+L^A}, p_2^{:K^B+L^B}) \in \text{pull}\,(f^{\uparrow F}, g^{\uparrow F}) = \text{pull}\,(f^{\uparrow K} \boxplus f^{\uparrow L}, g^{\uparrow K} \boxplus g^{\uparrow L}) \\
&\quad \text{means} \quad p_1 \triangleright (f^{\uparrow K} \boxplus f^{\uparrow L}) = p_2 \triangleright (g^{\uparrow K} \boxplus g^{\uparrow L}) \quad .
\end{aligned} \tag{D.28}$$

The pair co-product (such as $f^{\uparrow K} \boxplus f^{\uparrow L}$) preserves the left and right parts of the disjunctive type, so the condition (D.28) is satisfied only when p_1 and p_2 are both in the same part of the disjunction:

$$\begin{aligned}
&p_1 \triangleright (f^{\uparrow K} \boxplus f^{\uparrow L}) = p_2 \triangleright (g^{\uparrow K} \boxplus g^{\uparrow L}) \quad \text{means} \quad : \\
&\text{either :} \quad p_1 = k_1 + \mathbb{0} \quad , \quad p_2 = k_2 + \mathbb{0} \quad , \quad k_1 \triangleright f^{\uparrow K} = k_2 \triangleright g^{\uparrow K} \quad ; \\
&\text{or :} \quad p_1 = \mathbb{0} + l_1 \quad , \quad p_2 = \mathbb{0} + l_2 \quad , \quad l_1 \triangleright f^{\uparrow L} = l_2 \triangleright g^{\uparrow L} \quad .
\end{aligned}$$

The last conditions are now the same as for the relation $r^{\updownarrow F}$.

Function types We may not use this construction since F is assumed to be a polynomial functor.

Recursive types If $F^A \stackrel{\text{def}}{=} S^{A,F^A}$ where $S^{\bullet,\bullet}$ is a polynomial bifunctor then:

$$r^{\updownarrow F} = \left(r, \overline{r^{\updownarrow F}}\right)^{\updownarrow S} \quad , \qquad f^{\uparrow F} = f^{\uparrow S^{\bullet,F^A}} \fatsemi \left(\overline{f^{\uparrow F}}\right)^{\uparrow S^{C,\bullet}} \quad , \qquad g^{\uparrow F} = g^{\uparrow S^{\bullet,F^B}} \fatsemi \left(\overline{g^{\uparrow F}}\right)^{\uparrow S^{C,\bullet}} \quad .$$

The inductive assumptions are that Statement D.3.7.3 already holds for the recursively used lifting $\overline{r^{\updownarrow F}}$ and for simultaneous liftings of *two* pullback relations to $S^{\bullet,\bullet}$. We can write these assumptions as:

$$\begin{aligned}
&\text{for } \overline{r^{\updownarrow F}} : \quad \overline{r^{\updownarrow F}} \stackrel{!}{=} \text{pull}\left(\overline{f^{\uparrow F}}, \overline{g^{\uparrow F}}\right) \quad , \\
&\text{for } S : \quad \left(\text{pull}\,(f^{:A\to C}, g^{:B\to C}),\ \text{pull}\,(h^{:X\to Z}, k^{:Y\to Z})\right)^{\updownarrow S} \stackrel{!}{=} \text{pull}\,(f^{\uparrow S^{\bullet,X}} \fatsemi h^{\uparrow S^{C,\bullet}},\ g^{\uparrow S^{\bullet,Y}} \fatsemi k^{\uparrow S^{C,\bullet}}) \quad ,
\end{aligned}$$

where the last pullback relation involves functions of types $S^{A,X} \to S^{C,Z}$ and $S^{B,Y} \to S^{C,Z}$. We write:

$$\begin{aligned}
&\text{expect to equal pull}\,(f^{\uparrow F}, g^{\uparrow F}) : \quad r^{\updownarrow F} = \left(r, \overline{r^{\updownarrow F}}\right)^{\updownarrow S} = \left(\text{pull}\,(f, g),\ \text{pull}\left(\overline{f^{\uparrow F}}, \overline{g^{\uparrow F}}\right)\right)^{\updownarrow S} \\
&\text{assumption for } S : \quad = \text{pull}\left(f^{\uparrow S^{\bullet,F^A}} \fatsemi \left(\overline{f^{\uparrow F}}\right)^{\uparrow S^{C,\bullet}}, g^{\uparrow S^{\bullet,F^B}} \fatsemi \left(\overline{g^{\uparrow F}}\right)^{\uparrow S^{C,\bullet}}\right) \\
&\text{definitions of } (...)^{\uparrow F} : \quad = \text{pull}\,(f^{\uparrow F}, g^{\uparrow F}) \quad .
\end{aligned}$$

□

The strong dinaturality property of `foldFn` is now proved as a special case of Statement D.3.7.1:

Example D.3.7.4 Show that any fully parametric function $f : \forall A. L^{A\to A} \to A \to A$ is strongly dinatural when L is any polynomial functor.

Solution Define the profunctor $K^{X,Y} \stackrel{\text{def}}{=} X \to Y$ so that the type signature of f is written as:

$$\forall A. L^{A\to A} \to A \to A = L^{K^{A,A}} \to K^{A,A} \quad .$$

The profunctor K has the post-wedge property by Statement D.3.6.11(a) and (d). Then Statement D.3.7.1 means that the profunctor $L^{K^{X,Y}}$ also has the post-wedge property. We can now use Statement D.3.7.1 to find that f satisfies the strong dinaturality law.

D.4 Summary and further directions

We have proved several properties that apply to all fully parametric code:

- The lifting methods of any fully parametric bifunctor, profunctor, or bi-contrafunctor obey the commutativity law such as Eq. (6.9).
- All fully parametric functions of type $P^{A,A} \to Q^{A,A}$ (where P, Q are profunctors) obey the dinaturality law (D.4). The form of the law depends only on the function's type signature and applies to all fully parametric implementations of that type signature.
- A given type constructor may have one fully parametric and lawful implementation of the `Functor` or `Contrafunctor` typeclass instance. (For most other typeclasses, such as `Filterable` or `Monad`, many type constructors have several inequivalent and lawful typeclass instances.) The unique implementations are defined by the type constructions from Sections 6.2.3 and 6.2.4.
- Parametricity theorem: Any fully parametric expression $t : \forall A. Q^A$ satisfies the relational naturality law (D.16). In general, the relational naturality law expresses a property of *relations* rather than functions and is not equivalent to any equation satisfied by t.
- If the type signature of t satisfies the conditions of Statements D.3.6.10 or D.3.7.1, the function t satisfies the "strong dinaturality" law (D.25), which gives more information than the dinaturality law but is simpler to use than the relational naturality law (D.16).

What remains unexplored:

- Relational databases give programmers many operations such as `SELECT` and `JOIN`. Similar operations can be defined for binary relations. For instance, two relations $r : A \leftrightarrow B$ and $s : B \leftrightarrow C$ can be composed using an operation similar to `JOIN`, which yields a new relation (denoted by $r \circ s$) of type $A \leftrightarrow C$. However, relational composition turns out to be unnecessary for proving the practically useful parametricity properties. At the same time, properties of relational composition are complicated and not intuitive. For instance, it is not true that $(r \circ s)^{\updownarrow F} = r^{\updownarrow F} \circ s^{\updownarrow F}$ for arbitrary type constructors F. It is also not possible to use the relational composition as a replacement for the simultaneous lifting construction (Definition D.3.3.2).
- Parametricity can be extended to types and code constructions beyond those covered in this book. The scope of material in this Appendix is intentionally limited to theory that has already proven its practical importance for programmers. For instance, we did not cover parametricity for functions with typeclass constraints. The corresponding theory is complicated[7] and yet delivers few practically useful results. For example, that theory does not seem to solve Problem 15.3.2.7, which considers functions with the following type signature:

```
def epsilon[M[_]: Monad, A]: M[A] => M[A]
```

[7] The theory of parametricity applied to functions with typeclass constraints is developed by J. Voigtländer in the paper https://www.janis-voigtlaender.eu/papers/FreeTheoremsInvolvingTypeConstructorClasses.pdf

If `epsilon` is a monad morphism that works in the same way for all monads `M` and all types `A`, can we prove that `epsilon` is an identity function? The analogous question involving functors can be answered by the parametricity theorem. Using the Yoneda lemma for functors (Statement 13.7.1.3), one can prove that any fully parametric function with the type signature:

```
def epsilon[F[_]: Functor, A]: F[A] => F[A]
```

must be an identity function.

E Solutions of some exercises

Chapter 1

Exercise 1.6.1.2

```
def at(n: Double, maxN: Int) = (0 to maxN)
  .map { k => 1.0*(1 - k % 2 * 2) / (2 * k + 1) / math.pow(n, 2 * k + 1) }
  .sum
def p(n: Int) = 16 * at(5, n) - 4 * at(239, n)

scala> p(12)
res0: Double = 3.141592653589794
```

Exercise 1.6.1.3

```
def isPrime(n: Int) = (2 to n - 1).takeWhile(k => k * k <= n).forall(k => n % k != 0)
def ep(n: Int): Double = (2 to n)
  .filter(isPrime)
  .map  { k => 1.0 / (1.0 - 1.0 / k / k / k /k) }
  .product
val pi = 3.1415926535897932

scala> ep(100); pi*pi*pi*pi/90
res0_0: Double = 1.0823231553280295
res0_1: Double = 1.082323233711138
```

Chapter 2

Exercise 2.1.7.10

```
numsLists.map(_.sortBy(- _).take(3))
```

Exercise 2.1.7.11

```
a.flatMap(x => b.map(y => (x, y)))
```

Exercise 2.1.7.12

```
def payments[Person, Amount](data: Seq[Map[Person, Amount]]): Map[Person, Seq[Amount]] =
  data.flatMap(_.toSeq).groupBy(_._1).mapValues(_.map(_._2))
```

Exercise 2.2.6.5

```
def batching[A](xs: Seq[A], size: Int): Seq[Seq[A]] = {
  type Acc = (Seq[Seq[A]], Seq[A], Int)
  val init: Acc = ((Seq(), Seq(), 0))
  val (result, rem, _) = xs.foldLeft(init){ case ((seq, rem, len), x) =>
    val newLen = len + 1
    if (newLen > size) (seq ++ Seq(rem), Seq(x), 1)
    else (seq, rem ++ Seq(x), newLen)
  }
  result ++ Seq(rem)
}
```

Exercise 2.2.6.6

```
def weightBatching[A](xs: Seq[A], maxW: Double)(w: A => Double): Seq[Seq[A]] = {
```

```
  type Acc = (Seq[Seq[A]], Seq[A], Double)
  val init: Acc = ((Seq(), Seq(), 0.0))
  val (result, rem, _) = xs.foldLeft(init) { case ((seq, rem, weight), x) =>
    val wx = w(x)
    if (wx > maxW) (seq ++ Seq(rem, Seq(x)), Seq(), 0.0)
    else {
      val newWeight = weight + wx
      if (newWeight > maxW) (seq ++ Seq(rem), Seq(x), wx)
      else (seq, rem ++ Seq(x), newWeight)
    }
  }
  result ++ Seq(rem)
}
```

Exercise 2.2.6.7

```
def groupBy[A, K](xs: Seq[A])(by: A => K): Map[K, Seq[A]] = {
  val init: Map[K, Seq[A]] = Map()
  xs.foldLeft(init) { (res, x) =>
    val key = by(x)
    val seq = res.getOrElse(key, Seq()) ++ Seq(x)
    res.updated(key, seq)
  }
}
```

Exercise 2.5.2.3

```
def digitsOf(n: Int): Seq[Int] = Stream.iterate(n)(_ / 10).takeWhile(_ != 0).map(_ % 10).toList
def cubeDigits(n: Int): Int = digitsOf(n).map(x => x*x*x).sum
def cubes(n: Int): Stream[Int] = Stream.iterate(n)(cubeDigits)

def stopRepeats[T](str: Stream[T]): Stream[T] = {
  val halfSpeed = str.flatMap(x => Seq(x, x))
  val result = halfSpeed.zip(str).drop(1).takeWhile{ case (h, s) => h != s }.map(_._2)
  if (result.isEmpty) str.take(1) else str
}
def cubesReach1(n: Int): Boolean = stopRepeats(cubes(n)).contains(1)
```

Exercise 2.5.2.4

```
def prod3(a: Set[Int], b: Set[Int], c: Set[Int]): Set[Set[Int]] =
  a.flatMap(x => b.flatMap(y => c.map(z => Set(x, y, z))))
```

Exercise 2.5.2.5

```
def prodSet(sets: Set[Set[Int]]): Set[Set[Int]] =
    sets.foldLeft(Set[Set[Int]](Set())) {
      // Combine each of results so far with each element in current set
      case (accumSet: Set[Set[Int]], currSet: Set[Int]) =>
        for {
          s <- accumSet
          i <- currSet
        } yield s + i
      }
```

Exercise 2.5.2.6

```
@tailrec def pairs(goal: Int, xs: Array[Int])(
  res: Set[(Int, Int)] = Set(), left: Int = 0, right: Int = xs.length): Set[(Int, Int)] =
    if (left == right) res else {
      val sum = xs(left) + xs(right - 1)
      val (newLeft, newRight, newRes) = if (sum == goal)
                  (left + 1, right, res + ((xs(left), xs(right - 1))))
                else if (sum < goal)
                  (left + 1, right, res)
                else (left, right - 1, res)
      pairs(goal, xs)(newRes, newLeft, newRight)
```

```
}
```

Exercise 2.5.2.7

```
def revSentence(sentence: String): String = sentence.split(" ").reverse.mkString(" ")
```

Exercise 2.5.2.8

```
def digitsOf(n: Int): Seq[Int] = Stream.iterate(n)(_ / 10).takeWhile(_ != 0).map(_ % 10).toList
def revDigits(n: Int): Int = digitsOf(n).foldLeft(0){case (acc, d) => acc * 10 + d }
def isPalindrome(n: Int): Boolean = revDigits(n) == n
```

Exercise 2.5.2.9

```
def findPalindrome(n: Int): Int =
  Stream.iterate(n) { x => x + revDigits(x) } .filter(isPalindrome).head
```

Exercise 2.5.2.13

```
def unfold2[A,B](init: A)(next: A => Option[(A,B)]): Stream[B] = next(init) match {
   case None           => Stream()
   case Some((a, b))   => Stream.cons(b, unfold2(a)(next))
}
```

Chapter 3

Exercise 3.3.2.3

```
def toList[A](nel: NEL[A]): List[A] = nel match {
  case Last(x)         => List(x)
  case More(x, tail)   => x :: toList(tail)
} // Not tail-recursive.
def toList[A](nel: NEL[A]): List[A] = foldLeft(nel)(Nil:List[A]) {
  (prev, x) =>  x :: prev
}.reverse // Tail-recursive, but performs two traversals.
```

Exercise 3.4.2.8

```
sealed trait EvenList[A]
final case class Lempty[A]() extends EvenList[A]
final case class Lpair[A](x: A, y: A, tail: EvenList[A]) extends EvenList

def fmap[A, B](f: A => B): EvenList[A] => EvenList[B] = {
  case Lempty()            => Lempty[B]()
  case Lpair(x, y, tail)   => Lpair[B](f(x), f(y), fmap(f)(tail))
}
```

Chapter 4

Exercise 4.4.2.2

```
@tailrec def convergeN[X](p: X => Boolean)(x:X)(m:Int)(f: X => X): Option[X] =  {
              if (m <= 0) None
              else if (p(x)) Some(x) else converge(p)(f(x))(m-1)(f)             }
// Defining it as def convergeN[X]: (X => Boolean) => X => Int => (X => X) => Option[X] = ???
// will break tail recursion!
```

Exercise 4.4.2.3

```
def recover[E, A]: Option[Either[E, A]] => (E => A) => Option[A] = {
  case None             => _ => None
  case Some(Right(a))   => _ => Some(a)
  case Some(Left(e))    => f => Some(f(e))
}
```

Exercise 4.4.2.15 **(a)** Choose $f^{:A\to A} \stackrel{\text{def}}{=} (_ \to z)$ and compute $(f \fatsemi h)(x) = h(z) \stackrel{!}{=} h(x)$, for any x. So, $h(x)$ equals a fixed value $h(z)$. It follows that $h(x)$ does not depend on x, i.e., h is a constant function.

(b) Choose $f^{:A\to B} \stackrel{\text{def}}{=} _ \to b$ and compute $(f \fatsemi h)(x) = h(b) \stackrel{!}{=} (g \fatsemi f)(x) = f(g(x)) = b$. It follows that $h(b) = b$ for any $b^{:B}$, so h is an identity function. Substitute that into the law and get $f = g \fatsemi f$ for any function f. Substitute $f \stackrel{\text{def}}{=} \text{id}$ into that and derive $\text{id} = g \fatsemi \text{id} = g$. So, g is the identity function.

Chapter 5

Exercise 5.1.4.2 The type expression is: $A \times \text{Int} + A \times \text{Char} + A \times \text{Float}$.

