

The Science of Functional Programming

A Tutorial, with Examples in Scala

by Sergei Winitzki, Ph.D.

draft version, September 14, 2019

Published by **lulu.com** in the year 2019

Copyright © 2018-2019 by Sergei Winitzki.

Published and printed by **lulu.com**

ISBN 978-0-359-76877-6

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” (Appendix F).

A *Transparent* copy of the source code for the book (LyX, LaTeX, graphics files, and build scripts), together with a full-color hyperlinked PDF file, is available at <https://github.com/winitzki/sofp>. The source code is also included as a “file attachment” `sofp-src.tar.bz2` within the PDF file `sofp.pdf`. 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 pedagogical tutorial book presents the theoretical knowledge needed by practitioners of the functional programming paradigm. The main topics covered are: working with functional collections and recursion; the fundamental type and code constructions in functional programming; reasoning about types and code with the Curry-Howard correspondence; laws, structure theorems, and code derivation for functors, monads, and other important typeclasses; free type constructions; and recursive types. Detailed explanations are logically developed and accompanied by full derivations and worked examples tested in the Scala interpreter, as well as exercises. Readers should have a working knowledge of basic Scala; e.g. be able to write code that reads a small text file and prints the number of words in it. Readers should also know school-level mathematics; for example, should be able to simplify the expressions $\frac{1}{x-2} - \frac{1}{x+2}$ and $\frac{d}{dx} ((x+1) e^{-x})$.

Contents

Preface	1
Formatting conventions used in this book	1
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	11
1.4.1 Aggregation	11
1.4.2 Transformation	12
1.5 Summary	13
1.6 Exercises	14
1.6.1 Aggregation	14
1.6.2 Transformation	15
1.7 Discussion	15
1.7.1 Functional programming as a paradigm	15
1.7.2 Functional programming languages	16
1.7.3 The mathematical meaning of “variables”	16
1.7.4 Iteration without loops	17
1.7.5 Nameless functions in mathematical notation	18
1.7.6 Named and nameless expressions and their uses	19
1.7.7 Historical perspective on usage of 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 (Scala’s Maps) 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	34
2.2.2 Implementing functions by recursion	35
2.2.3 Tail recursion	36
2.2.4 Implementing general aggregation (<code>foldLeft</code>)	39
2.2.5 Solved examples: using <code>foldLeft</code>	41
2.2.6 Exercises: Using <code>foldLeft</code>	44

Contents

2.3	Converting a single value into a sequence	46
2.4	Transforming a sequence into another sequence	47
2.5	Summary	48
2.5.1	Solved examples	49
2.5.2	Exercises	55
2.6	Discussion	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	60
3	The logic of types. I. Disjunctive types	64
3.1	Scala's case classes	64
3.1.1	Tuple types with names	64
3.1.2	Case classes with type parameters	66
3.1.3	Tuples with one part and with zero parts	67
3.1.4	Pattern matching for case classes	68
3.2	Disjunctive types	68
3.2.1	Motivation and first examples	68
3.2.2	Solved examples: Pattern matching for disjunctive types	70
3.2.3	Standard disjunctive types: Option, Either, Try	73
3.3	Lists and trees: recursive disjunctive types	79
3.3.1	Lists	80
3.3.2	Tail recursion with List	81
3.3.3	Binary trees	84
3.3.4	Rose trees	85
3.3.5	Regular-shaped trees	85
3.3.6	Abstract syntax trees	87
3.4	Summary	89
3.4.1	Solved examples	89
3.4.2	Exercises	93
3.5	Discussion	94
3.5.1	Disjunctive types as mathematical sets	94
3.5.2	Disjunctive types in other programming languages	95
3.5.3	Disjunctions and conjunctions in formal logic	96
II	Intermediate level	98
4	The logic of types. II. Higher-order functions	99
4.1	Functions that return functions	99
4.1.1	Motivation and first examples	99
4.1.2	Curried and uncurried functions	100
4.1.3	Equivalence of curried and uncurried functions	101
4.2	Fully parametric functions	102
4.2.1	Examples. Function composition	104
4.2.2	Laws of function composition	105
4.2.3	Example: A function that violates parametricity	107
4.3	Symbolic calculations with nameless functions	108
4.3.1	Calculations with curried functions	108
4.3.2	Solved examples: Deriving a function's type from its code	110
4.4	Summary	113
4.4.1	Solved examples	113

4.4.2 Exercises	118
4.5 Discussion	119
4.5.1 Higher-order functions	119
4.5.2 Name shadowing and the scope of bound variables	119
4.5.3 Operator syntax for function applications	120
4.5.4 Deriving a function's code from its type signature	121
5 The logic of types. III. The Curry-Howard correspondence	123
5.1 Values computed by fully parametric functions	123
5.1.1 Motivation	123
5.1.2 Type notation and CH -propositions for standard type constructions	124
5.1.3 Solved examples: Type notation	127
5.1.4 Exercises: Type notation	130
5.2 The logic of CH -propositions	130
5.2.1 Motivation and first examples	130
5.2.2 Example: Failure of Boolean logic for type reasoning	132
5.2.3 The rules of proof for CH -propositions	133
5.2.4 Example: Proving a CH -proposition and deriving code	137
5.3 Solved examples: Equivalence of types	140
5.3.1 Logical identity does not correspond to type equivalence	140
5.3.2 Arithmetic identity corresponds to type equivalence	143
5.3.3 Type cardinalities and type equivalence	147
5.3.4 Type equivalence involving function types	149
5.4 Summary	155
5.4.1 Solved examples	156
5.4.2 Exercises	165
5.5 Discussion	166
5.5.1 Using the Curry-Howard correspondence for writing code	166
5.5.2 Implications for designing new programming languages	168
5.5.3 Uses of the void type	169
5.5.4 Relationship between Boolean logic and constructive logic	169
6 Functors, contrafunctors, and profunctors	172
6.1 Practical use	172
6.1.1 Motivation: Type constructors that wrap data	172
6.1.2 Example: Option and the identity law	173
6.1.3 Motivation for the composition law	174
6.1.4 Functors: definition and examples	175
6.1.5 Functor block expressions	179
6.1.6 Examples of non-functors	182
6.1.7 Contrafunctors	187
6.1.8 Subtyping, covariance, and contravariance	189
6.1.9 Solved examples: functors and contrafunctors	191
6.1.10 Exercises: functors and contrafunctors	195
6.2 Laws and structure	196
6.2.1 Reformulations of laws	196
6.2.2 Bifunctors	197
6.2.3 Type constructions for functors	199
6.2.4 Type constructions for contrafunctors	206
6.2.5 Solved examples: How to recognize functors and contrafunctors	209
6.3 Summary	211
6.3.1 Exercises: Functor and contrafunctor constructions	211

6.4	Discussion	212
6.4.1	Profunctors	212
6.4.2	Subtyping with injective or surjective conversion functions	213
7	Typeclasses and functions of types	215
7.1	Motivation and first examples	215
7.1.1	Constraining type parameters	215
7.1.2	Functions of types and values	215
7.1.3	Partial functions of types and values	216
7.2	Implementing typeclasses	217
7.2.1	Creating a partial function on types	217
7.2.2	Scala's <code>implicit</code> values	219
7.2.3	Implementing typeclasses by making instances <code>implicit</code>	220
7.2.4	Extension methods	221
7.2.5	Solved examples: Implementing typeclasses in practice	222
7.2.6	Typeclasses for type constructors	228
7.3	Deriving typeclass instances via structural analysis of types	229
7.3.1	Extractors	229
7.3.2	The <code>Eq</code> typeclass	234
7.3.3	Semigroups	236
7.3.4	Monoids	240
7.3.5	Pointed functors: motivation and laws	242
7.3.6	Pointed functors: structural analysis	246
7.3.7	Co-pointed functors	248
7.3.8	Pointed contrafunctors	251
7.4	Summary	252
7.4.1	Solved examples	253
7.4.2	Exercises	253
7.5	Discussion	254
7.5.1	Recursive types and recursive values	254
7.5.2	Inductive typeclasses and their properties	254
7.5.3	Inheritance and automatic conversions of typeclasses	255
7.5.4	Typeclasses with more than one type parameter (type relations)	255
8	Computations in functor blocks. I. Filterable functors	256
8.1	Slides	256
8.1.1	Computations within a functor context	256
8.1.2	Filterable functors: Intuitions I	256
8.1.3	Examples of filterable functors I	257
8.1.4	Filterable functors: Intuitions II	257
8.1.5	Examples of filterable functors II: Checking the laws	257
8.1.6	Worked examples I: Programming with filterables	258
8.1.7	Exercises I	258
8.1.8	Filterable functors: The laws in depth I	258
8.1.9	Filterable functors: Using <code>deflate</code>	259
8.1.10	Summary: The methods and the laws	261
8.1.11	Structure of filterable functors	262
8.1.12	* Worked examples II: Constructions of filterable functors I	262
8.1.13	Worked examples II: Constructions of filterable functors IV	263
8.1.14	* Exercises II	264
8.1.15	* Bonus slide I: Definition of filterable contrafunctors	264
8.1.16	* Bonus slide II: Structure of filterable contrafunctors	264