Chapter 7

Exercise 7.2.5.1 It is assumed that $\phi : \forall A.\, F^A \to G^A$ satisfies its naturality law: for any $f^{:A\to B}$,

$$f^{\uparrow F} \fatsemi \phi = \phi \fatsemi f^{\uparrow G} \quad .$$

(a) To verify the naturality law of $\phi^{\uparrow K}$: for any $f^{:A\to B}$,

$$f^{\uparrow F \uparrow K} \fatsemi \phi^{\uparrow K} \stackrel{?}{=} \phi^{\uparrow K} \fatsemi f^{\uparrow G \uparrow K} \quad ,$$

$$\text{composition under } {}^{\uparrow K} : \quad (f^{\uparrow F} \fatsemi \phi)^{\uparrow K} \stackrel{?}{=} (\phi \fatsemi f^{\uparrow G})^{\uparrow K} \quad .$$

The last equation holds due to the naturality law of ϕ.

(b) The naturality law of the pair product $(\phi \boxtimes \psi)$ says that for any $p^{:A\to B}$,

$$p^{\uparrow (F\times K)} \fatsemi (\phi \boxtimes \psi) \stackrel{?}{=} (\phi \boxtimes \psi) \fatsemi p^{\uparrow (G\times L)} \quad .$$

Begin with the left-hand side of this equation:

$$\begin{aligned}
\text{left-hand side}: \quad & p^{\uparrow (F\times K)} \fatsemi (\phi \boxtimes \psi) \\
\text{definition of } {}^{\uparrow (F\times K)}: \quad & = \left(f^{:F^A} \times k^{:K^A} \to (f \triangleright p^{\uparrow F}) \times (k \triangleright p^{\uparrow K})\right) \fatsemi (\phi \boxtimes \psi) \\
\text{definition of } \phi \boxtimes \psi: \quad & = f \times k \to \phi(f \triangleright p^{\uparrow F}) \times \psi(k \triangleright p^{\uparrow K}) = f \times k \to (f \triangleright p^{\uparrow F} \fatsemi \phi) \times (k \triangleright p^{\uparrow K} \fatsemi \psi) \quad .
\end{aligned}$$

To rewrite the right-hand side, introduce the function argument into $\phi \boxtimes \psi$:

$$\begin{aligned}
\text{right-hand side}: \quad & (\phi \boxtimes \psi) \fatsemi p^{\uparrow (G\times L)} = \left(f^{:F^A} \times k^{:K^A} \to (f \triangleright \phi) \times (k \triangleright \psi)\right) \fatsemi p^{\uparrow (G\times L)} \\
\text{definition of } {}^{\uparrow (G\times L)}: \quad & = f \times k \to (f \triangleright \underline{\phi \triangleright p^{\uparrow G}}) \times (k \triangleright \underline{\psi \triangleright p^{\uparrow L}}) = f \times k \to (f \triangleright \phi \fatsemi p^{\uparrow G}) \times (k \triangleright \psi \fatsemi p^{\uparrow L}) \quad .
\end{aligned}$$

The remaining differences between the two sides disappear due to the naturality laws of ϕ and ψ:

$$p^{\uparrow F} \fatsemi \phi = \phi \fatsemi p^{\uparrow G} \quad , \qquad p^{\uparrow K} \fatsemi \psi = \psi \fatsemi p^{\uparrow L} \quad .$$

The naturality law of the pair co-product $(\phi \boxplus \psi)$ says that for any $p^{:A\to B}$,

$$p^{\uparrow (F+K)} \fatsemi (\phi \boxplus \psi) \stackrel{?}{=} (\phi \boxplus \psi) \fatsemi p^{\uparrow (G+L)} \quad .$$

Begin with the left-hand side of this equation:

$$\text{left-hand side}: \quad p^{\uparrow (F+K)} \fatsemi (\phi \boxplus \psi)$$

$$\text{definitions of } {}^{\uparrow (F+K)} \text{ and of } \phi \boxplus \psi: \quad = \begin{array}{c|cc|} & F^B & K^B \\ \hline F^A & p^{\uparrow F} & \mathbb{0} \\ K^A & \mathbb{0} & p^{\uparrow K} \end{array} \fatsemi \begin{array}{c|cc|} & G^B & L^B \\ \hline F^B & \phi & \mathbb{0} \\ K^B & \mathbb{0} & \psi \end{array}$$

$$\text{matrix composition}: \quad = \begin{array}{c|cc|} & G^B & L^B \\ \hline F^A & p^{\uparrow F} \fatsemi \phi & \mathbb{0} \\ K^A & \mathbb{0} & p^{\uparrow K} \fatsemi \psi \end{array} \quad .$$

The right-hand side is rewritten in a similar way:

$$\text{right-hand side}: \quad (\phi \boxplus \psi) ⨾ p^{\uparrow(G+L)}$$

$$= \begin{array}{c||cc|} & G^A & L^A \\ \hline F^A & \phi & \mathbb{0} \\ K^A & \mathbb{0} & \psi \end{array} ⨾ \begin{array}{c||cc|} & G^B & L^B \\ \hline G^A & p^{\uparrow G} & \mathbb{0} \\ L^A & \mathbb{0} & p^{\uparrow L} \end{array} = \begin{array}{c||cc|} & G^B & L^B \\ \hline F^A & \phi ⨾ p^{\uparrow G} & \mathbb{0} \\ K^A & \mathbb{0} & \psi ⨾ p^{\uparrow L} \end{array} \quad .$$

The remaining differences between the two sides disappear due to the naturality laws of ϕ and ψ.

Chapter 8

Exercise 8.4.2.3 (a)

```
def monoidFunc[A: Monoid, R] = Monoid[R => A](
  { (x, y) => r => x(r) |+| y(r) }, _ => implicitly[Monoid[A]].empty
)
```

In the code notation:

$$f^{:R\to A} \oplus g^{:R\to A} \stackrel{\text{def}}{=} a \to f(a) \oplus_A g(a) \quad , \qquad e \stackrel{\text{def}}{=} (_ \to e_A) \quad .$$

Proof of monoid laws:

$$\begin{aligned} &a \triangleright ((f \oplus g) \oplus h) = (a \triangleright (f \oplus g)) \oplus_A h(a) = f(a) \oplus_A g(a) \oplus_A h(a) \quad . \\ &a \triangleright (f \oplus (g \oplus h)) = f(a) \oplus_A (a \triangleright (g \oplus h)) = f(a) \oplus_A g(a) \oplus_A h(a) \quad . \\ &a \triangleright (e \oplus f) = e(a) \oplus_A f(a) = e_A \oplus_A f(a) = f(a) = a \triangleright f \quad . \\ &a \triangleright (f \oplus e) = f(a) \oplus_A e(a) = f(a) \oplus_A e_A = f(a) = a \triangleright f \quad . \end{aligned}$$

Exercise 8.4.2.13 **(a)** A counterexample is the functor $F^A \stackrel{\text{def}}{=} R \to A$, where R is a fixed type.

(b) The function $C^A \times C^B \to C^{A+B}$ cannot be implemented for $C^A \stackrel{\text{def}}{=} (A \to P) + (A \to Q)$. This more complicated contrafunctor C is necessary because the simpler contrafunctor $C^A \stackrel{\text{def}}{=} A \to P$ does not provide a counterexample.

Exercise 8.4.2.15 Define the method ex_F as:

$$\text{ex}_F \stackrel{\text{def}}{=} x^{:F^A} \to x \triangleright (a^{:A} \to a \times 1)^{\uparrow F} \triangleright q^{A,\mathbb{1}} \triangleright \pi_1 \quad \text{or equivalently:} \quad \text{ex}_F \stackrel{\text{def}}{=} (a^{:A} \to a \times 1)^{\uparrow F} ⨾ q^{A,\mathbb{1}} ⨾ \pi_1 \quad .$$

To show that the naturality law ($f^{\uparrow F} ⨾ \text{ex}_F = \text{ex}_F ⨾ f$) holds, use the identity $(f \boxtimes g) ⨾ \pi_1 = \pi_1 ⨾ f$:

$$\begin{aligned} f^{\uparrow F} ⨾ \text{ex}_F &= \underline{f^{\uparrow F} ⨾ (a^{:A} \to a \times 1)^{\uparrow F}} ⨾ q^{A,\mathbb{1}} ⨾ \pi_1 = (a^{:A} \to a \times 1)^{\uparrow F} ⨾ \underline{(f \boxtimes \text{id})^{\uparrow F} ⨾ q^{A,\mathbb{1}}} ⨾ \pi_1 \\ &= (a^{:A} \to a \times 1)^{\uparrow F} ⨾ q^{A,\mathbb{1}} ⨾ \underline{(f \boxtimes \text{id}^{\uparrow F}) ⨾ \pi_1} = \underline{(a^{:A} \to a \times 1)^{\uparrow F} ⨾ q^{A,\mathbb{1}} ⨾ \pi_1} ⨾ f = \text{ex}_F ⨾ f \quad . \end{aligned}$$

Given a method ex_F, define q as:

$$q^{A,B} \stackrel{\text{def}}{=} f^{:F^{A\times B}} \to (f \triangleright \pi_1^{\uparrow F} \triangleright \text{ex}_F^A) \times (f \triangleright \pi_2^{\uparrow F}) \quad \text{or equivalently:} \quad q^{A,B} \stackrel{\text{def}}{=} \Delta ⨾ (\pi_1^{\uparrow F} \boxtimes \pi_2^{\uparrow F}) ⨾ (\text{ex}_F^A \boxtimes \text{id}) \quad .$$

Show that the required laws hold for q. Identity law:

$$f \triangleright q^{\mathbb{1},B} = (f \triangleright \pi_1^{\uparrow F} \triangleright \text{ex}_F^{\mathbb{1}}) \times (f \triangleright \pi_2^{\uparrow F}) = 1 \times (f \triangleright \pi_2^{\uparrow F})$$

because $\text{ex}_F^{\mathbb{1}}$ produces a value of type $\mathbb{1}$, which can only be 1.

To prove the naturality law:

$$\begin{aligned} (f^{:A\to C} \boxtimes g^{:B\to D})^{\uparrow F} ⨾ \Delta ⨾ (\pi_1^{\uparrow F} \boxtimes \pi_2^{\uparrow F}) ⨾ (\text{ex}_F^A \boxtimes \text{id}) &= \Delta ⨾ (\pi_1^{\uparrow F} \boxtimes \pi_2^{\uparrow F}) ⨾ (\text{ex}_F^A \boxtimes \text{id}) ⨾ f \boxtimes (g^{\uparrow F}) \\ \Delta ⨾ ((f \boxtimes g)^{\uparrow F} \boxtimes (f \boxtimes g)^{\uparrow F}) ⨾ (\pi_1^{\uparrow F} \boxtimes \pi_2^{\uparrow F}) ⨾ (\text{ex}_F^A \boxtimes \text{id}) &= \Delta ⨾ (\pi_1^{\uparrow F} ⨾ \text{ex}_F ⨾ f) \boxtimes (\pi_2^{\uparrow F} ⨾ g^{\uparrow F}) \\ \Delta ⨾ (\pi_1^{\uparrow F} ⨾ \underline{f^{\uparrow F} ⨾ \text{ex}_F}) \boxtimes (\pi_2^{\uparrow F} ⨾ g^{\uparrow F}) &= \Delta ⨾ (\pi_1^{\uparrow F} ⨾ \underline{\text{ex}_F ⨾ f}) \boxtimes (\pi_2^{\uparrow F} ⨾ g^{\uparrow F}) \end{aligned}$$

To prove the associativity law, write the left-hand side as:

$$f^{:F^{A\times B\times C}} \triangleright q^{A,B\times C} \fatsemi (\mathrm{id}^A \boxtimes q^{B,C}) = (f \triangleright \pi_1^{\uparrow F} \triangleright \mathrm{ex}_F^A) \times (f \triangleright (a\times b\times c \to b\times c)^{\uparrow F} \triangleright q^{B,C})$$
$$= (f \triangleright \pi_1^{\uparrow F} \triangleright \mathrm{ex}_F^A) \times (f \triangleright (a\times b\times c \to b\times c)^{\uparrow F} \triangleright \pi_1^{\uparrow F} \triangleright \mathrm{ex}_F^B) \times (f \triangleright (a\times b\times c \to b\times c)^{\uparrow F} \triangleright \pi_2^{\uparrow F})$$
$$= (f \triangleright \mathrm{ex}_F \triangleright (a\times b\times c \to a)) \times (f \triangleright \mathrm{ex}_F \triangleright (a\times b\times c \to b)) \times (f \triangleright (a\times b\times c \to c)^{\uparrow F}) \quad .$$

The right-hand side is then simplified to the same expression:

$$f^{:F^{A\times B\times C}} \triangleright q^{A\times B,C} = (f \triangleright \mathrm{ex}_F \triangleright (a\times b\times c \to a\times b)) \times (f \triangleright (a\times b\times c \to c)^{\uparrow F})$$
$$= (f \triangleright \mathrm{ex}_F \triangleright (a\times b\times c \to a)) \times (f \triangleright \mathrm{ex}_F \triangleright (a\times b\times c \to b)) \times (f \triangleright (a\times b\times c \to c)^{\uparrow F}) \quad .$$

Chapter 9

Exercise 9.2.5.6 Starting from `liftOpt`'s law, derive the naturality law:

$$\mathrm{liftOpt}\,(g) \fatsemi \mathrm{liftOpt}\,(f \fatsemi \mathrm{pu}_{\mathrm{Opt}}) = \mathrm{liftOpt}(f \fatsemi \mathrm{pu}_{\mathrm{Opt}} \diamond_{\mathrm{Opt}} g) = \mathrm{liftOpt}(f \fatsemi g) \quad .$$

Now use the naturality-identity law and get $\mathrm{liftOpt}\,(f \fatsemi \mathrm{pu}_{\mathrm{Opt}}) = f^{\downarrow C}$. The result is the naturality law $\mathrm{liftOpt}\,(f \fatsemi g) = \mathrm{liftOpt}\,(g) \fatsemi f^{\downarrow C}$.

Exercise 9.3.2.7 A counterexample is the functor $F^A \stackrel{\text{def}}{=} \mathbb{1} + A \times (Z \to A)$. For this functor, one can implement `deflate`'s type signature, but the code cannot obey the identity law because it must always return $1 + \mathbb{0}$.

Chapter 10

Exercise 10.2.9.4 The given non-standard implementation of `flatten` will return the same results as the standard `flatten` method of the `List` type constructor, except if one of the nested sub-lists is empty. In that case, the `flatten` function returns an empty list (unlike `List`'s standard `flatten` method). The `pure` method remains unchanged. The code is:

```
def pure[A](x: A): List[A] = List(x) // pu_L
def flatten[A](p: List[List[A]]): List[A] = if (p.exists(_.isEmpty)) Nil else p.flatten  // ftn_L
```

To verify the monad laws, we use the known fact that the standard `List` monad obeys the laws. So, we only need to check the laws in the cases when the new `flatten` function is applied to a value of type `List[List[A]]` having an empty nested sub-list. That case cannot arise in the identity laws since neither $\mathrm{pu}_L(x^{:A})$ nor $\mathrm{pu}_L^{\uparrow L}(x^{:\mathrm{List}^A})$ ever returns a value with a nested empty sub-list:

```
pure(List(a, b, c)) == List(List(a, b, c))
List(a, b, c).map(pure) == List(List(a), List(b), List(c))
```

Applying `flatten` to these values gives the initial list `List(a, b, c)`. So, both identity laws hold.

It remains to check the associativity law, which is an equality between functions $\mathrm{ftn}_L \fatsemi \mathrm{ftn}_L$ and $\mathrm{ftn}_L^{\uparrow L} \fatsemi \mathrm{ftn}_L$ of type `List[List[List[A]]] => List[A]`. A value p of type `List[List[List[A]]]` could contain a nested empty list at the first and/or the second nesting depth, for instance:

```
val p1: List[List[List[Int]]] = List(List[List[Int]](), List(List(123), List(456)))
val p2: List[List[List[Int]]] = List(List(List[Int](), List(123)), List(List(456)))
```

Whenever p contains an empty sub-list at the first nesting depth, we will have $p \triangleright \mathrm{ftn}_L = \mathrm{Nil}$ because ftn_L explicitly checks for the existence of an empty sub-list. So, $p \triangleright \mathrm{ftn}_L \fatsemi \mathrm{ftn}_L = \mathrm{Nil}$. On the other hand, $p \triangleright \mathrm{ftn}_L^{\uparrow L}$ will be again a list containing an empty sub-list, for example:

```
scala> p1.map(flatten)
res0: List[List[Int]] = List(List(), List(123, 456))
```

We will then have $p \triangleright \text{ftn}_L^{\uparrow L} \triangleright \text{ftn}_L = \text{Nil}$, and the law holds. If p contains an empty sub-list at the *second* nesting depth, $p \triangleright \text{ftn}_L$ will contain an empty sub-list at the first nesting depth, for example:

```
scala> flatten(p2)
res1: List[List[Int]] = List(List(), List(123), List(456))
```

So $p \triangleright \text{ftn}_L \fatsemi \text{ftn}_L = \text{Nil}$. On the other hand, $p \triangleright \text{ftn}_L^{\uparrow L}$ will also be a list with an empty sub-list at the first nesting depth, for example:

```
scala> p2.map(flatten)
res2: List[List[Int]] = List(List(), List(456))
```

Applying ftn_L to the last result, we will get an empty list. Thus, $p \triangleright \text{ftn}_L \triangleright \text{ftn}_L = \text{Nil}$, and the law again holds. So, we have shown that the associativity law holds for the non-standard `List` monad.