8.1.17	Addendum	265
8.2	Practical use	265
8.2.1	Discussion	265
8.3	Laws and structure	265
8.3.1	Discussion	265
9	Computations in functor blocks. II. Semimonads and monads	266
9.1	Slides, part 1	266
9.1.1	Computations within a functor context: Semimonads	266
9.1.2	How <code>flatMap</code> works with lists	266
9.1.3	What is <code>flatMap</code> doing with the data in a collection?	266
9.1.4	Worked examples I: List-like monads	267
9.1.5	Intuitions for pass/fail monads	267
9.1.6	Worked examples II: Pass/fail monads	268
9.1.7	Intuitions for tree-like monads	268
9.1.8	Worked examples III: Tree-like monads	268
9.1.9	Worked examples IV: Single-value monads	268
9.1.10	Deriving the types of single-value monads	269
9.1.11	Exercises I	269
9.2	Slides, part 2	270
9.2.1	Semimonad laws I: The intuitions	270
9.2.2	Semimonad laws II: The laws for <code>flatMap</code>	270
9.2.3	Semimonad laws III: The laws for <code>flatten</code>	270
9.2.4	Equivalence of a natural transformation and a “lifting”	271
9.2.5	Semimonad laws IV: Deriving the laws for <code>flatten</code>	271
9.2.6	Checking the associativity law for standard monads	272
9.2.7	Motivation for monads	272
9.2.8	The monad laws formulated in terms of <code>pure</code> and <code>flatten</code>	272
9.2.9	Formulating laws via Kleisli functions	273
9.2.10	* Motivation for categories and functors	273
9.2.11	* From Kleisli back to <code>flatMap</code>	274
9.2.12	Structure of semigroups and monoids	274
9.2.13	Structure of (semi)monads	274
9.2.14	Exercises II	275
9.2.15	Exercises II (continued)	275
9.2.16	Addendum: Miscellaneous remarks on monads	275
9.3	Practical use	276
9.3.1	Discussion	276
9.4	Laws and structure	276
9.4.1	Discussion	276
10	Applicative functors, contrafunctors, and profunctors	277
10.1	Slides, Part I	277
10.1.1	Motivation for applicative functors	277
10.1.2	Defining <code>map2</code> , <code>map3</code> , etc.	277
10.1.3	Practical examples of using <code>mapN</code>	278
10.1.4	Exercises I	278
10.2	Slides, Part II	278
10.2.1	Deriving the <code>ap</code> operation from <code>map2</code>	278
10.2.2	Deriving the <code>zip</code> operation from <code>map2</code>	279
10.2.3	* Equivalence of the operations <code>ap</code> and <code>zip</code>	279
10.2.4	Motivation for applicative laws. Naturality laws for <code>map2</code>	280