Exercise 10.2.9.11 The values p and q must have type $\text{List}^{\text{List}^A}$. There are two possibilities: p is an empty list ($p = 1 + \mathbb{0}$), and $p = \mathbb{0} + h \times t$. If p is empty, so is $p \triangleright \text{ftn}$ and the law holds. In the other case, we have (due to the code of $+\!\!+$) that:

$$(\mathbb{0} + h \times t) +\!\!+ q = \mathbb{0} + h \times (t +\!\!+ q) \quad ,$$

and so:

$$\begin{aligned}
& (p +\!\!+ q) \triangleright \text{ftn} = (\mathbb{0} + h \times (t +\!\!+ q)) \triangleright \text{ftn} = h +\!\!+ \underline{(t +\!\!+ q) \triangleright \overline{\text{ftn}}} \\
\text{inductive assumption}: \quad & = \underline{h +\!\!+ (t \triangleright \overline{\text{ftn}})} +\!\!+ (q \triangleright \overline{\text{ftn}}) \\
\text{code of ftn}: \quad & = (\mathbb{0} + h \times t) \triangleright \overline{\text{ftn}} +\!\!+ (q \triangleright \overline{\text{ftn}}) = (p \triangleright \text{ftn}) +\!\!+ (q \triangleright \text{ftn}) \quad .
\end{aligned}$$

Exercise 10.2.9.13 Define the empty element (e_M) of the monoid M^W as:

$$e_M \stackrel{\text{def}}{=} \text{pu}_M(e_W) \quad .$$

The binary operation $\oplus_M$ of the monoid M^W may be implemented through $\oplus_W$ as:

$$p \oplus_M q \stackrel{\text{def}}{=} p \triangleright \text{flm}_M\left(u^{:W} \to q \triangleright (v^{:W} \to u \oplus_W v)^{\uparrow M}\right) \quad .$$

To check the left identity law of M^W:

$$\begin{aligned}
\text{expect to equal } p: \quad & e_M \oplus_M p = e_W \triangleright \underline{\text{pu}_M \triangleright \text{flm}_M}\left(u^{:W} \to p \triangleright (v^{:W} \to u \oplus_W v)^{\uparrow M}\right) \\
& = e_W \triangleright \left(u^{:W} \to p \triangleright (v^{:W} \to u \oplus_W v)^{\uparrow M}\right) = p \triangleright (v \to \underline{e_W \oplus_W v})^{\uparrow M} \\
\text{left identity law of } W: \quad & = p \triangleright (v \to v)^{\uparrow M} = p \triangleright \text{id}^{\uparrow M} = p \quad .
\end{aligned}$$

To check the right identity law of M^W:

$$\begin{aligned}
\text{expect to equal } p: \quad & p \oplus_M e_M = p \triangleright \text{flm}_M\left(u^{:W} \to e_W \triangleright \underline{\text{pu}_M \triangleright (v^{:W} \to u \oplus_W v)^{\uparrow M}}\right) \\
\text{naturality of } \text{pu}_M: \quad & = p \triangleright \text{flm}_M\left(u^{:W} \to \underline{e_W \triangleright (v^{:W}} \to u \oplus_W v) \triangleright \text{pu}_M\right) \\
\text{apply function}: \quad & = p \triangleright \text{flm}_M\left(u^{:W} \to (\underline{u \oplus_W e_W}) \triangleright \text{pu}_M\right) \\
\text{right identity law of } W: \quad & = p \triangleright \text{flm}_M\left(u^{:W} \to u \triangleright \text{pu}_M\right) = p \triangleright \text{id} = p \quad .
\end{aligned}$$

To check the associativity law of M^W, we use the associativity law of flm_M:

$$\begin{aligned}
\text{left-hand side}: \quad & (p \oplus_M q) \oplus_M r \\
& = p \triangleright \underline{\text{flm}_M}\left(u \to q \triangleright (v \to u \oplus_W v)^{\uparrow M}\right) \triangleright \underline{\text{flm}_M}\left(t \to r \triangleright (w \to t \oplus_W w)^{\uparrow M}\right) \\
\text{associativity of } \text{flm}_M: \quad & = p \triangleright \text{flm}_M\left(\underline{(}u \to q \triangleright (v \to u \oplus_W v)^{\uparrow M}\underline{)} \fatsemi \text{flm}_M\left(t \to r \triangleright (w \to t \oplus_W w)^{\uparrow M}\right)\right) \\
\text{compute composition}: \quad & = p \triangleright \text{flm}_M\left(u \to q \triangleright (v \to u \oplus_W v)^{\uparrow M} \triangleright \underline{\text{flm}_M}\left(t \to r \triangleright (w \to t \oplus_W w)^{\uparrow M}\right)\right) \\
\text{naturality of } \text{flm}_M: \quad & = p \triangleright \text{flm}_M\left(u \to q \triangleright \text{flm}_M\left(v \to \underline{u \oplus_W v}\right) \underline{\fatsemi (t} \to r \triangleright (w \to \underline{t} \oplus_W w)^{\uparrow M}\right)\right) \\
\text{compute composition}: \quad & = p \triangleright \text{flm}_M\left(u \to q \triangleright \text{flm}_M\left(v \to r \triangleright (w \to (u \oplus_W v) \oplus_W w)^{\uparrow M}\right)\right) \quad .
\end{aligned}$$

Now write the right-hand side of the law:

$$\begin{aligned}
\text{right-hand side}: \quad & p \oplus_M (q \oplus_M r) = p \triangleright \mathrm{flm}_M \big(u \to (q \oplus r) \triangleright (t \to u \oplus_W t)^{\uparrow M}\big) \\
\text{substitute } q \oplus r: \quad & = p \triangleright \mathrm{flm}_M \big(u \to q \triangleright \mathrm{flm}_M (v \to r \triangleright (w \to v \oplus_W w)^{\uparrow M}) \triangleright \underline{(t \to u \oplus_W t)^{\uparrow M}}\big) \\
\text{naturality of } \mathrm{flm}_M: \quad & = p \triangleright \mathrm{flm}_M \big(u \to q \triangleright \mathrm{flm}_M (v \to r \triangleright \underline{(w \to u \oplus_W w)^{\uparrow M} \fatsemi (t \to u \oplus_W t)^{\uparrow M}})\big) \\
\text{composition under } {}^{\uparrow M}: \quad & = p \triangleright \mathrm{flm}_M \big(u \to q \triangleright \mathrm{flm}_M (v \to r \triangleright (w \to u \oplus_W (v \oplus_W w))^{\uparrow M})\big) \quad .
\end{aligned}$$

The difference between the two sides now vanishes due to the assumed associativity law of W:

$$(u \oplus_W v) \oplus_W w \overset{!}{=} u \oplus_W (v \oplus_W w) \quad .$$

Exercise 10.2.9.14 **(b)** The code is converted into monad methods like this:

$$\begin{aligned}
r_1 &= p \triangleright \mathrm{flm}_M (x \to q \triangleright (y \to f(x,y))^{\uparrow M}) \quad , \\
r_2 &= q \triangleright \mathrm{flm}_M (y \to p \triangleright (x \to f(x,y))^{\uparrow M}) \quad .
\end{aligned}$$

For a commutative monad M, we have $r_1 = r_2$. The monoid operation $\oplus_M$ is defined by:

$$p^{:M^{\mathbb{1}}} \oplus_M q^{:M^{\mathbb{1}}} \overset{\text{def}}{=} p \triangleright \mathrm{flm}_M (1 \to q) \quad .$$

Commutativity of $\oplus_M$ means that:

$$p \triangleright \mathrm{flm}_M (1 \to q) = p \oplus_M q \overset{?}{=} q \oplus_M p = q \triangleright \mathrm{flm}_M (1 \to p) \quad .$$

Use $f(x,y) \overset{\text{def}}{=} 1$ in the definitions of r_1 and r_2; then the above equation is equivalent to $r_1 = r_2$.

Exercise 10.2.9.16 The second definition, $\mathrm{pu}_L \overset{\text{def}}{=} a \to \mathbb{0} + \mathrm{pu}_F(a)$, fails the right identity law:

$$\mathrm{pu}_L^{\uparrow L} \fatsemi \mathrm{ftn}_L = \begin{array}{c||ccc|} & A & F^A & F^{L^A} \\ \hline A & \mathbb{0} & \mathrm{pu}_F & \mathbb{0} \\ F^A & \mathbb{0} & \mathbb{0} & \mathrm{pu}_L^{\uparrow F} \end{array} \fatsemi \begin{array}{c||cc|} & A & F^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ F^A & \mathbb{0} & \mathrm{id} \\ F^{L^A} & \mathbb{0} & \gamma^{\uparrow F} \fatsemi \mathrm{ftn}_F \end{array} = \begin{array}{c||cc|} & A & F^A \\ \hline A & \mathbb{0} & \mathrm{pu}_F \\ F^A & \mathbb{0} & \ldots \end{array} \neq \mathrm{id} \quad .$$

This matrix cannot be equal to the identity function because it has a missing diagonal element.

Exercise 10.2.9.19 If M is a semimonad, we have the Kleisli composition $\diamond_M$ that satisfies the associativity law. Define $\diamond_L$ by:

$$f^{:A \to M^{M^B}} \diamond_L g^{:B \to M^{M^C}} \overset{\text{def}}{=} f \fatsemi \mathrm{ftn}_M \diamond_M g \quad .$$

Here, the expression of the form $f \fatsemi h \diamond_M g$ does not need parentheses (see Statement 10.2.6.4).

It is inconvenient to mix the Kleisli composition and the `flatten` method, so we express `flatten` as:

$$\mathrm{ftn}_M^{:M^{M^A} \to M^A} = \mathrm{flm}_M (\mathrm{id}^{:M^A \to M^A}) = \mathrm{id}^{:M^{M^A} \to M^{M^A}} \diamond_M \mathrm{id}^{:M^A \to M^A} \quad .$$

For brevity, we will omit type annotations from now on. So, we can express the Kleisli composition $\diamond_L$ through $\diamond_M$ by:

$$f \diamond_L g \overset{\text{def}}{=} f \fatsemi \underline{(\mathrm{id} \diamond_M \mathrm{id}) \diamond_M g} = f \fatsemi (\mathrm{id} \diamond_M (\mathrm{id} \diamond_M g)) = (f \fatsemi \mathrm{id}) \diamond_M (\mathrm{id} \diamond_M g) = f \diamond_M \mathrm{id} \diamond_M g \quad .$$

Associativity of $\diamond_L$ then follows from associativity of $\diamond_M$ as:

$$\begin{aligned}(f \diamond_L g) \diamond_L h &= (f \diamond_M \mathrm{id} \diamond_M g) \diamond_M \mathrm{id} \diamond_M h = f \diamond_M \mathrm{id} \diamond_M g \diamond_M \mathrm{id} \diamond_M h \quad , \\ f \diamond_L (g \diamond_L h) &= f \diamond_M \mathrm{id} \diamond_M (g \diamond_M \mathrm{id} \diamond_M h) = f \diamond_M \mathrm{id} \diamond_M g \diamond_M \mathrm{id} \diamond_M h \quad .\end{aligned}$$

This definition of $\diamond_L$ corresponds to a definition of ftn_L that flattens the *first* three layers of M in $M \circ M \circ M \circ M$:

$$\mathrm{ftn}_L \stackrel{\mathrm{def}}{=} \mathrm{ftn}_M \fatsemi \mathrm{ftn}_M \quad .$$

An alternative definition will flatten the *last* three layers:

$$\mathrm{ftn}_L \stackrel{\mathrm{def}}{=} (\mathrm{ftn}_M \fatsemi \mathrm{ftn}_M)^{\uparrow M} \quad .$$

Both definitions satisfy the associativity law and so define a semimonad L,

$$\mathrm{ftn}_L^{\uparrow L} \fatsemi \mathrm{ftn}_L = \mathrm{ftn}_L \fatsemi \mathrm{ftn}_L \quad .$$

Let us verify that directly. For the first definition:

$$\begin{aligned}& \underline{(\mathrm{ftn}_M \fatsemi \mathrm{ftn}_M)^{\uparrow M \uparrow M} \fatsemi \mathrm{ftn}_M} \fatsemi \mathrm{ftn}_M = \mathrm{ftn}_M \fatsemi \underline{(\mathrm{ftn}_M \fatsemi \mathrm{ftn}_M)^{\uparrow M}} \fatsemi \mathrm{ftn}_M \\ \text{associativity of } M: \quad & = \underline{\mathrm{ftn}_M \fatsemi \mathrm{ftn}_M^{\uparrow M \uparrow M}} \fatsemi \mathrm{ftn}_M^{\uparrow M} \fatsemi \mathrm{ftn}_M = \underline{\mathrm{ftn}_M^{\uparrow M} \fatsemi \mathrm{ftn}_M} \fatsemi \underline{\mathrm{ftn}_M^{\uparrow M} \fatsemi \mathrm{ftn}_M} \\ \text{associativity of } M: \quad & = \mathrm{ftn}_M \fatsemi \mathrm{ftn}_M \fatsemi \mathrm{ftn}_M \fatsemi \mathrm{ftn}_M \quad .\end{aligned}$$

For the second definition, we just apply $^{\uparrow M}$ to the preceding derivation.

A definition of `flatten` that flattens separately the first two and the last two layers of M ($\mathrm{ftn}_L \stackrel{\mathrm{def}}{=} \mathrm{ftn}_M \fatsemi \mathrm{ftn}_M^{\uparrow M}$) will fail the associativity law when M is chosen in a suitable way.

If M is a full monad, we have pu_M that satisfies the identity laws. Define pu_L by:

$$\mathrm{pu}_L \stackrel{\mathrm{def}}{=} \mathrm{pu}_M \fatsemi \mathrm{pu}_M \quad .$$

There is no other way of defining pu_L since we cannot obtain a value of type M^A other than via pu_M.

With any of the two possible definitions of ftn_L, at least one of the identity laws for $\diamond_L$ fails.

For the definition of ftn_L that flattens the first three layers, the right identity law will fail:

$$\mathrm{pu}_L^{\uparrow L} \fatsemi \mathrm{ftn}_L = \underline{(\mathrm{pu}_M \fatsemi \mathrm{pu}_M)^{\uparrow M \uparrow M} \fatsemi \mathrm{ftn}_M} \fatsemi \mathrm{ftn}_M = \mathrm{ftn}_M \fatsemi \mathrm{pu}_M^{\uparrow M} \fatsemi \underline{\mathrm{pu}_M^{\uparrow M} \fatsemi \mathrm{ftn}_M} = \mathrm{ftn}_M \fatsemi \mathrm{pu}_M^{\uparrow M} \quad .$$

In general, this function is not equal to an identity function, because `flatten` merges the two monadic layers of M and in that way loses information about a value of type M^{M^A}.

For the definition of ftn_L that flattens the last three layers, the left identity law will fail:

$$\mathrm{pu}_L \fatsemi \mathrm{ftn}_L = \mathrm{pu}_M \fatsemi \underline{\mathrm{pu}_M \fatsemi (\mathrm{ftn}_M} \fatsemi \mathrm{ftn}_M)^{\uparrow M} = \underline{\mathrm{pu}_M \fatsemi \mathrm{ftn}_M} \fatsemi \mathrm{pu}_M \fatsemi \mathrm{ftn}_M^{\uparrow M} = \mathrm{pu}_M \fatsemi \mathrm{ftn}_M^{\uparrow M} = \mathrm{ftn}_M \fatsemi \mathrm{pu}_M \quad .$$

In general, this function is not equal to an identity function.

Chapter 11

Exercise 11.5.6.6 We may choose $f = \mathrm{id}$ and derive $u = v$ from the given law. Conversely, if $u = v$ then $u \fatsemi f^{\uparrow F} = v \fatsemi f^{\uparrow F}$ for any function f.