10.2.5	Associativity and identity laws for <code>map2</code>	280
10.2.6	Deriving the laws for <code>zip</code> : naturality law	280
10.2.7	Deriving the laws for <code>zip</code> : associativity law	281
10.2.8	Deriving the laws for <code>zip</code> : identity laws	281
10.2.9	Similarity between applicative laws and monoid laws	282
10.2.10	A third naturality law for <code>map2</code>	282
10.2.11	Applicative operation <code>ap</code> as a “lifting”	282
10.2.12	Deriving the category laws for (id_{\circ}, \odot)	283
10.2.13	Deriving the functor laws for <code>ap</code>	283
10.2.14	Constructions of applicative functors	284
10.2.15	All non-parameterized exp-poly types are monoids	284
10.2.16	Definition and constructions of applicative contrafunctors	284
10.2.17	Definition and laws of profunctors	285
10.2.18	Definition and constructions of applicative profunctors	285
10.2.19	Commutative applicative functors	286
10.2.20	Categorical overview of “regular” functor classes	286
10.2.21	Exercises	287
10.3	Practical use	287
10.3.1	Discussion	287
10.4	Laws and structure	287
11	Traversable functors and profunctors	288
11.1	Slides	288
11.1.1	Motivation for the <code>traverse</code> operation	288
11.1.2	Deriving the <code>sequence</code> operation	288
11.1.3	Polynomial functors are traversable	289
11.1.4	Motivation for the laws of the <code>traverse</code> operation	289
11.1.5	Formulation of the laws for <code>traverse</code>	289
11.1.6	Derivation of the laws for <code>sequence</code>	290
11.1.7	Constructions of traversable and bitraversable functors	290
11.1.8	Foldable functors: traversing with respect to a monoid	291
11.1.9	Traversable contrafunctors and profunctors are not useful	291
11.1.10	Examples of usage	291
11.1.11	Naturality with respect to applicative functor as parameter	292
11.1.12	Exercises	292
11.2	Discussion	292
III	Advanced level	293
12	“Free” type constructions	294
12.1	Slides	294
12.1.1	The interpreter pattern I. Expression trees	294
12.1.2	The interpreter pattern II. Variable binding	294
12.1.3	The interpreter pattern III. Type safety	294
12.1.4	The interpreter pattern IV. Cleaning up the DSL	295
12.1.5	The interpreter pattern V. Define Monad-like methods	295
12.1.6	The interpreter pattern VI. Refactoring to an abstract DSL	296
12.1.7	The interpreter pattern VII. Handling errors	296
12.1.8	The interpreter pattern VIII. Monadic DSLs: summary	296
12.1.9	Monad laws for DSL programs	297
12.1.10	Free constructions in mathematics: Example I	297

12.1.11 Free constructions in mathematics: Example II	297
12.1.12 Worked example I: Free semigroup	298
12.1.13 Worked example II: Free monoid	298
12.1.14 Mapping a free semigroup to different targets	298
12.1.15 Church encoding I: Motivation	299
12.1.16 Church encoding II: Disjunction types	299
12.1.17 Church encoding III: How it works	299
12.1.18 Worked example III: Free functor I	300
12.1.19 Encoding with an existential type: How it works	300
12.1.20 Worked example III: Free functor II	301
12.1.21 Church encoding IV: Recursive types and type constructors	301
12.1.22 Church encoding V: Type classes	301
12.1.23 Properties of free type constructions	302
12.1.24 Recipes for encoding free typeclass instances	302
12.1.25 Properties of inductive typeclasses	303
12.1.26 Worked example IV: Free contrafunctor	303
12.1.27 Worked example V: Free pointed functor	303
12.1.28 Worked example VI: Free filterable functor	304
12.1.29 Worked example VII: Free monad	304
12.1.30 Worked example VIII: Free applicative functor	304
12.1.31 Laws for free typeclass constructions	305
12.1.32 Combining the generating constructors in a free typeclass	305
12.1.33 Combining different free typeclasses	306
12.1.34 Exercises	306
12.1.35 Corrections	306
12.2 Discussion	306
13 Computations in functor blocks. III. Monad transformers	307
13.1 Slides	307
13.1.1 Computations within a functor context: Combining monads	307
13.1.2 Combining monadic effects I. Trial and error	307
13.1.3 Combining monadic effects II. Lifting into a larger monad	308
13.1.4 Laws for monad liftings I. Identity laws	308
13.1.5 Laws for monad liftings II. Simplifying the laws	308
13.1.6 Laws for monad liftings III. The naturality law	309
13.1.7 Monad transformers I: Motivation	309
13.1.8 Monad transformers II: The requirements	309
13.1.9 Monad transformers III: First examples	310
13.1.10 Monad transformers IV: The zoology of <i>ad hoc</i> methods	310
13.2 Practical use	311
13.3 Laws and structure	311
13.3.1 Laws of monad transformers	311
13.3.2 Examples of incorrect monad transformers	311
13.3.3 Examples of failure to define a generic monad transformer	312
13.3.4 Properties of monadic morphisms	313
13.3.5 Functor composition with transformed monads	316
13.3.6 Stacking two monads	317
13.3.7 Stacking any number of monads	319
13.4 Monad transformers via functor composition: General properties	320
13.4.1 Motivation for the <code>swap</code> function	320
13.4.2 Deriving the necessary laws for <code>swap</code>	322
13.4.3 Intuition behind the laws of <code>swap</code>	325