Exercise 11.5.6.11 To verify the law, write:

$$\begin{aligned}& \mathrm{ap}\,(r)(\mathrm{pu}_L(a)) = \mathrm{zip}(r \times \mathrm{pu}_L(a)) \triangleright \mathrm{eval}^{\uparrow L} \\ \text{right identity law of zip}: \quad & = r \triangleright (f \to f \times a)^{\uparrow L} \fatsemi \mathrm{eval}^{\uparrow L} \\ \text{composition under } ^{\uparrow L}: \quad & = r \triangleright (f \to f(a)) \quad .\end{aligned}$$

Exercise 11.5.6.2 **(a)** We cannot implement `zip` as a fully parametric function having this type:

$$\text{zip}_F(p^{:(A\to P)\to Q} \times q^{:(B\to P)\to Q}) \stackrel{\text{def}}{=} h^{:A\times B\to P} \to ???^{:Q} \quad .$$

```
import io.chymyst.ch._
type S[A] = (A => P) => Q

scala> def zip[A, B](p: S[A], q: S[B]): S[(A, B)] = implement
type ((A => P) => Q) => ((B => P) => Q) => (Tuple2[A,B] => P) => Q cannot be implemented
```

The reason it cannot be implemented is that the only way of getting a value of type Q is to call the given functions p or q. But we cannot call p or q since we cannot supply their arguments: we have a function of type $A \times B \to P$, and we cannot produce a function of type $A \to P$ or $B \to P$ out of it.

The only solution is for `zip` to ignore its arguments and always return the empty value e_Q. However, that implementation loses information and would fail the identity laws.

(b) We *can* implement the type signature of `zip`, but only in a trivial way:

$$\text{zip}_F(p^{:(A\to P)\to \mathbb{1}+A} \times q^{:(B\to P)\to \mathbb{1}+B}) \stackrel{\text{def}}{=} h^{:A\times B\to P} \to ???^{:\mathbb{1}+A\times B} \quad .$$

The functions p and q cannot be called since we cannot supply their arguments, just as in part **(a)**. The only solution is that `zip` should ignore its arguments and always return $1 + \mathbb{0}^{:A\times B}$. However, that implementation loses information and would fail the identity laws.

Exercise 11.5.6.8 We define $\text{ex}_F(h \times g) \stackrel{\text{def}}{=} \text{ex}_H(h)$ and obtain:

$$\text{ex}_F(\text{zip}_F(h_1 \times g_1 \times h_2 \times g_2)) = \text{ex}_F(\text{zip}_H(h_1 \times h_2) \times \text{zip}_G(g_1 \times g_2)) = \text{ex}_H(\text{zip}_H(h_1 \times h_2)) \quad ,$$
$$\text{ex}_F(h_1 \times g_1) \times \text{ex}_F(h_2 \times g_2) = \text{ex}_H(h_1) \times \text{ex}_H(h_2) \quad .$$

The two sides are equal due to the compatibility law of zip_H and ex_H.

Exercise 11.5.6.15 The first and the third occurrences of A in Q^A are contravariant while the others are covariant. So, we define a profunctor $P^{X,Y}$ and get:

$$P^{X,Y} \stackrel{\text{def}}{=} (X \to \text{Int}) \times Y \times (X \to Y) \quad , \qquad Q^A = P^{A,A} \quad .$$

Exercise 11.5.6.16(b) Hint: Use $P^A \stackrel{\text{def}}{=} A \to Z$ as a counterexample.

Chapter 12

Exercise 12.3.9.1 First, we express `consume` via `sequence` and back:

$$\text{consume}_L(f^{:L^A\to B})(p^{:L^{F^A}}) = p \triangleright \text{seq}_L \triangleright f^{\uparrow F} \quad , \quad \text{or equivalently}: \quad \text{consume}_L(f) = \text{seq}_L \,⨾\, f^{\uparrow F} \quad , \tag{E.1}$$

$$\text{seq}_L = \text{consume}_L(\text{id}^{:L^A\to L^A}) \quad . \tag{E.2}$$

The pattern is similar to that in Statement 10.3.2.1 and to the equivalence of `sequence` and `traverse`.

We need to derive the equivalence between `consume` and `sequence` in both directions. To figure out the necessary naturality law, we begin with the direction that restores `consume` from `sequence` because `consume` is the more complicated function (having two type parameters).

(a) Given a function `consume`, we define `sequence` via Eq. (E.2) and then define a new function `consume'` via Eq. (E.1). Then we need to show that `consume'` equals `consume`. For an arbitrary $f^{:L^A\to B}$, we write:

$$\text{expect to equal consume}_L(f): \quad \text{consume}'_L(f) = \text{seq}_L \,⨾\, f^{\uparrow F} = \text{consume}_L(\text{id}) \,⨾\, f^{\uparrow F} = ???$$

If we know nothing about `consume`, we cannot conclude that $\text{consume}_L(f) \stackrel{?}{=} \text{consume}_L(\text{id}) \,⨾\, f^{\uparrow F}$ because the function f is arbitrary and the value $\text{consume}_L(f)$ does not need to be related in any way to $\text{consume}_L(\text{id})$. We must use a naturality law involving a lifted function applied after `consume`:

$$\text{consume}_L(f^{:L^A\to B}) \,⨾\, (g^{:B\to C})^{\uparrow F} = \text{consume}_L(f \,⨾\, g) \quad . \tag{E.3}$$

If this law holds, we can derive the required equation:

$$\text{consume}'_L(f) = \text{consume}_L(\text{id}) \fatsemi f^{\uparrow F} = \text{consume}_L(\text{id} \fatsemi f) = \text{consume}_L(f) \quad .$$

(b) Given a function `sequence`, we first define `consume` via Eq. (E.1) and then define a new function `sequence'` via Eq. (E.2). We then show that `sequence'` equals `sequence`:

$$\text{seq}'_L = \text{consume}_L(\text{id}) = \text{seq}_L \fatsemi \underline{\text{id}^{\uparrow F}} = \text{seq} \quad .$$

If a function `consume` is defined via `sequence`, the naturality law (E.3) will hold automatically:

$$\text{consume}_L(f) \fatsemi g^{\uparrow F} = \text{seq}_L \fatsemi \underline{f^{\uparrow F} \fatsemi g^{\uparrow F}} = \text{seq}_L \fatsemi (f \fatsemi g)^{\uparrow F} = \text{consume}_L\,(f \fatsemi g) \quad .$$

Exercise 12.3.9.7 The naturality law of `ftr` with respect to the type parameter B is:

$$\text{ftr}\,(f^{:A\to G^B}) \fatsemi (g^{:B\to C})^{\uparrow H} = \text{ftr}\,(f \fatsemi g^{\uparrow G}) \quad . \tag{E.4}$$

The type signature of `tr` has only one type parameter, so `tr` has only one naturality law:

$$(f^{:A\to B})^{\uparrow G\uparrow F} \fatsemi \text{tr} = \text{tr} \fatsemi f^{\uparrow H} \quad . \tag{E.5}$$

1) We show that the naturality law (E.5) of `tr` follows from the two naturality laws of `ftr`. Express `tr` as ftr (id) and substitute into the two sides of `tr`'s naturality law:

$$\begin{aligned}
\text{left-hand side}: \quad & f^{\uparrow G\uparrow F} \fatsemi \text{tr} = f^{\uparrow G\uparrow F} \fatsemi \text{ftr}\,(\text{id}) \\
\text{naturality law (10.26)}: \quad & = \text{ftr}\,(f^{\uparrow G} \fatsemi \text{id}) = \text{ftr}\,(f^{\uparrow G}) \quad , \\
\text{right-hand side}: \quad & \text{tr} \fatsemi f^{\uparrow H} = \text{ftr}\,(\text{id}) \fatsemi f^{\uparrow H} \\
\text{naturality law (E.4)}: \quad & = \text{ftr}\,(\text{id} \fatsemi f^{\uparrow G}) = \text{ftr}\,(f^{\uparrow G}) \quad .
\end{aligned}$$

The two sides of the law are now equal.

2) We show that the second naturality law of `ftr` follows from the naturality law of `tr`. Express `ftr` through `tr` as $\text{ftr}\,(f) = f^{\uparrow F} \fatsemi \text{tr}$, and substitute into the two sides of `ftr`'s naturality law (E.4):

$$\begin{aligned}
\text{left-hand side}: \quad & \text{ftr}\,(f) \fatsemi g^{\uparrow H} = f^{\uparrow F} \fatsemi \underline{\text{tr} \fatsemi g^{\uparrow H}} \\
\text{naturality law (E.5) of tr}: \quad & = f^{\uparrow F} \fatsemi g^{\uparrow G\uparrow F} \fatsemi \text{tr} = (f \fatsemi g^{\uparrow G})^{\uparrow F} \fatsemi \text{tr} \quad , \\
\text{right-hand side}: \quad & \text{ftr}\,(f \fatsemi g^{\uparrow G}) = (f \fatsemi g^{\uparrow G})^{\uparrow F} \fatsemi \text{tr} \quad .
\end{aligned}$$

The two sides of the law are now equal.

Exercise 12.3.9.8 To verify the identity law (12.25), set $F = \text{Id}$ in the definition (12.27) of seq_L:

$$\begin{aligned}
\text{expect to equal id}: \quad & \text{seq}_L^{\text{Id},A} = (\text{seq}_N^{\text{Id},A})^{\uparrow M} \fatsemi \text{seq}_M^{\text{Id},N^A} \\
\text{identity laws of seq}_M \text{ and seq}_N: \quad & = \text{id}^{\uparrow M} \fatsemi \text{id} = \text{id} \quad .
\end{aligned}$$

To verify the composition law (12.26), begin with the left-hand side:

$$\begin{aligned}
\text{expect to equal seq}_L^{F\circ G,A}: \quad & \text{seq}_L^{F,G^A} \fatsemi (\text{seq}_L^{G,A})^{\uparrow F} \\
& = (\text{seq}_N^{F,G^A})^{\uparrow M} \fatsemi \underline{\text{seq}_M^{F,N^{G^A}} \fatsemi \big((\text{seq}_N^{G,A})^{\uparrow M} \fatsemi \text{seq}_M^{G,N^A}\big)^{\uparrow F}} \\
\text{naturality law of seq}_M: \quad & = (\text{seq}_N^{F,G^A})^{\uparrow M} \fatsemi (\text{seq}_N^{G,A})^{\uparrow F\uparrow M} \fatsemi \text{seq}_M^{F,G^{N^A}} \fatsemi (\text{seq}_M^{G,N^A})^{\uparrow F} \\
\text{composition laws of seq}_M \text{ and seq}_N: \quad & = (\text{seq}_N^{F\circ G,A})^{\uparrow M} \fatsemi \text{seq}_M^{F\circ G,N^A} = \text{seq}_L^{F\circ G,A} \quad .
\end{aligned}$$

Exercise 12.3.9.13 The empty elements of the monoids M^R and M^S are, by definition, $\mathrm{pu}_M(e_R)$ and $\mathrm{pu}_M(e_S)$ respectively. The binary operations of M^R and M^S are:

$$p \underset{M^R}{\oplus} q \stackrel{\text{def}}{=} p \triangleright \mathrm{flm}_M\big(u^{:R} \to q \triangleright (v^{:R} \to u \oplus_R v)^{\uparrow M}\big) \quad ,$$
$$p \underset{M^S}{\oplus} q \stackrel{\text{def}}{=} p \triangleright \mathrm{flm}_M\big(u^{:S} \to q \triangleright (v^{:S} \to u \oplus_S v)^{\uparrow M}\big) \quad .$$

To verify the properties of the monoid morphism, begin with the identity law:

$$\text{expect to equal } e_{M^S}: \quad \underline{e_{M^R}} \triangleright \phi^{\uparrow M} = e_R \triangleright \underline{\mathrm{pu}_M \triangleright \phi^{\uparrow M}} = \underline{e_R \triangleright \phi} \triangleright \mathrm{pu}_M = e_S \triangleright \mathrm{pu}_M = e_{M^S} \quad .$$

Now check the composition law. For any $p^{:M^R}$ and $q^{:M^R}$:

$$\begin{aligned}
\text{expect to equal } (p \underset{M^R}{\oplus} q) \triangleright \phi^{\uparrow M}: \quad & (p \triangleright \phi^{\uparrow M}) \underset{M^S}{\oplus} (q \triangleright \phi^{\uparrow M}) \\
& = (p \triangleright \underline{\phi^{\uparrow M}) \triangleright \mathrm{flm}_M}\big(u \to q \triangleright \phi^{\uparrow M} \triangleright (v \to u \oplus_S v)^{\uparrow M}\big) \\
\text{naturality of } \mathrm{flm}_M: \quad & = p \triangleright \mathrm{flm}_M\big(u \to q \triangleright (\underline{\phi \fatsemi} (v \to \phi(u) \oplus_S v))^{\uparrow M}\big) \\
\text{compute composition}: \quad & = p \triangleright \mathrm{flm}_M\big(u \to q \triangleright (v \to \underline{\phi(u) \oplus_S \phi(v)})^{\uparrow M}\big) \\
\text{composition law of } \phi: \quad & = p \triangleright \mathrm{flm}_M\big(u \to q \triangleright (v \to \phi(u \oplus_R v))^{\uparrow M}\big) \\
\text{move } \phi^{\uparrow M} \text{ outside}: \quad & = p \triangleright \mathrm{flm}_M\big(u \to q \triangleright (v \to u \oplus_R v)^{\uparrow M} \fatsemi \phi^{\uparrow M}\big) \\
\text{naturality of } \mathrm{flm}_M: \quad & = \underline{p \triangleright \mathrm{flm}_M\big(u \to q \triangleright (v \to u \oplus_R v)^{\uparrow M}\big)} \fatsemi \phi^{\uparrow M} \\
& = (p \underset{M^R}{\oplus} q) \triangleright \phi^{\uparrow M} \quad .
\end{aligned}$$

Exercise 12.3.9.14 The empty elements of the monoids M^R and N^R are, by definition, are $\mathrm{pu}_M(e_R)$ and $\mathrm{pu}_N(e_R)$ respectively. The binary operations of M^R and N^R are:

$$p \underset{M^R}{\oplus} q \stackrel{\text{def}}{=} p \triangleright \mathrm{flm}_M\big(u^{:R} \to q \triangleright (v^{:R} \to u \oplus_R v)^{\uparrow M}\big) \quad ,$$
$$p \underset{N^R}{\oplus} q \stackrel{\text{def}}{=} p \triangleright \mathrm{flm}_N\big(u^{:R} \to q \triangleright (v^{:R} \to u \oplus_R v)^{\uparrow N}\big) \quad .$$

To verify the properties of the monoid morphism, begin with the identity law:

$$\text{expect to equal } e_{N^R}: \quad \underline{e_{M^R}} \triangleright \phi = \underline{e_R \triangleright \mathrm{pu}_M} \triangleright \phi = e_R \triangleright \mathrm{pu}_N = e_{N^R} \quad .$$

Now check the composition law. For any $p^{:M^R}$ and $q^{:M^R}$:

$$\begin{aligned}
\text{expect to equal } (p \underset{M^R}{\oplus} q) \triangleright \phi: \quad & (p \triangleright \phi) \underset{N^R}{\oplus} (q \triangleright \phi) = (p \triangleright \phi) \triangleright \mathrm{flm}_N\big(u \to q \triangleright \underline{\phi \triangleright (v \to u \oplus_R v)^{\uparrow N}}\big) \\
\text{naturality of } \phi: \quad & = p \triangleright \underline{\phi \fatsemi \big(u \to q \triangleright (v \to u \oplus_R v)^{\uparrow M} \fatsemi \phi\big)^{\uparrow N}} \fatsemi \mathrm{ftn}_N \\
\text{naturality of } \phi: \quad & = p \triangleright \big(u \to q \triangleright (v \to u \oplus_R v)^{\uparrow M}\big)^{\uparrow M} \fatsemi \underline{\phi \fatsemi \phi^{\uparrow N} \fatsemi \mathrm{ftn}_N} \\
\text{composition law of } \phi: \quad & = p \triangleright \big(u \to q \triangleright (v \to u \oplus_R v)^{\uparrow M}\big)^{\uparrow M} \underline{\fatsemi \mathrm{ftn}_M} \fatsemi \phi \\
\text{definition of } \mathrm{ftn}_M: \quad & = p \triangleright \mathrm{flm}_M\big(u \to q \triangleright (v \to u \oplus_R v)^{\uparrow M}\big) \fatsemi \phi = (p \underset{M^R}{\oplus} q) \triangleright \phi \quad .
\end{aligned}$$

Chapter 14

Exercise 14.9.5.4 The identity law of monad morphisms is:

$$\mathrm{pu}_{\mathrm{Id}} \fatsemi \phi = \mathrm{pu}_M \quad .$$

Since $\mathrm{pu}_{\mathrm{Id}} = \mathrm{id}$, we get $\phi = \mathrm{pu}_M$. So, there can be no other monad morphisms $\mathrm{Id} \rightsquigarrow M$.

Exercise 14.9.5.5 The identity law of monad morphisms says that $\text{pu}_{\mathbb{1}} \fatsemi \phi = \text{pu}_M$. However, $\text{pu}_{\mathbb{1}} = _^{:A} \to 1$ and ignores its argument. So, pu_M is also a function that ignores its argument. By the monad M's left identity law, $\text{pu}_M \fatsemi \text{ftn}_M = \text{id}^{:M^A \to M^A}$. So, the identity function $\text{id} : M^A \to M^A$ ignores its argument. It always returns the same value of type M^A, say m_0. It follows that the type M^A has only one distinct value, namely m_0. So $M^A \cong \mathbb{1}$. This argument holds for each type A.