Contents

13.4.4	Deriving swap from flatten	326
13.4.5	Monad transformer identity law: Proofs	329
13.4.6	Monad transformer lifting laws: Proofs	330
13.4.7	Monad transformer runner laws: Proofs	331
13.4.8	Summary of results	334
13.5	Composed-inside transformers: Linear monads	335
13.5.1	Definitions of swap and flatten	335
13.5.2	Laws of swap	336
13.5.3	Composition of transformers for linear monads	341
13.6	Composed-outside transformers: Rigid monads	342
13.6.1	Rigid monad construction 1: choice	342
13.6.2	Rigid monad construction 2: composition	354
13.6.3	Rigid monad construction 3: product	359
13.6.4	Rigid monad construction 4: selector	359
13.6.5	Rigid functors	359
13.7	Recursive monad transformers	365
13.7.1	Transformer for the free monad FreeT	365
13.7.2	Transformer for the list monad ListT	365
13.8	Monad transformers for monad constructions	365
13.8.1	Product of monad transformers	365
13.8.2	Free pointed monad transformer	365
13.9	Irregular and incomplete monad transformers	365
13.9.1	The state monad transformer StateT	365
13.9.2	The continuation monad transformer ContT	365
13.9.3	The codensity monad transformer CodT	365
13.10	Summary and discussion	365
13.10.1	Exercises	365
14	Recursive types	367
14.1	Fixpoints and type recursion schemes	367
14.2	Row polymorphism and OO programming	367
14.3	Column polymorphism	367
14.4	Discussion	367
15	Co-inductive typeclasses. Comonads	368
15.1	Practical use	368
15.2	Laws and structure	368
15.3	Co-semigroups and co-monoids	368
15.4	Co-free constructions	368
15.5	Co-free comonads	368
15.6	Comonad transformers	368
15.7	Discussion	368
16	Irregular typeclasses	369
16.1	Distributive functors	369
16.2	Monoidal monads	369
16.3	Lenses and prisms	369
16.4	Discussion	369
IV	Discussions	370
17	Summary and outlook	371

18 “Applied functional type theory”: A proposal	372
18.1 AFTT is not covered by computer science curricula	372
18.2 AFTT is not category theory, type theory, or formal logic	373
19 Essay: Software engineers and software artisans	375
19.1 Engineering disciplines	375
19.2 Artisanship: Trades and crafts	375
19.3 Programmers today are artisans, not engineers	376
19.3.1 No requirement of formal study	376
19.3.2 No mathematical formalism guides software development	377
19.3.3 Programmers avoid academic terminology	378
19.4 Towards software engineering	378
19.5 Does software need engineers, or are artisans good enough?	380
20 Essay: Towards functional data engineering with Scala	381
20.1 Data is math	381
20.2 Functional programming is math	381
20.3 The power of abstraction	382
20.4 Scala is Java on math	383
20.5 Summary	383
V Appendixes	384
A Notations	385
A.1 Summary of notations for types and code	385
A.2 Detailed explanations	386
B Glossary of terms	391
B.1 On the current misuse of the term “algebra”	393
C The Curry-Howard correspondence	394
C.1 Slides	394
C.2 Intuitionistic propositional logic (IPL)	400
C.3 Example: The logic of types is not Boolean	400
C.4 Using truth values in Boolean logic and in IPL	401
D Category theory	402
E A humorous disclaimer	403
F GNU Free Documentation License	404
F.0.0 Applicability and definitions	404
F.0.1 Verbatim copying	404
F.0.2 Copying in quantity	404
F.0.3 Modifications	404
List of Tables	406
List of Figures	407
Index	408

Preface

The goal of this book is to teach programmers how to reason mathematically about types and code, in a way that is directly relevant to software practice.

The material is presented here at medium to advanced level. It requires a certain amount of mathematical experience and is not suitable for people who are unfamiliar with school-level algebra, or unwilling to learn difficult concepts through prolonged mental concentration and effort.

The first part is introductory and may be suitable for beginners in programming. Starting from Chapter 5, the material becomes unsuitable for beginners.