Exercise 14.9.5.6 **(a)** For any monad morphism $\phi : M \rightsquigarrow N$, the monadic naturality law must hold:

$$\text{dbl} \fatsemi \phi \stackrel{?}{=} \phi \fatsemi \text{dbl} \quad .$$

The left-hand side applied to an arbitrary value $m^{:M^A}$ is:

$$\begin{aligned}
& m \triangleright \text{dbl} \fatsemi \phi = m \triangleright (_ \to m)^{\uparrow M} \fatsemi \underline{\text{ftn}_M \fatsemi \phi} \\
\text{monad morphism law of } \phi : \quad & = m \triangleright \underline{(_ \to m)^{\uparrow M} \fatsemi \phi^{\uparrow M}} \fatsemi \phi \fatsemi \text{ftn}_M \\
\text{compute composition} : \quad & = m \triangleright \underline{(_ \to m \triangleright \phi)^{\uparrow M} \fatsemi \phi} \fatsemi \text{ftn}_M \\
\text{naturality of } \phi : \quad & = m \triangleright \phi \fatsemi (_ \to m \triangleright \phi)^{\uparrow M} \fatsemi \text{ftn}_M \quad .
\end{aligned}$$

The right-hand side applied to m is:

$$\begin{aligned}
& m \triangleright \phi \fatsemi \text{dbl} = m \triangleright \phi \triangleright \text{dbl} = m \triangleright \phi \triangleright (_ \to m \triangleright \phi)^{\uparrow M} \fatsemi \text{ftn}_M \\
& = m \triangleright \phi \fatsemi (_ \to m \triangleright \phi)^{\uparrow M} \fatsemi \text{ftn}_M \quad .
\end{aligned}$$

Both sides are now equal.

(b) The identity law holds:

$$a \triangleright \text{pu}_M \fatsemi \text{dbl} = a \triangleright \text{pu}_M \triangleright \text{flm}_M (_ \to a \triangleright \text{pu}_M) = a \triangleright (_ \to a \triangleright \text{pu}_M) = a \triangleright \text{pu}_M \quad .$$

The composition law:

$$\text{dbl}^{\uparrow M} \fatsemi \text{dbl} \fatsemi \text{ftn}_M \stackrel{?}{=} \text{ftn}_M \fatsemi \text{dbl} \quad .$$

Simplify the left-hand side, applying to an arbitrary value $p : M^{M^A}$.

$$\begin{aligned}
\text{left-hand side} : \quad & p \triangleright \text{dbl}^{\uparrow M} \fatsemi \text{dbl} \fatsemi \text{ftn}_M = p \triangleright \text{dbl}^{\uparrow M} \triangleright (_ \to p \triangleright \text{dbl}^{\uparrow M})^{\uparrow M} \fatsemi \underline{\text{ftn}_M \fatsemi \text{ftn}_M} \\
\text{associativity law of } M : \quad & = p \triangleright \text{dbl}^{\uparrow M} \fatsemi (_ \to p \triangleright \text{dbl}^{\uparrow M})^{\uparrow M} \fatsemi \text{ftn}_M^{\uparrow M} \fatsemi \text{ftn}_M \\
& = p \triangleright \big(\text{dbl} \fatsemi (_ \to p \triangleright \text{dbl}^{\uparrow M}) \fatsemi \text{ftn})\big)^{\uparrow M} \fatsemi \text{ftn}_M = p \triangleright \big(_ \to p \triangleright \text{dbl}^{\uparrow M} \fatsemi \text{ftn}_M)\big)^{\uparrow M} \fatsemi \text{ftn}_M \\
& = p \triangleright \text{flm}_M \big(_ \to p \triangleright \text{flm}_M (\text{dbl})\big) \quad .
\end{aligned}$$

The right-hand side:

$$\begin{aligned}
\text{right-hand side} : \quad & p \triangleright \text{ftn}_M \fatsemi \text{dbl} = p \triangleright \underline{\text{ftn}_M \fatsemi (_ \to p \triangleright \text{ftn}_M)^{\uparrow M}} \fatsemi \text{ftn}_M \\
\text{naturality of } \text{ftn}_M : \quad & = p \triangleright (_ \to p \triangleright \text{ftn}_M)^{\uparrow M \uparrow M} \fatsemi \underline{\text{ftn}_M \fatsemi \text{ftn}_M} \\
\text{associativity law of } M : \quad & = p \triangleright (_ \to p \triangleright \text{ftn}_M)^{\uparrow M \uparrow M} \fatsemi \text{ftn}_M^{\uparrow M} \fatsemi \text{ftn}_M \\
& = p \triangleright \big((_ \to p \triangleright \text{ftn}_M)^{\uparrow M} \fatsemi \text{ftn}_M\big)^{\uparrow M} \fatsemi \text{ftn}_M \\
& = p \triangleright \text{flm}_M \big(\text{flm}_M (_ \to p \triangleright \text{ftn}_M)\big) \quad .
\end{aligned}$$

The simplification is stuck — we cannot make the two sides coincide; no law can be applied to simplify further.

Choosing specific values $p = \text{pu}_M(m)$ or $p = \text{pu}_M^{\uparrow M}(m)$ does not help to obtain a counter-example: the laws hold for those specific choices.

To obtain a counter-example, consider a specific non-commutative monad, such as `List`. For the `List` monad, the `double` function repeats the entire list as many times as elements in the list:

```
scala> double(List(1, 2))
res0: List[Int] = List(1, 2, 1, 2)

scala> val ll = List(List(1, 2), List(3))
ll: List[List[Int]] = List(List(1, 2), List(3))

scala> double(ll.flatten)
res1: List[Int] = List(1, 2, 3, 1, 2, 3, 1, 2, 3)

scala> double(ll.map(double))
res2: List[List[Int]] = List(List(1, 2, 1, 2), List(3), List(1, 2, 1, 2), List(3))

scala> double(ll.map(double)).flatten
res3: List[Int] = List(1, 2, 1, 2, 3, 1, 2, 1, 2, 3)
```

The composition law does not hold.

Exercise 14.9.5.12 The identity law does not hold:

$$\text{expect to equal } a \triangleright \mathrm{pu}_{\mathrm{Search}}: \quad a \triangleright \mathrm{pu}_{\mathrm{Sel}} \fatsemi \mathrm{finder} = (_^{:A \to \mathbb{1}+P} \to a) \triangleright \mathrm{finder}$$

$$= p \to p(a) \triangleright \begin{array}{c|c|cc} & & \mathbb{1} & A \\ \hline & \mathbb{1} & \mathrm{id} & \mathbb{0} \\ & P & \mathbb{0} & a \end{array} = p \to a \triangleright p \fatsemi (_ \to a)^{\uparrow \mathrm{Opt}} \neq p \to \mathbb{0} + a \quad .$$

The result should have been $\mathrm{pu}_{\mathrm{Search}}(a) = _ \to \mathrm{pu}_{\mathrm{Opt}}(a) = _ \to \mathbb{0} + a$.

Exercise 14.9.5.13 To verify the identity law:

$$\text{expect to equal } a \triangleright \mathrm{pu}_{\mathrm{Cont}}: \quad a \triangleright \mathrm{pu}_{\mathrm{Sel}} \fatsemi \mathrm{scc} = f \to f(f \triangleright (\underline{a \triangleright \mathrm{pu}_{\mathrm{Sel}}}))$$

$$\text{use definition of } \mathrm{pu}_{\mathrm{Sel}}: \quad = f \to f(f \triangleright (_ \to a)) = f \to f(a) = a \triangleright \mathrm{pu}_{\mathrm{Cont}} \quad .$$

To verify the composition law, we begin with the flipped Kleisli formulation of the two monads. The flipped Kleisli functions have types $(B \to P) \to A \to P$ (for `Cont`) and $(B \to P) \to A \to B$ (for `Sel`).

$$f^{:(B \to P) \to A \to P} \tilde{\diamond}_{\mathrm{Cont}} g^{:(C \to P) \to B \to P} \stackrel{\mathrm{def}}{=} h^{:C \to P} \to h \triangleright g \triangleright f = g \fatsemi f \quad ,$$
$$f^{:(B \to P) \to A \to B} \tilde{\diamond}_{\mathrm{Sel}} g^{:(C \to P) \to B \to C} \stackrel{\mathrm{def}}{=} h^{:C \to P} \to f(g(h) \fatsemi h) \fatsemi g(h) \quad .$$

We need to check that `Sel`'s Kleisli composition ($\tilde{\diamond}_{\mathrm{Sel}}$) is mapped to `Cont`'s Kleisli composition ($\tilde{\diamond}_{\mathrm{Cont}}$). First, we need to modify `scc` so that it works on the flipped Kleisli functions:

$$\tilde{\mathrm{scc}}^{A,B} : ((B \to P) \to A \to B) \to (B \to P) \to A \to P \quad , \quad \tilde{\mathrm{scc}} = c^{:((B \to P) \to A \to B)} \to k^{:B \to P} \to c(k) \fatsemi k \quad .$$

Note that the implementation of `scc` is uniquely determined by its type signature, and so is the implementation of $\tilde{\mathrm{scc}}$. So, we can spare ourselves the effort of translating scc into $\tilde{\mathrm{scc}}$.

Now it remains to show that for any $f^{:(B \to P) \to A \to B}$ and $g^{:(C \to P) \to B \to C}$ the law holds:

$$(f \triangleright \tilde{\mathrm{scc}}^{A,B}) \tilde{\diamond}_{\mathrm{Cont}} (g \triangleright \tilde{\mathrm{scc}}^{B,C}) = (f \tilde{\diamond}_{\mathrm{Sel}} g) \triangleright \tilde{\mathrm{scc}}^{A,C} \quad .$$

Rewrite the two sides of the law separately:

$$\text{left-hand side}: \quad (f \triangleright \tilde{\mathrm{scc}}) \tilde{\diamond}_{\mathrm{Cont}} (g \triangleright \tilde{\mathrm{scc}}) = (g \triangleright \tilde{\mathrm{scc}}) \fatsemi (f \triangleright \tilde{\mathrm{scc}}) = (k \to g(k) \fatsemi k) \fatsemi (h \to f(h) \fatsemi h)$$
$$= k \to f(g(k) \fatsemi k) \fatsemi g(k) \fatsemi k \quad ,$$

$$\text{right-hand side}: \quad (f \tilde{\diamond}_{\mathrm{Sel}} g) \triangleright \tilde{\mathrm{scc}} = \big(h \to f(g(h) \fatsemi h) \fatsemi g(h)\big) \triangleright \tilde{\mathrm{scc}} = k \to \big(h \to f(g(h) \fatsemi h) \fatsemi g(h)\big)(k) \fatsemi k$$
$$= k \to f(g(k) \fatsemi k) \fatsemi g(k) \fatsemi k \quad .$$

Both sides are now equal.

Exercise 14.9.5.15 The operations pu_N and ftn_N are defined by:

$$\mathrm{pu}_N \stackrel{\text{def}}{=} \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \end{array} \quad , \qquad \mathrm{ftn}_N \stackrel{\text{def}}{=} \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ M^A & \mathbb{0} & \mathrm{id} \\ M^{N^A} & \mathbb{0} & \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M \end{array} \quad .$$

To verify the identity law:

$$\text{expect to equal } \mathrm{pu}_M : \quad \mathrm{pu}_N \fatsemi \gamma_M = \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \end{array} \fatsemi \begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \\ M^A & \mathrm{id} \end{array} = \mathrm{pu}_M \quad .$$

To verify the composition law, transform separately the two sides of the law:

$$\text{left-hand side : } \quad \gamma_M \fatsemi \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M = \begin{array}{c||c} & M^{N^A} \\ \hline N^A & \mathrm{pu}_M \\ M^{N^A} & \mathrm{id} \end{array} \fatsemi \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M = \begin{array}{c||c} & M^A \\ \hline N^A & \underline{\mathrm{pu}_M \fatsemi \gamma_M^{\uparrow M}} \fatsemi \mathrm{ftn}_M \\ M^{N^A} & \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M \end{array}$$

$$= \begin{array}{c||c} & M^A \\ \hline N^A & \gamma_M \fatsemi \underline{\mathrm{pu}_M \fatsemi \mathrm{ftn}_M} \\ M^{N^A} & \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M \end{array} = \begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \\ M^A & \mathrm{id} \\ M^{N^A} & \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M \end{array} \quad ,$$

$$\text{right-hand side : } \quad \mathrm{ftn}_N \fatsemi \gamma_M = \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ M^A & \mathbb{0} & \mathrm{id} \\ M^{N^A} & \mathbb{0} & \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M \end{array} \fatsemi \begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \\ M^A & \mathrm{id} \end{array} = \begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \\ M^A & \mathrm{id} \\ M^{N^A} & \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M \end{array} \quad .$$

The two sides are now equal.

Exercise 14.9.5.16 The operations ftn_L and ftn_N are defined as usual for a free pointed monad:

$$\mathrm{ftn}_L \stackrel{\text{def}}{=} \begin{array}{c||c} & L^A \\ \hline L^A & \mathrm{id} \\ K^{L^A} & k \to \mathbb{0} + k \triangleright \gamma_K^{\uparrow K} \fatsemi \mathrm{ftn}_K \end{array} = \begin{array}{c||cc} & A & K^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ K^A & \mathbb{0} & \mathrm{id} \\ K^{L^A} & \mathbb{0} & \gamma_K^{\uparrow K} \fatsemi \mathrm{ftn}_K \end{array} \quad ,$$

$$\mathrm{ftn}_N \stackrel{\text{def}}{=} \begin{array}{c||c} & N^A \\ \hline N^A & \mathrm{id} \\ M^{N^A} & m \to \mathbb{0} + m \triangleright \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M \end{array} = \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ M^A & \mathbb{0} & \mathrm{id} \\ M^{N^A} & \mathbb{0} & \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M \end{array} \quad .$$

We define ψ by:

$$\psi = \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ K^A & \mathbb{0} & \phi \end{array} \quad .$$

To verify the identity law for ψ:

$$\mathrm{pu}_L \fatsemi \psi = \begin{array}{c||cc} & A & K^A \\ \hline A & \mathrm{id} & \mathbb{0} \end{array} \fatsemi \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ K^A & \mathbb{0} & \phi \end{array} = \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \end{array} = \mathrm{pu}_N \quad .$$

To verify the composition law for ψ, write both sides of the law separately:

$$\psi^{\uparrow L} \fatsemi \psi \fatsemi \mathrm{ftn}_N = \begin{array}{c||ccc} & A & M^A & K^{N^A} \\ \hline A & \mathrm{id} & \mathbb{0} & \mathbb{0} \\ K^A & \mathbb{0} & \phi & \mathbb{0} \\ K^{L^A} & \mathbb{0} & \mathbb{0} & \psi^{\uparrow K} \end{array} \fatsemi \begin{array}{c||ccc} & A & M^A & M^{N^A} \\ \hline A & \mathrm{id} & \mathbb{0} & \mathbb{0} \\ M^A & \mathbb{0} & \mathrm{id} & \mathbb{0} \\ K^{N^A} & \mathbb{0} & \mathbb{0} & \phi \end{array} \fatsemi \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ M^A & \mathbb{0} & \mathrm{id} \\ M^{N^A} & \mathbb{0} & \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M \end{array}$$

$$= \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ K^A & \mathbb{0} & \phi \\ K^{L^A} & \mathbb{0} & \psi^{\uparrow K} \fatsemi \phi \fatsemi \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M \end{array} \quad ,$$

$$\mathrm{ftn}_L \fatsemi \psi = \begin{array}{c||cc} & A & K^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ K^A & \mathbb{0} & \mathrm{id} \\ K^{L^A} & \mathbb{0} & \gamma_K^{\uparrow K} \fatsemi \mathrm{ftn}_K \end{array} \fatsemi \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ K^A & \mathbb{0} & \phi \end{array} = \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ K^A & \mathbb{0} & \phi \\ K^{L^A} & \mathbb{0} & \gamma_K^{\uparrow K} \fatsemi \mathrm{ftn}_K \fatsemi \phi \end{array} \quad .$$

It remains to show that:

$$\psi^{\uparrow K} \fatsemi \phi \fatsemi \gamma_M^{\uparrow M} \fatsemi \mathrm{ftn}_M \stackrel{?}{=} \gamma_K^{\uparrow K} \fatsemi \mathrm{ftn}_K \fatsemi \phi \quad .$$

The monad morphism law for ϕ gives:

$$\phi \fatsemi \phi^{\uparrow M} \fatsemi \mathrm{ftn}_M = \mathrm{ftn}_K \fatsemi \phi \quad .$$

It remains to show that:

$$\underline{\psi^{\uparrow K} \fatsemi \phi} \fatsemi \gamma_M^{\uparrow M} \stackrel{?}{=} \underline{\gamma_K^{\uparrow K} \fatsemi \phi} \fatsemi \phi^{\uparrow M} \quad .$$

$$\phi \fatsemi \psi^{\uparrow M} \fatsemi \gamma_M^{\uparrow M} = \phi \fatsemi (\psi \fatsemi \gamma_M)^{\uparrow M} \stackrel{?}{=} \phi \fatsemi \gamma_K^{\uparrow M} \fatsemi \phi^{\uparrow M} = \phi \fatsemi (\gamma_K \fatsemi \phi)^{\uparrow M} \quad .$$