The presentation in this book is self-contained, defining and explaining all required notations, concepts, and Scala language features from scratch. The emphasis is on clarity and understandability of all examples, mathematical notions, derivations, and code. To achieve a clearer presentation of the material, the book uses some non-standard notations (Appendix A) and terminology (Appendix B).

The vision of this book is to explain the mathematical principles that guide the practice of functional programming – that is, help people to write code. Therefore, all mathematical developments in this book are motivated and justified by practical programming issues and are accompanied by code examples that illustrate their usage. For instance, the equational laws for standard typeclasses (functor, applicative, monad, etc.) are first motivated heuristically with code examples. Only then a set of mathematical equations is derived and the laws are formulated as equations.

Each new concept or technique is clarified by solved examples and exercises. Answers to exercises are not provided, but it is verified that the exercises are doable and free of errors. More difficult examples and exercises are marked by an asterisk (*).

A software engineer needs to know only a few fragments of mathematical theory; namely, the fragments that answer questions arising in the practice of functional programming. So the theoretical material is kept to the minimum; *ars longa, vita brevis*. (Chapter 18 presents more discussion of the scope of the required theory.) Mathematical generalizations are not pursued beyond practical relevance or immediate pedagogical usefulness. This limits the scope of required mathematical knowledge to bare rudiments of category theory, type theory, and formal logic. For instance, this book does not mention “introduction/elimination rules”, “strong normalization”, “complete partial order domains”, “adjoint functors”, “pullbacks”, “topoi”, or “algebras”, because learning these concepts will not help a functional programmer write code. This book is also not an introduction to today’s research in the theory of programming languages. Instead, the focus is on practically useful material – including some rarely mentioned constructions, such as the “filterable functor” and “applicative contrafunctor” typeclasses.

All code examples are intended only for explanation and illustration, and are not optimized for performance or stack safety.

The author thanks Joseph Kim and Jim Kleck for going through the exercises and reporting some errors in earlier versions of this book.

Formatting conventions used in this book

- Text in boldface indicates a new concept or term that is being defined. Text in italics is a logical emphasis. Example:

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

- Sample Scala code is written inline using a small monospaced font, such as this: `val a = "xyz"`. Longer code examples are written in separate code blocks, which may also show the output from the Scala interpreter for certain lines of code:

```
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 a new notation for types where e.g. the Scala type expression `((A, B)) => Option[A]` is written as $A \times B \Rightarrow 1 + A$. Also, a new notation for code is introduced and developed in Chapters 5–6 for easier reasoning about equational laws. For example, the functor composition law is written in the code notation as

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

where L is a functor and $f:A \Rightarrow B$ and $g:B \Rightarrow C$ are arbitrary functions of the specified types. The symbol $;$ denotes the forward composition of functions (Scala's `andThen` method). Appendix A summarizes the conventions of the type and code notations.

- Derivations of laws are written in a two-column notation where the right column contains formulas in the code notation and the left column indicates the property or law used to derive the expression at right. A green underline in the *previous* expression shows the part rewritten using the indicated law:

$$\begin{aligned} \text{expect to equal } pu_M : & \quad pu_M^{\uparrow \text{Id}} ; pu_M ; ftn_M \\ \text{lifting to the identity functor :} & \quad = pu_M ; \underline{pu_M} ; ftn_M \\ \text{left identity law for } M : & \quad = pu_M . \end{aligned}$$

A green underline is sometimes also used at the *last* step of the derivation, to indicate the part of the expression that resulted from the most recent rewriting.

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 writing Scala code for some computational tasks.

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

First, we write a mathematical formula for the result:

$$\prod_{k=1}^{10} k \quad .$$

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

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

The Scala interpreter indicates that the result is the value 3628800 of type `Int`. To define a name for this value, we use the “`val`” syntax:

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

```
scala> fac10 == 3628800
res1: Boolean = true
```

The code `(1 to 10).product` is an **expression**, which means that (1) the code can be evaluated (e.g. using the Scala interpreter) and yields a value, and (2) the code can be inserted as a part of a larger expression. For example, we could write