It remains to show that $\psi \fatsemi \gamma_M = \gamma_K \fatsemi \phi$. For that, we use the identity law of ϕ:

$$\psi \fatsemi \gamma_M = \begin{array}{c||cc} & A & M^A \\ \hline A & \mathrm{id} & \mathbb{0} \\ K^A & \mathbb{0} & \phi \end{array} \fatsemi \begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \\ M^A & \mathrm{id} \end{array} = \begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \\ K^A & \phi \end{array} \quad ,$$

$$\gamma_K \fatsemi \phi = \begin{array}{c||c} & K^A \\ \hline A & \mathrm{pu}_K \\ K^A & \mathrm{id} \end{array} \fatsemi \phi = \begin{array}{c||c} & M^A \\ \hline A & \underline{\mathrm{pu}_K \fatsemi \phi} \\ K^A & \phi \end{array} = \begin{array}{c||c} & M^A \\ \hline A & \mathrm{pu}_M \\ K^A & \phi \end{array} \quad .$$

Exercise 14.9.5.17 **(a)** Verify the naturality law of ψ, assuming the naturality laws of ϕ and χ:

$$(f^{:A\to B})^{\uparrow(K+L)} \fatsemi \psi = \begin{array}{c||cc} & K^B & L^B \\ \hline K^A & f^{\uparrow K} & \mathbb{0} \\ L^A & \mathbb{0} & f^{\uparrow L} \end{array} \fatsemi \begin{array}{c||c} & M^B \\ \hline K^B & \phi \\ L^B & \chi \end{array} = \begin{array}{c||c} & M^B \\ \hline K^A & \underline{f^{\uparrow K} \fatsemi \phi} \\ L^A & \underline{f^{\uparrow L} \fatsemi \chi} \end{array} = \begin{array}{c||c} & M^B \\ \hline K^A & \phi \fatsemi f^{\uparrow M} \\ L^A & \chi \fatsemi f^{\uparrow M} \end{array} \quad ,$$

$$\psi \fatsemi f^{\uparrow M} = \begin{array}{c||c} & M^A \\ \hline K^A & \phi \\ L^A & \chi \end{array} \fatsemi f^{\uparrow M} = \begin{array}{c||c} & M^B \\ \hline K^A & \phi \fatsemi f^{\uparrow M} \\ L^A & \chi \fatsemi f^{\uparrow M} \end{array} \quad .$$

(b) Given the definitions of ϕ and χ, we can write ψ as:

$$\psi = \begin{array}{c||c} & M^A \\ \hline K^A & \phi \\ L^A & \chi \end{array} \quad .$$

Assuming the naturality law of ψ, verify the naturality law of ϕ by applying to an arbitrary value $k^{:K^A}$ and expressing $f^{\uparrow K}$ through $f^{\uparrow(K+L)}$:

$$\text{expect to equal } k \triangleright \phi \fatsemi f^{\uparrow M} : \quad k \triangleright f^{\uparrow K} \fatsemi \phi = k \triangleright f^{\uparrow K} \triangleright (k \to k + \mathbb{0}) \triangleright \psi = \big((k \triangleright f^{\uparrow K}) + \mathbb{0}\big) \triangleright \psi$$

$$= (k + \mathbb{0}) \triangleright \begin{array}{c||cc} & K^B & L^B \\ \hline K^A & f^{\uparrow K} & \mathbb{0} \\ L^A & \mathbb{0} & f^{\uparrow L} \end{array} \fatsemi \psi = (k + \mathbb{0}) \triangleright \underline{f^{\uparrow(K+L)} \fatsemi \psi}$$

$$\text{naturality law of } \psi : \quad = (\underline{k + \mathbb{0}}) \triangleright \psi \fatsemi f^{\uparrow M} = k \triangleright \underline{(x \to x + \mathbb{0}) \fatsemi \psi} \fatsemi f^{\uparrow M} = k \triangleright \phi \fatsemi f^{\uparrow M} \quad .$$

The proof of the naturality law of χ is analogous.

Exercise 14.9.5.18 **(a)** Choose the monads $K^A \stackrel{\text{def}}{=} R \to A$ and $M^A \stackrel{\text{def}}{=} S \to A$, where R, S are fixed (but arbitrary) types. Define $P^A \stackrel{\text{def}}{=} A + T_K^{M,A}$ and show that there exist no monad morphisms $\phi : M^A \to P^A$. If such ϕ exists, it must satisfy the identity law, $\text{pu}_M \fatsemi \phi = \text{pu}_P$. The type signature of ϕ is:

$$\phi : (R \to A) \to A + (S \to R \to A) \quad .$$

The only fully parametric implementation is:

$$\phi : f^{:R\to A} \to \mathbb{0} + (_^{:S} \to f) \quad ,$$

because we cannot produce values of type $A + \mathbb{0}$ since we cannot create values of the unknown type R. However, this implementation of ϕ does not satisfy the identity law since pu_P must return values of type $A + \mathbb{0}$.

(b) Given $\theta_K : K^A \to M^A$, we define:

$$\theta_L : L^A \to M^A \quad , \qquad \theta_L \stackrel{\text{def}}{=} \begin{array}{c||c} & M^A \\ \hline A & \text{pu}_M \\ K^A & \theta_K \end{array} \quad .$$

The proof becomes shorter if we express θ_L as:

$$\theta_L = \gamma_K \fatsemi \theta_K \quad , \qquad \gamma_K \stackrel{\text{def}}{=} \begin{array}{c||c} & K^A \\ \hline A & \text{pu}_K \\ K^A & \text{id} \end{array} \quad .$$

We already know that $\gamma_K : L \rightsquigarrow K$ is a monad morphism (Exercise 14.9.5.15). So, θ_L is a composition of monad morphisms.

(c) Choose $K^A \stackrel{\text{def}}{=} \mathbb{1}$ (the unit monad) and note that $L^A = A + \mathbb{1}$ is the standard `Option` monad (there are no other lawful monad implementations for the type constructor $L^A \stackrel{\text{def}}{=} A + \mathbb{1}$). Choose $M \stackrel{\text{def}}{=} L$, so that a monad morphism $\theta_L \stackrel{\text{def}}{=} \text{id}^{:L \rightsquigarrow M}$ exists. If it were possible to define a monad morphism $K \rightsquigarrow M$, we would have a monad morphism $\mathbb{1} \to \mathbb{1} + A$, but this is impossible: the only natural transformation of type $\mathbb{1} \to \mathbb{1} + A$ is $1 \to 1 + \mathbb{0}$, which does not satisfy the identity law of monad morphisms, $\text{pu}_K ⨾ \theta_K = \text{pu}_M$, since it never returns any values of type $\mathbb{0} + A$. (Generally, the existence of a monad morphism $\mathbb{1} \rightsquigarrow M$ means that $M = \mathbb{1}$.)

(d) By Exercise 14.9.5.17, we have a natural transformation $\theta_K : K^A \to A$ defined by:

$$\theta_K \stackrel{\text{def}}{=} (k \to \mathbb{0} + k) ⨾ \theta_L \quad .$$

The given runner θ_L is then expressed through θ_K as:

$$\theta_L = \begin{array}{c||c} & A \\ \hline A & \text{id} \\ K^A & \theta_K \end{array} \quad ,$$

because the natural transformation in the upper row of the matrix has the type signature $A \to A$ and so must be an identity function.

It remains to verify the monad morphism laws of $\theta_K : K \rightsquigarrow \text{Id}$. The identity law is:

$$\text{pu}_K ⨾ \theta_K \stackrel{?}{=} \text{pu}_{\text{Id}} = \text{id} \quad .$$

Since the function $p \stackrel{\text{def}}{=} \text{pu}_K ⨾ \theta_K$ has type signature $A \to A$ and is a natural transformation (as a composition of two natural transformations), that function must satisfy $p ⨾ f = f ⨾ p$ for any function $f^{:A \to B}$, and so p must be equal to the identity function.

The composition law of θ_K is an equation for functions of type $K^{K^A} \to A$:

$$\text{ftn}_K ⨾ \theta_K \stackrel{?}{=} \theta_K^{\uparrow K} ⨾ \theta_K ⨾ \text{ftn}_{\text{Id}} = \theta_K ⨾ \theta_K \quad .$$

Applied to an arbitrary value $k : K^{K^A}$, this law becomes:

$$k ▷ \text{ftn}_K ⨾ \theta_K \stackrel{?}{=} k ▷ \theta_K^{\uparrow K} ⨾ \theta_K \quad .$$

Since θ_K is defined via θ_L, we need to use the composition law of θ_L:

$$\text{ftn}_L ⨾ \theta_L \stackrel{!}{=} \theta_L^{\uparrow L} ⨾ \theta_L ⨾ \text{ftn}_{\text{Id}} = \theta_L^{\uparrow K} ⨾ \theta_L \quad .$$

Rewrite this law as:

$$\text{ftn}_L ⨾ \theta_L = \begin{array}{c||cc} & A & K^A \\ \hline A & \text{id} & \mathbb{0} \\ K^A & \mathbb{0} & \text{id} \\ K^{L^A} & \mathbb{0} & \gamma_K^{\uparrow K} ⨾ \text{ftn}_K \end{array} ⨾ \begin{array}{c||c} & A \\ \hline A & \text{id} \\ K^A & \theta_K \end{array} = \begin{array}{c||c} & A \\ \hline A & \text{id} \\ K^A & \theta_K \\ K^{L^A} & \gamma_K^{\uparrow K} ⨾ \text{ftn}_K ⨾ \theta_K \end{array} \quad ,$$

$$\theta_L^{\uparrow L} ⨾ \theta_L = \begin{array}{c||cc} & A & K^A \\ \hline A & \text{id} & \mathbb{0} \\ K^A & \mathbb{0} & \text{id} \\ K^{L^A} & \mathbb{0} & \theta_L^{\uparrow K} \end{array} ⨾ \begin{array}{c||c} & A \\ \hline A & \text{id} \\ K^A & \theta_K \end{array} = \begin{array}{c||c} & A \\ \hline A & \text{id} \\ K^A & \theta_K \\ K^{L^A} & \theta_L^{\uparrow K} ⨾ \theta_K \end{array} \quad .$$

The third rows of the matrices give the equation for functions of type $K^{L^A} \to A$:

$$\gamma_K^{\uparrow K} ⨾ \text{ftn}_K ⨾ \theta_K \overset{!}{=} \theta_L^{\uparrow K} ⨾ \theta_K \quad .$$

Apply both sides of this equation to an arbitrary value p of type $K^{\mathbb{0}+K^A}$, defined via an arbitrary value $k^{:K^{K^A}}$ as $p \overset{\text{def}}{=} k \triangleright (x \to \mathbb{0}+x)^{\uparrow K}$:

$$p \triangleright \gamma_K^{\uparrow K} ⨾ \text{ftn}_K ⨾ \theta_K \overset{!}{=} p \triangleright \theta_L^{\uparrow K} ⨾ \theta_K \quad ,$$

$$\text{or equivalently :} \quad k \triangleright (x \to \mathbb{0}+x)^{\uparrow K} ⨾ \gamma_K^{\uparrow K} ⨾ \text{ftn}_K ⨾ \theta_K \overset{!}{=} k \triangleright (x \to \mathbb{0}+x)^{\uparrow K} ⨾ \theta_L^{\uparrow K} ⨾ \theta_K \quad .$$

We compute some sub-expressions separately:

$$(x \to \mathbb{0}+x) ⨾ \gamma_K = (x \to \mathbb{0}+x) ⨾ \begin{array}{|c||c|} & K^A \\ \hline A & \text{pu}_K \\ K^A & \text{id} \end{array} = x \to x = \text{id} \quad ,$$

$$(x \to \mathbb{0}+x) ⨾ \theta_L = \theta_K \quad .$$

After these simplifications, the composition law of θ_L gives:

$$k \triangleright \text{ftn}_K ⨾ \theta_K \overset{!}{=} k \triangleright \theta_K^{\uparrow K} ⨾ \theta_K \quad .$$

This is the composition law of θ_K.

Exercise 14.9.5.19 Consider the monad morphism $\varepsilon : K \rightsquigarrow \mathbb{1}$ defined by $\varepsilon \overset{\text{def}}{=} _ \to 1$. (This is a monad morphism because all its laws reduce to the equation $1 = 1$.) By Exercise 14.9.5.16, we can implement a corresponding monad morphism between free pointed monads $\text{Id}+K \rightsquigarrow \text{Id}+\mathbb{1}$. Define δ as that monad morphism. The monad $\text{Id}+\mathbb{1}$ (the free pointed monad on $\mathbb{1}$) is the standard `Option` monad. So, we have implemented a monad morphism $\delta : A + K^A \to \text{Opt}^A$.

Exercise 14.9.5.21 **(a)** Consider the requirement of mapping $L^{L^A} \to L^A$ via a `flatten` function:

$$\text{ftn}_L : \mathbb{1} + L^A \times M^{L^{L^A}} \to \mathbb{1} + A \times M^{L^A} \quad , \qquad \text{ftn}_L = ???$$

The result value of this function cannot always be 1, or else it will fail the monad laws. This function must sometimes return a pair of type $A \times M^{L^A}$. Let us rewrite the type signature of `flatten` as

$$\text{ftn}_L : \mathbb{1} + (\mathbb{1} + A \times M^{L^A}) \times M^{L^{L^A}} \to \mathbb{1} + A \times M^{L^A} \quad .$$

Consider input values of the form:

$$x \overset{\text{def}}{=} \mathbb{0}^{:\mathbb{1}} + (1 + \mathbb{0}^{:A \times M^{L^A}}) \times m^{:M^{L^{L^A}}} \quad .$$

the result of evaluating $\text{ftn}_L(x)$ must be $1+\mathbb{0}$: a fully parametric function cannot extract values of type A from a value of type M^A with an unknown monad M. Note that the value $(1 + \mathbb{0}^{:A \times M^{L^A}})$ represents an empty list; let us denote that value by $\text{Nil}^{:L^A}$. So, we must have:

$$\text{ftn}_L(\mathbb{0} + \text{Nil} \times (...)) = \text{Nil} \quad .$$

Since the only way to extract list values is to run the M-effects, the implementation of ftn_L must proceed recursively, extracting list elements left to right. Now consider $M = \text{Id}$ (so we can simply omit the functor layers of M) and the nested list value:

$$z \overset{\text{def}}{=} [p, q, \text{Nil}, r, s, ...] \quad ,$$

where p, q, r, s are some sub-lists of type L^A with some type A. Then the value z has the form:

$$z = \mathbb{0} + p \times (\mathbb{0} + q \times (\mathbb{0} + \text{Nil} \times (\mathbb{0} + r \times (\mathbb{0} + s \times (1 + \mathbb{0}))))) \quad .$$

As we must have $\text{ftn}_L(\mathbb{0} + \text{Nil} \times t) = \text{Nil}$ for any $t^{:M^{L^{L^A}}}$, the recursive evaluation of $\text{ftn}_L(z)$ will give:

$$\text{ftn}_L(z) = ...\text{ftn}_L(\mathbb{0} + \text{Nil} \times (\mathbb{0} + r \times (\mathbb{0} + s \times (1 + \mathbb{0})))) = ...\text{Nil} \quad .$$

So, the result may be some function of p and q (which could be $p{+\!\!+}q$) but cannot depend on r and s. We have shown that ftn_L must ignore all sub-lists that follow an empty sub-list. A possible behavior of ftn_L is:

$$\text{ftn}_L([[1,2],[3,4],[],[5,6,7,8]]) = [1,2,3,4] \quad ,$$

instead of the full flattened list $[1,2,3,4,5,6,7,8]$. Regardless of how we implement ftn_L (and whether it satisfies the monad laws), the result of evaluating $\text{ftn}_L(z)$ cannot be the full concatenation $p{+\!\!+}q{+\!\!+}r{+\!\!+}s$ because the computation must ignore the sub-lists r and s.

(b) The code for the non-standard `flatten` function is:

```
def flatten[A](p: List[List[A]]): List[A] = p.takeWhile(_.nonEmpty).flatten  // ftn_L
```

The new `flatten` function gives the same results as `List`'s standard `flatten` method, except if one of the nested sub-lists is empty. Then the `flatten` function truncates the result after the first empty sub-list.