```
scala> 100 + (1 to 10).product + 100
res0: Int = 3629000
```

Example 1.1.1.2: Factorial as a function Define a function that takes an integer n and computes the factorial of 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:

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

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

Having defined the function `f`, we can now apply it to an integer argument:

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

It is an error to apply `f` to a non-integer value, e.g. to a string:

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

1.1.2 Nameless functions

The formula and the code, as written above, both involve *naming* the function as “ f ”. Sometimes a function does not really need a name, – for instance, if the function is used only once. “Nameless” mathematical functions are denoted using the symbol \mapsto (pronounced “mapped to”) like this:

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

So the mathematical notation for the nameless factorial function is

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

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 arrow symbol `=>` 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 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 interpreter² called `ammonite` prints something like this,

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

¹In Scala, the two ASCII characters `=>` and the single Unicode character \Rightarrow have the same meaning. This book uses the symbol \Rightarrow for symbolic calculations and `=>` for Scala code. However, when doing calculations *by hand*, one could write \rightarrow instead of \Rightarrow since it is faster. Several programming languages, such as OCaml and Haskell, use the symbols \rightarrow or the Unicode equivalent, \rightarrow , for the function arrow.

²<https://ammonite.io/>

This seems to indicate some identifying number, or perhaps a memory location.

One may imagine that a “function value” represents a block of compiled machine code, – code that will actually run and evaluate the function’s body when the function is applied to its argument.

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

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

However, functions can be used without naming them. We can 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
```

One would not often write code like this because there is no advantage in creating a nameless function and then applying it right away to an argument. This is so because we can evaluate the expression

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

by substituting 10 instead of `n` in the function body, which gives us

```
(1 to 10).product
```

If a nameless function uses the argument several times, for example

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

it is still better 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 **locally scoped** 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 as

$$\text{is_prime}(n) = \forall k \in [2, n-1] : n \neq 0 \bmod k . \quad (1.1)$$

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 should satisfy a given condition, $n \neq 0 \bmod k$. Formula (1.1) is translated into Scala code as

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

In this code, the two parts of the mathematical formula are implemented in a way that is closely similar to the mathematical notation, except for the arrow after k .

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

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

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

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

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

In Scala, it is optional – but strongly recommended – to specify the return type of named functions. The required syntax looks like this,

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

However, we do not need to specify the type `Int` for the argument `k` of the nameless function `k => n % k != 0`. 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`.

1.1.3 Nameless functions and bound variables

The code for `is_prime` differs from the mathematical formula (1.1) 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. (1.1) does not seem to involve any functions.

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 infix syntax `x.f(z)`, or equivalently `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 often easier to read and is widely used, e.g. when we write $x + y$ rather than something like $plus(x, y)$.

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

The infix 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 infix syntax, we could define a function `for_all` with two arguments and write code like this,

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

This would have brought the syntax somewhat closer to the formula (1.1).

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 mathematical formula

$$\forall k \in [2, n - 1] : n \neq 0 \bmod k \quad (1.2)$$

does not seem to use any functions but defines the symbol `k` that goes over the range $[2, n - 1]$. The variable `k` is then used for writing the predicate $n \neq 0 \bmod k$.

Let us investigate the role of `k` more closely. The mathematical variable `k` is actually defined *only inside* the expression “ $\forall k : \dots$ ” and makes no sense outside that expression. This becomes clear by looking at Eq. (1.1): 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 “ $\dots \bmod k$ ”.

So, the mathematical notation in Eq. (1.2) 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 $n \neq 0 \bmod k$, 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

$$k \mapsto n \neq 0 \bmod k$$

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. (1.1), 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 $k \Rightarrow n \neq 0 \bmod k$ (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. (1.1) an equivalent definition

$$\text{is_prime}(n) = \forall z \in [2, n - 1] : n \neq 0 \bmod z ,$$

or in Scala code,

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

The argument z in the nameless function $z \Rightarrow n \% z != 0$ 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 $z \Rightarrow n \% z != 0$, we would also need to change every place in the 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 $\forall k : p(k)$, $\exists k : p(k)$, $\sum_{k=a}^b f(k)$, $\int_0^1 k^2 dk$, $\lim_{n \rightarrow \infty} f(n)$, and $\operatorname{argmax}_k f(k)$. 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 need to create a nameless function whose argument is that variable, e.g. `k=>p(k)` or `k=>f(k)` for the examples just shown. Only then will our code 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 $k \Rightarrow p(k)$ 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 $x \Rightarrow f(x)$ 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 like this,

$$\begin{aligned} \text{count_even}(L) &= \sum_{k \in L} \text{is_even}(k) , \\ \text{is_even}(k) &= \begin{cases} 1 & \text{if } k = 0 \bmod 2 \\ 0 & \text{otherwise} \end{cases} . \end{aligned}$$

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

³Certain features of Scala allow programmers to write code that looks like `f(x)` but actually involves additional implicit or default arguments of the function `f`, or an implicit type conversion for its argument `x`. In those cases, replacing the code `x => f(x)` by `f` will fail to compile. But these complications do not arise when working with simple functions.