It turns out that this code fails the monad's associativity law of `flatten`. That law is an equality of functions $\text{ftn}_L \fatsemi \text{ftn}_L$ and $\text{ftn}_L^{\uparrow L} \fatsemi \text{ftn}_L$ of type `List[List[List[A]]] => List[A]`. The failure is found when a value p of type `List[List[List[A]]]` contains a nested empty list at the *second* nesting depth, following some non-empty lists. Here is an example that triggers the failure of the law:

```
val p: List[List[List[Int]]] = List(List(List(1, 2, 3)), List(List(4), Nil), List(List(5, 6)))
```

Here the nested list `List(List(4), Nil)` contains an empty list (`Nil`) after a non-empty list. Applying both sides of the law to `p`, we find that the two sides of the law are not equal:

```
scala> flatten(flatten(p))
res0: List[List[Int]] = List(1, 2, 3, 4)

scala> flatten(p.map(flatten))
res1: List[List[Int]] = List(1, 2, 3, 4, 5, 6)
```

(c) Try implementing the method $\text{flift} : M^A \to L^A$. The function `flift` must produce a value of type $L^A \cong \mathbb{1} + A \times M^{L^A}$. Since M is an arbitrary monad, we cannot extract a value of type A out of M^A while keeping the code fully parametric. So, we can implement `flift` only by defining $\text{flift} \stackrel{\text{def}}{=} _^{:M^A} \to \text{Nil}^{:L^A}$. However, that implementation loses information and fails the identity law:

$$\text{pu}_M \fatsemi \text{flift} = (_ \to \text{Nil}) \neq \text{pu}_L \quad .$$

Exercise 14.9.5.22 **(a)** Denote for brevity $\text{Cod}_F^{M,A} \stackrel{\text{def}}{=} C^A$. The naturality law for functions $c^{:C^A}$ says that for any $k^{:A \to F^X}$ and $q^{:X \to Y}$, we have:

$$(k^{:A \to F^X} \fatsemi q^{\uparrow F}) \triangleright c^Y = k \triangleright c^X \fatsemi q^{\uparrow M \uparrow F} \quad .$$

The flipped Kleisli method $\tilde{\text{pu}}_C$ is defined by:

$$\tilde{\text{pu}}_C : \forall X. (A \to F^X) \to A \to F^{M^X} \quad , \qquad \tilde{\text{pu}}_C \stackrel{\text{def}}{=} \forall X. k^{:A \to F^X} \to k \fatsemi \text{pu}_M^{\uparrow F} \quad .$$

To verify the left identity law:

$$\begin{aligned}
\text{expect to equal } g: \quad & \tilde{\text{pu}}_C \tilde{\diamond} g^{:\forall Y. (B \to F^Y) \to A \to F^{M^Y}} = \forall Z. k^{:B \to F^Z} \to \underline{(k \triangleright g^Z \triangleright \tilde{\text{pu}}_C^{M^Z})} \fatsemi \text{ftn}_M^{\uparrow F} \\
& = \forall Z. k^{:C \to F^Z} \to (g^Z(k) \fatsemi \underline{\text{pu}_M^{\uparrow F}) \fatsemi \text{ftn}_M^{\uparrow F}} \\
\text{left identity law of } M: \quad & = \forall Z. k^{:C \to F^Z} \to g^Z(k) = g \quad .
\end{aligned}$$

To verify the right identity law:

$$\begin{aligned}
\text{expect to equal } f: \quad & f^{:\forall X.\,(B\to F^X)\to A\to F^{M^X}} \,\tilde{\diamond}\, \tilde{\text{pu}}_C = \forall Z.\, k^{:B\to F^Z} \to \big(\underline{k \triangleright \tilde{\text{pu}}_C^Z} \triangleright f^{M^Z}\big) \,⨾\, \text{ftn}_M^{\uparrow F} \\
& = \forall Z.\, k^{:B\to F^Z} \to \big(\underline{(k \,⨾\, \text{pu}_M^{\uparrow F}) \triangleright f^{M^Z}}\big) \,⨾\, \text{ftn}_M^{\uparrow F} \\
\text{naturality law of } f: \quad & = \forall Z.\, k^{:B\to F^Z} \to \big(k \triangleright f^Z \,⨾\, \underline{\text{pu}_M^{\uparrow M\uparrow F}}\big) \,⨾\, \underline{\text{ftn}_M^{\uparrow F}} \\
\text{right identity law of } M: \quad & = \forall Z.\, k^{:B\to F^Z} \to k \triangleright f^Z = f \quad .
\end{aligned}$$

To verify the associativity law, write its two sides separately; omit all types for brevity:

$$\begin{aligned}
& (f \,\tilde{\diamond}\, g) \,\tilde{\diamond}\, h = l \to \big(l \triangleright h \triangleright \underline{(f \,\tilde{\diamond}\, g)}\big) \,⨾\, \text{ftn}_M^{\uparrow F} = l \to \big(\underline{l \triangleright h \triangleright (k} \to (k \triangleright g \triangleright f) \,⨾\, \text{ftn}_M^{\uparrow F})\big) \,⨾\, \text{ftn}_M^{\uparrow F} \\
& \quad = l \to \big((l \triangleright h \triangleright g \triangleright f) \,⨾\, \text{ftn}_M^{\uparrow F}\big) \,⨾\, \text{ftn}_M^{\uparrow F} = l \to l \triangleright h \,⨾\, g \,⨾\, f \,⨾\, \text{ftn}_M^{\uparrow F} \,⨾\, \text{ftn}_M^{\uparrow F} \quad , \\
& f \,\tilde{\diamond}\, (g \,\tilde{\diamond}\, h) = l \to \big(l \triangleright (g \,\tilde{\diamond}\, h) \triangleright f\big) \,⨾\, \text{ftn}_M^{\uparrow F} = l \to \big(l \triangleright (k \to (k \triangleright h \triangleright g) \,⨾\, \text{ftn}_M^{\uparrow F}) \triangleright f\big) \,⨾\, \text{ftn}_M^{\uparrow F} \\
& \quad = l \to \big(\big(\underline{(l \triangleright h \triangleright g) \,⨾\, \text{ftn}_M^{\uparrow F}) \triangleright f}\big) \,⨾\, \text{ftn}_M^{\uparrow F} = l \to l \triangleright h \triangleright g \triangleright f \,⨾\, \text{ftn}_M^{\uparrow M\uparrow F} \,⨾\, \text{ftn}_M^{\uparrow F} \quad .
\end{aligned}$$

In the last line, we have used the naturality law of f. The remaining difference between the two sides is:

$$\text{ftn}_M^{\uparrow F} \,⨾\, \text{ftn}_M^{\uparrow F} \stackrel{?}{=} \text{ftn}_M^{\uparrow M\uparrow F} \,⨾\, \text{ftn}_M^{\uparrow F} \quad ,$$

which follows from the associativity law of ftn_M.

(b) The `flatMap` method must have the type signature:

$$\text{flm}_L : \big((A \to X) \to M^X\big) \to (A \to (B \to X) \to M^X) \to (B \to X) \to M^X \quad .$$

Choose $M^A \stackrel{\text{def}}{=} \mathbb{1} + A$; now we need to implement the type signature:

$$\begin{aligned}
& \text{flm}_L : \big((A \to X) \to \mathbb{1} + X\big) \to (A \to (B \to X) \to \mathbb{1} + X) \to (B \to X) \to \mathbb{1} + X \quad , \\
& \text{flm}_L \stackrel{\text{def}}{=} p^{:(A\to X)\to \mathbb{1}+X} \to q^{:(A\to(B\to X)\to\mathbb{1}+X)} \to r^{:B\to X} \to ???^{:\mathbb{1}+X} \quad .
\end{aligned}$$

Can this function ever return a value of type $\mathbb{0} + X$? When we try filling out the typed hole $???^{:\mathbb{1}+X}$, we cannot apply the function r since we have no available values of type B. We could substitute r into the second curried argument of q, obtaining a function of type $A \to \mathbb{1} + X$. But we have no available values of type A. We also cannot apply the function p since its argument is of type $A \to X$, but we only have $A \to \mathbb{1} + X$, which is not guaranteed to return nonempty values. So, the only way of implementing `flatMap` via fully parametric code is to return the constant value $1 + \mathbb{0}^{:X}$. This would lose information and violate an identity law of monads.

F A humorous disclaimer

The following text is quoted in part from an anonymous online source ("Project Guten Tag") dating back at least to 1997. The original text is no longer available on the Internet.

WARRANTO LIMITENSIS; DISCLAMATANTUS DAMAGENSIS
Solus exceptus "Rectum Replacator Refundiens" describitus ecci,

1. Projectus (etque nunquam partum quis hic etext remitibus cum PROJECT GUTEN TAG™ identificator) disclamabat omni liabilitus tuus damagensis, pecuniensisque, includibantus pecunia legalitus, et

2. REMEDIA NEGLIGENTITIA NON HABET TUUS, WARRANTUS DESTRUCTIBUS CONTRACTUS NULLIBUS NI LIABILITUS SUMUS, INCLUTATIBUS NON LIMITATUS DESTRUCTIO DIRECTIBUS, CONSEQUENTIUS, PUNITIO, O INCIDENTUS, NON SUNT SI NOS NOTIFICAT VOBIS.

Sit discubriatus defectus en etextum sic entram diariam noventam recibidio, pecuniam tuum refundatorium receptorus posset, sic scribatis vendor. Sit veniabat medium physicalis, vobis idem reternat et replacator possit copius. Sit venitabat electronicabilis, sic viri datus chansus segundibus.

HIC ETEXT VENID "COMO-ASI". NIHIL WARRANTI NUNQUAM CLASSUM, EXPRESSITO NI IMPLICATO, LE MACCHEN COMO SI ETEXTO BENE SIT O IL MEDIO BENE SIT, INCLUTAT ET NON LIMITAT WARRANTI MERCATENSIS, APPROPRIATENSIS PURPOSEM.

Statuen varias non permitatent disclamabaris ni warranti implicatoren ni exclusioni limitatio damagaren consequentialis, ecco lo qua disclamatori exclusatorique non vobis applicant, et potat optia alia legali.

G GNU Free Documentation License

Version 1.2, November 2002

Copyright (c) 2000,2001,2002 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document free in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

G.0.0 Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

G.0.1 Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section G.0.2.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

G.0.2 Copying in quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

G.0.3 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections G.0.1 and G.0.2 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties — for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section G.0.2 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section G.0.3. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section G.0.3) to Preserve its Title (section G.0.0) will typically require changing the actual title.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) <year> <your name>. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being <list their titles>, with the Front-Cover Texts being <list>, and with the Back-Cover Texts being <list>.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Copyright

Copyright (c) 2000, 2001, 2002 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

List of Tables

1.1 Translating mathematics into code. 13
1.2 Nameless functions in programming languages. 20

2.1 Implementing mathematical induction. 48

4.1 Some notation for symbolic reasoning about code. 111

5.1 The correspondence between type constructions and $\mathcal{CH}$-propositions. 125
5.2 Proof rules for the constructive logic. 132
5.3 Examples of logical formulas that are true theorems in Boolean logic. 136
5.4 Examples of logical formulas that are *not* true in Boolean logic. 136
5.5 Logic identities with disjunction and conjunction, and the possible type equivalences. 142
5.6 Logical identities with implication, and the corresponding type equivalences and arithmetic identities. 148
5.7 Proof rules of constructive logic are true also in the Boolean logic. 167

6.1 Example translations of functor blocks into `map` methods. 182
6.2 Type constructions defining a functor L^A. 200
6.3 Recursive disjunctive types defined using type equations. 205
6.4 Type constructions defining a contrafunctor C^A. 207

7.1 This book's notation for the nine basic code constructions. 216

8.1 Type constructions producing the `Extractor` typeclass. 246
8.2 Type constructions producing the `Eq` typeclass. 249
8.3 Type constructions producing the `Semigroup` typeclass. 253
8.4 Type constructions producing the `Monoid` typeclass. 255
8.5 Type constructions producing the `Pointed` functor typeclass. 262
8.6 Type constructions producing the `Copointed` functor typeclass. 266
8.7 Type constructions producing the `Pointed` contrafunctor typeclass. 268
8.8 Structure of typeclass instance values for various inductive typeclasses. 284

11.1 Some typeclasses that follow the functorial pattern. 467

14.1 Correct and incorrect compositions of monads. 562
14.2 Known monad transformers for some monads. 565
14.3 Examples of monads stacked in different orders. 572
14.4 Known operations for some monads. 581

D.1 The six type constructions that may be used in fully parametric programs. 748
D.2 The nine code constructions of fully parametric programs. 749

List of Figures

3.1 The disjoint domain represented by the type `RootsOfQ`. 93

5.1 Proof tree for sequent (5.6). 133

8.1 Implementing a recursive instance of the `Monoid` typeclass via a `trait`. 256
8.2 Implementing type-safe computations with units of length and mass. 288

12.1 A counterexample violating the composition law of `traverse`. 506
12.2 Implementing `sequence` for square matrices. 519
12.3 Tests for the `sequence` method implemented in Figure 12.2. 520

16.1 Randomly chosen pages from books on Scala programming. 716
16.2 Randomly chosen pages from books on mathematics and physics. 717
16.3 Randomly chosen pages from books on applied functional type theory. 717

17.1 A programmer performs a derivation before writing Haskell code. 722

18.1 Mixing incompatible data types produces nonsensical results. 724
18.2 The Pyongyang method of error-free software engineering. 725

C.1 Axioms and derivation rules of the LJ algorithm. Each of the rules "(Left$\wedge_i$)" and "(Right$\vee_i$)" have two versions, with $i = 1$ or $i = 2$. 743
C.2 Proof transformers for the rules of the LJ algorithm. 744
C.3 Proof transformers for the four new rules of the LJT algorithm. 747

Index

F-algebra, 741
M-filterable contrafunctor, 343, 390, 406
M-filterable functor, 340, 343, 406
n queens problem, 349
▹-notation
    see "pipe notation", 218, 736
λ-calculus, 20

$\mathbb{2}$ (the `Boolean` type), 145
3-functor, 512
3-traversable 3-functor, 512
8 queens problem, 348

abstract syntax tree, 86, 359
accumulator argument, 36
aggregation, 2, 10, 39, 40, 415
algebra, 741
algebraic, 741
algebraic data types, 164, 740, 741
anonymous function
    see "nameless functions", 20, 740
applicative composition (⊙), 463
applicative functor
    commutative, 430
applicative functors, 275, 713
applicative morphism, 462, 501
applicative naturality law
    of `sequence`, 507
    of `traverse`, 501
applied functional type theory, 713, 722
argument list, 51, 231
assembly language, 19
associativity law
    of addition, 37
    of applicative composition, 463
    of `flatMap`, 371
    of `flatMap` for `Option`, 317
    of function composition, 103
    of Kleisli composition, 379
    of `map2`, 427
    of `merge`, 222
    of monoids, 239
    of `prod` for `ListT`, 652
    of semigroups, 236
    of splittable functors, 709
    of `zip`, 429

backward composition, 102, 104, 736
Bartosz Milewski, 773
bifunctor, 197, 481, 509
binary search, 50
binary tree, 83
bitraversable bifunctor, 509
Boolean logic, 135
bound type parameter, 537
bound variable, 7, 59, 215, 526
Brent Yorgey, 754

callback, 366
cardinality, 145
Cartesian product, 32, 145
case class, 63
`case` expression, 22
catamorphism, 557
category of monads, 402, 587
    initial object, 702
    terminal object, 702
category theory, 341
    category, 341
    functor, *341*, 402, 463
    in functional programming, 343, 727
    morphism, 341
    object, 341
Church encoding, 554
closure, 98
co-induction, 487
co-inductive typeclass, 285
co-pointed bifunctor, 265
co-pointed functor, 262
    in the category of monads, 588
co-product type
    see "disjunctive type", 69, 739
co-product types, 123
code inference, 120, 134, 374, 748
code notation, 215, 739
Collatz sequence, 55
commutative diagram, 174, 198
commutativity law
    of bifunctors, 198, 754, 767
    of profunctors, 754

of `zip`, 430
companion object, 232
compatibility law
of `deflate` and `flatten`, 698
of `extract` and `zip`, 437
composition law
of M-filterable contrafunctors, 390
of `ap`, 465
of applicative morphisms, 462, 501
in the point-free style, 503
of `bisequence`, 509
of contrafunctors, 187
of `filter`, 293
of filterable contrafunctor, 320
of `flatMap`, 379
of foreign monad runners, 586
of functors, 172
of `liftOpt`, 311
of `liftOpt` for contrafunctors, 325
of monad morphisms, 402
of monad runners, 400
of monad transformer's lift, 583
of pair product, 220
of `sequence`, 507
of `sequence3`, 513
of `traverse`, 505
computation expressions (F#), 739
Confucius, 300
conjunction (in logic), 95
conjunctive normal form, 350
constant contrafunctor, 206
constant function, 102, 215
constant functor, 200, 286
example of use, 287
constructive logic, 132, 169
continuation-passing, 48, *366*
contrafunctor, 187, 190, 739
contramonads, 397
contravariant functor
see "contrafunctor", 190
contravariant position, 191
corpus, 415
covariant position, 191
curried arguments, 98
curried function, 98
Curry-Howard correspondence, 138, 725
proof transformer, 745
`curryhoward` library, 134, 415, 570, 746
currying, 100, 665

Dan Doel, 557
data transformation, 10, 24
Day convolution, 591, 705
decidable logic, 154
default value, 37
dependency injection, 362
dependent object type (DOT) calculus, 713
dependent type, 153, 228
derivation rule, 130
destructuring, 22
dictionary, 19
dinatural transformation, 751
dinaturality law, 751
disjoint union
see "labeled union", 93
disjunction (in logic), 95
disjunctive functions, 216, 734, 737
disjunctive normal form, 350
disjunctive type, 69, 739
in matrix notation, 144, 216
distributive law
of `comb`, 658
of `concat`, 395
`do`-notation (Haskell), 739
domain of a function, 4
domain-specific language (DSL), 522
dot product
of two matrices, 353
of vector and matrix, 353
DSL program, 522
dynamic programming, 54, 57

eager collection, 59
eager value, 59
effect constructor, 528, 531
effect runner, 528, 531
eight code constructions, 129, 215
embedded DSL, 522
embedded `if`, 70
empty case class, 66
endofunctor, 341
in the category of monads, 588
enumeration type, 88, 94
Euler product, 14
evidence value, 269, 286
exception, 58, 70, 77, 166
exercises, 14, 32, 55, 82, 84, 85, 91, 116, 127, 145, 162, 195, 211, 226, 276, 300, 313, 326, 335, 353, 369, 394, 404, 422, 468, 490, 513, 546, 702, 708
existential quantifier, 536
expanded form of a function, 103
use in recursive values, 245
exponent, 147

exponential-polynomial type, 164, 208, 255, 739
expression, 4
expression block, 6
extension methods, 233
extractor typeclass, 241

factorial function, 4
filterable
    contrafunctor, 319
    defined via `takeWhile`, 299, 319
    functor, 293
    type constructor, 291
first-order logic, 154
fixpoint of a functor, 552
fixpoint type, 277
"flipped Kleisli" technique, *381*, 388, 671, 704
fold fusion, 415
foldable bifunctor, 500
foldable functor, 470, 499
`for`-comprehensions (Python), 739
formal logic, 129
forward composition, 102, 104, 736
free monad, 392, 522, 533, 722
    encodings, 534
    monad laws, 532
    raw tree encoding, 535
free pointed functor, 386
free pointed monad, 386, 627
free variable, 7, 215, 769
fully parametric
    code, 129, 215, 748
    code constructions, 129, 749
    function, 101, 102, 173, 183
    profunctor, 755
    program, 748
    type constructions, 748
function as a value, 5, 100
function composition, 102
functional programming paradigm, 15
functor, 174
    higher-order, 552
    in category theory, *341*, 402, 463
    laws of, 186
    lifting notation, 196
functor block, 179, 290, 739
    source, 180, 290
functor co-product, 202
functor composition, 203, 204
functor exponential, 202, 203
functor product, 200, 314

Gabriella Gonzalez (known as Gabriel Gonzalez prior to 2021), 534
GADT, 183, 748
    see "unfunctor", 740
generalized algebraic data types, 183
generic functions, 102

higher-order function, 117
higher-order functor, 552
higher-order type function, 283

identity function, 102
identity functor, 199
identity laws
    of M-filterable contrafunctors, 390
    of `ap`, 465, 468
    of applicative composition, 463
    of applicative morphisms, 462, 501
    of `bisequence`, 509
    of contrafunctors, 187
    of `deflate`, 331
    of equality, *185*, 246
    of `filter`, 293
    of filterable contrafunctors, 320
    of foreign monad runners, 586
    of function composition, 103
    of functors, 171
    of `inflate`, 335
    of `liftOpt`, 309, 310
    of `liftOpt` for contrafunctors, 325
    of `map`, 160
    of `map2`, 428
    of monad morphisms, 401
    of monad runners, 400
    of monad transformer's lift, 583
    of monoids, 238
    of pair product, 220
    of `prod` for `ListT`, 652
    of `pure` and `flatten`, 378
    of relational lifting, 774
    of `sequence`, 507
    of `sequence3`, 513
    of `traverse`, 504
    of `zip`, 429
identity matrix, 222
identity morphism, 341
identity relation, 758
image of a function, 214
immutable value, 61
implication (in logic), 128, 135
implicit value, 231
inductive typeclass, 283
infinite data types, 487
infinite loop in type recursion, 78, 79, 278
infix syntax, 7, 9

information loss, 90, 91, 140, 141, 160, 171
interpreter, 87
intuitionistic propositional logic, 132
inverse function, 139, 214
isomorphic types, 139
iterator, 60

Janis Voigtländer, 773, 786
Joachim de Lataillade, 773
John de Goes, 289
John Reynolds, 748
jokes, 533, 588, 715, 724, 725, 809
Josiah Willard Gibbs, 721

Kan extension, 692
Kelley Robinson, 534
"kind projector" plugin, 259, 273, 572, 736
kind signature, 283
Kleisli
    category, 343, 401
    functions, 338, *378*, 739
    morphisms, 739
    prononciation of the name, 311
Kleisli composition, 378
    for `Option`, 311
    with function composition, 380
Kurt Gödel, 168

labeled union, 93, 146
lambda-function
    see "nameless function", 20, 740
law of de Morgan, 157
law of excluded middle, 169
lawful functor, 186
lazy collection, 60, 444
lazy value, 59
left inverse, 214, 331
Lehmer's algorithm, 364
lifting, 74, 172, 196
Liskov substitution principle, 720
LJT algorithm, 134, 742, *747*
local scope, 6, 19, 59, 69
logical axiom, 130
logical implication, 124, 128
loop detection, 53

Machin's formula, 14
`map/reduce` programming style, *10*, 20, 469
mathematical induction, 13, 17, 33
    base case, 34
    inductive assumption, 34
    inductive step, 34
matrix notation, 144, 216, *737*
matrix transposition, 352, 411
Mauro Jaskelioff, 501, 581, 689
method, 739
method syntax, 7
monad morphism, *401*, 584
monad transformers, 564
    base lift, 589
    base runner laws, 589
    completeness, 664
    full definition, 589
    functor laws of runner, 589
    identity laws, 589
    "lift relation" technique, 577
    lifting law, 589
    monad laws, 589
    runner laws, 589
    stacking, 593
    universal base runner, 585
monadic naturality law, 590
    of base runner (`brun`), 588
    of `flift`, 587
    of foreign runner (`frun`), 588
    of lifted operations, 691
    of monoid composition, 656
    of `prod` for `ListT`, 653
    of `swap`, 605
monadic program, 362, 399
    `Monatron` library (MTL-style), 689
    MTL-style, 578
monads, 340, 346
    3-swap law, 679
    category of, 402, 587
    choice monad, 617
    codensity monad, 576, 592, *672*, 692
    commutative, 395
    composed codensity monad, 592, 705
    continuation monad (`Cont`), 366
    effectful operations, 578
    free pointed, 386, 627, 704
    generalized `Search` monad, 669
    interactive input-output monad, 546
    invalid `List` monad with truncation, 705
    lazy/eager evaluation monad (`Eval`), 365
    linear, 609, 615
    `List` monad, 374
    `List` monad with empty sub-lists, 395
    `Reader` monad, 360, 703
    rigid, 615
    runner, 362, 365, 367, *399*
    `Search` monad, 392, 617
    `Sel` (selector) monad, 392, 616, 676, 699
    stack of, 571, 593

State monad, 364, 703
Writer monad, 363, 703
Monatron library (MTL-style), 689
monoid, *238*, 430, 469
monoid morphism, *491*
monoidal naturality law, 491
morphism, 341
MTL-style monadic programs, 578
Monatron library, 689
mutability, 62

nameless function, 5, 740
nameless type-to-type function, 259
natural transformation, 336, 750
in the category of monads, 588
naturality law
combined with identity law, 310
in terms of relations, 760, 769
of co-pointed bifunctors, 265
of deflate, 305
of extract, 262
of filter, 293
of filter for contrafunctors, 326
of flatMap, 370, 371
of flatMap for Option, 318
of inflate for contrafunctors, 326
of liftOpt, 309, 313, 323
of merge, 222
of monad runners, 400
of pure, 257
of pure for contrafunctors, 265
of sequence, 505
of the function Δ, 220
of traverse, 501
nested recursive types, 516
Newton's method, 112
nine code constructions, 215, 748
non-degeneracy law
of splittable functors, 709
non-empty list, 81
nondegeneracy law
of base lift (blift), 589
of base runners (brun), 586, 589
of pointed and co-pointed functors, 263

object-oriented inheritance, 213, 289
object-oriented programming, 341, 720, 726
on-call value, 59, 149, 568
opaque type, 63
open problems, 710
operator syntax, 119
order of a function, 117

pair co-product of functions, *226*, 760
pair co-product of relations, 760
pair mapper of relations, 759
pair product of functions, 200, *220*, *737*, 760
pair product of relations, 760
palindrome integer, 56
paradigm of programming, 15
parametric code, 65
parametricity theorem, 339, 373
relation between values, 758
partial application, 99, 120
partial function, 58, 70, 112, 214, 228
partial function law
of filter, 293
reverse order in contrafunctors, 325
partial type-to-value function, 228, 740
pattern matching, 22
in matrix notation, 144
infallible, 58
pattern variables, 22
Paweł Szulc, 185
perfect numbers, 57
perfect-shaped tree, 84, 300, 516
is not a monad, 359
phantom type parameter, 287
Philip Wadler, 748
pipe notation, 151, 196, 218, 318, 736
operator precedence, 211, 219
planned exception, 77
point-free style, *218*, 370, 503
pointed contrafunctor, 264
pointed functor, *257*, 498
in the category of monads, 588
pointed type, 236
polymorphic function, 530, 531
polynomial functor, 202, 209, 740
recursive, 179, 499
polynomial functors, 174
polynomial type, 164
predicate, 6
procedure, 92
product type, 145, 740
product types, 123
profunctor, *207*, 213, 329, 406, 414, 451, 740
commutativity law, 754
pointed, 452
post-wedge property, 780
pushout property, 776
wedge relation, 759
proof (in logic), 122
proof by induction, 34
proof transformer, 745

proof tree, 743
proposition (in logic), 95
pullback relation, *759*
`pure` compatibility laws, 703
pure function, 61, 62
purity laws
  of monad transformer runners, 586
pushout property of profunctors, 776
pushout relation, *776*

Ralf Hinze, 517
recursion scheme, *204*, 356, 481
  using a profunctor, 329
recursive function, 35
  accumulator argument, 36
  proving laws for, 205
  termination, 245
recursive type equation, 277, 552
recursive types, 78
  infinite loop, 78
  unrolling trick, 334, 444, 639
    proof, 444
referential transparency, 61, 360, 748, 749
reflexivity law, *185*, 246, 247
relations between values
  see "value relation", 758
reverse relation, 758
Richard Bird, 510, 715
Richard Bornat, 129, 747
Riemann's zeta function, 14
right inverse, 214
right Kan extension, 692
rigid functors, 682
  open questions, 711
Robert C. Martin, 719
Ronald Backhouse, 748
rose tree, 84
Roy Dyckhoff, 164, 742
run-time error, 58
Rúnar Bjarnason, 534
runner, 87
Russell O'Connor, 275, 277

saturated application, 100
Scala method, 100
Scala's `Iterator` class, 60–62
search functor, 328
semigroup, 236
semimonads, 344, 346
  example of usage, 363
sequent (in logic), 122
  goal, 122
  premises, 122
serializer, 245
shadowed name, 59, 118
side effect, 62, 360
Simon Peyton Jones, 533
Simpson's rule, 18
six type constructions, 123, 208
solved examples, 10, 27, 41, 48, 69, 73, 81, 85, 88, 108, 111, 125, 141, 146, 147, 154, 175, 191, 210, 234, 269, 295, 302, 324, 330, 346, 355, 358, 367, 373, 402, 452, 471, 485, 502, 561, 572, 573, 684, 696, 751, 761, 763, 782, 786
`Spark`'s `RDD` data type, 431
splittable functors, 708
stack memory, 36
staggered factorial function, 14
stream, 45
strong dinaturality law, 777
  of Church encoding, 554
structural analysis, *206*, 241
structure functor of a typeclass, 283
subtyping, 189, 720, 748
  for disjunctive types, 190
  injective or surjective, 214
  subtype conversion function, *189*, 213
sum type
  see "disjunctive type", 146, 739
symbolic calculations, 105, 215
symmetry law of equality, 185
System F_ω (Haskell), 713

tagged union type
  see "disjunctive type", 739
tagless final, 722, 741
tail recursion, 36
theorems for free, 748
total function, 58, 228
trace of a matrix, 353
trampolines, 48
transitivity law of equality, 185
traversable functor, 470
▷-notation
  see "pipe notation", 151
"trivial" semigroup, 249
truth table, 135
tuples, 21, 740
  accessors, 21
  as function arguments, 101
  fields, 21
  nested, 22
  nested in pattern matching, 47
  parts, 21

turnstile (⊢) symbol, 122, 130
type alias, 41, 63, 124
type annotation, 73, 90, 734
type casts, 169
type checking, 111, 524
type constructor, 66
    contravariant, 190
    covariant, 190
type diagram, 174
type domain, 229
type equation, 208
type equivalence, 100, 139, 258
    accidental, 146, 162
    examples, 303, 308, 371
type error, 5, 21, 22, 31, 58, 64
type expression, 101
type inference, *110*, 111, 120, 748
type notation, 123, 125, 740
    operator precedence, 125
type parameter, 24, 65
type reflection, 184
type relation, *286*, 577, 758
    many-to-many, 286
    many-to-one, 287
type safety, 166, 524
type-level function, 203
type-to-type function, 228
type-to-value function, 228
typeclass, 227
    `Applicative`, 427
    `Bifunctor`, 272
    co-inductive, 285
    constraint, 227
    `Contrafunctor`, 272
    `Copointed`, 262
    `Eq`, 246
    evidence argument, 269
    evidence value, 286
    `Extractor`, 241
    `Functor`, 240
    functorial pattern, 467
    inductive, 283
    inheritance, 287
    instance value, 229
    `Monad`, 377
    `Monoid`, 238
    `Pointed`, 257
    `Semigroup`, 236
    `Semimonad`, 371
    `Show`, 245
    structure functor, 283
    `Zippable`, 426
typed hole, 158
types, 16
    equivalent, 139
    exponential-polynomial types, 164
    higher-kinded, 283
    isomorphic, 139
    pointed, 236
    polynomial type constructors, 174
    polynomial types, 164
    structural analysis, 241
    subtyping, 189
    universally quantified, 753

uncurried function, 98, 100
uncurrying, 100, 102, 665
undecidable logic, 154
unevaluated expression, 86
`unfold` function, 52, 57
unfunctor, 183, 188, 213, 229, 451, 531, 735, 740, 748
unit type, 66, 734
    named, 67, 94, 123, 126, 215
universal quantifier, 124, 128
unplanned exception, 77
unrolling trick for recursive types, 334, 444, 639
    proof, 444

value relation, 758
value-like behavior, 61, 360
variable, 16
variance annotation, 190
⊢ (turnstile) symbol, 122, 130
verifying laws with `scalacheck`, 240
void type, *73*, 78, 141, 149, 166, 190, 229, 278, 734
    in matrix notation, 217, 293

Wallis product, 11
wedge law
    of profunctors, 750, 773
wedge relation, *759*, 773
well-typed expression, 111
“wrapped unit” value, 258, 429

Yoneda identity
    contravariant, 551
    covariant, 550
    for functors, 552

This book is a pedagogical in-depth tutorial and reference on the theory of functional programming (FP) as practiced in the early 21st century. Starting from issues found in practical coding, the book builds up the theoretical intuition, knowledge, and techniques that programmers need for rigorous reasoning about types and code. Examples are given in Scala, but most of the material applies equally to other FP languages.

The book's topics include working with FP-style collections; reasoning about recursive functions and types; the Curry-Howard correspondence; laws, structural analysis, and code for functors, monads, and other typeclasses built upon exponential-polynomial data types; techniques of symbolic derivation and proof; free typeclass constructions; and parametricity theorems.

Long and difficult, yet boring explanations are logically developed in excruciating detail through 1804 Scala code snippets, 172 statements with step-by-step derivations, 99 diagrams, 214 solved examples with tested Scala code, and 267 exercises. Discussions further build upon each chapter's material.

Beginners in FP will find tutorials about the `map/reduce` programming style, type parameters, disjunctive types, and higher-order functions. For more advanced readers, the book shows the practical uses of the Curry-Howard correspondence and the parametricity theorems without unnecessary jargon; proves that all the standard monads (e.g., `List` or `State`) satisfy the monad laws; derives lawful instances of `Functor` and other typeclasses from types; shows that monad transformers need 18 laws; and explains the use of Yoneda identities for reasoning about the Church encoding and the free typeclasses.

Readers should have a working knowledge of programming; e.g., be able to write code that prints the number of distinct words in a sentence. The difficulty of this book's mathematical derivations is at the level of undergraduate calculus, similar to that of multiplying matrices or simplifying the expressions:

$$\frac{1}{x-2} - \frac{1}{x+2} \quad \text{and} \quad \frac{d}{dx}\left((x+1)f(x)e^{-x}\right) \quad .$$

Sergei Winitzki received a Ph.D. in theoretical physics. After a career in academic research, he currently works as a software engineer.