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

```
def is_even(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 is_even = (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 `is_even` 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 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 infix 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, applies that function to each element of the list, and puts all the results into 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 $x \mapsto x^2 + 100x$. 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.

An infix method, such as `.map`, can be also used with a “dotless” syntax:

```
scala> 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 operation such as $x \mapsto x^2 + 100x$, it is easier to work with a nameless function.

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

```
def count_even(s: List[Int]) = s.map(is_even).sum
```

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

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

It is customary in Scala to use infix methods when chaining several operations. For instance `s.map(...).sum` means first apply `s.map(...)`, which returns a *new* list, and then apply `.sum` to that list. To make the code more readable, we 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 a compiled code, the dots can be at the beginning of the lines since the compiler reads the entire code at once.)

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

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

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

Note that the Scala interpreter prints the types differently for 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:

$$\begin{aligned} \text{forall } (S, p) &= \forall k \in S : p(k) = \text{true} \\ \text{exists } (S, p) &= \exists k \in S : p(k) = \text{true} \end{aligned}$$

However, there is no mathematical notation for operations such as “removing elements from a list”, so we will focus on the Scala syntax for these functions.

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

```
scala> List(1, 2, 3, 4, 5).filter(k => k % 3 != 0)
res5: List[Int] = List(1, 2, 4, 5)
```

The `.takeWhile` method truncates a given list, returning a new list with the initial portion of values from the original list for which predicate keeps being `true`:

```
scala> List(1, 2, 3, 4, 5).takeWhile(k => k % 3 != 0)
res6: List[Int] = List(1, 2)
```

In all these cases, the predicate's argument `k` must 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 `.max`, `.min`, `.sum`, and `.product` are defined on lists of *numeric types*, such as `Int`, `Double`, and `Long`. The other methods are defined on lists of all types.

Using these methods, we can solve many problems that involve transforming and aggregating data stored in lists (as well as in arrays, sets, or other similar data structures). A **transformation** is

a function taking a list of values and returning another list of values; examples of transformation functions are `.filter` and `.map`. An **aggregation** is a function taking a list of values and returning a *single* value; examples of aggregation functions 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 Aggregation

Example 1.4.1.1 Improve the code for `is_prime` by limiting the search to $k^2 \leq n$:

$$\text{is_prime}(n) = \forall k \in [2, n-1] \text{ such that } k^2 \leq n : n \neq 0 \bmod k .$$

Solution: Use `.takeWhile` to truncate the initial list when $k^2 \leq n$ becomes false:

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

Example 1.4.1.2 Compute $\prod_{k \in [1,10]} |\sin(k+2)|$.

Solution

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

Example 1.4.1.3 Compute $\sum_{k \in [1,10]; \cos k > 0} \sqrt{\cos k}$.

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]`,

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

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 product⁴ truncated up to $\frac{2n}{2n+1}$:

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

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

Solution Define the helper function `wallis_frac(i)` that computes the i^{th} 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 limit of the Wallis product is $\frac{\pi}{2}$, so the cosine of `wallis(n)` tends to zero in the limit of large n .

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 this 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 = 4*math.atan(1)
pi: Double = 3.141592653589793

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

Example 1.4.1.7 Check numerically the infinite product formula

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

Solution Compute this product up to $k = n$ for $x = 0.1$ and 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 $f: \text{Int} \Rightarrow \text{Int}$, 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 Transformation

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]]`.

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

Table 1.1: Translating mathematics into code.

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 the type of `t` from the code; no other type would work since 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 $j^2 > 2k$.

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 $j^2 > 2k$, 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 $\sum_{k=1}^n f(k)$ etc.
- 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

$$f(f(f(\dots f(0)\dots)) > 1000 ,$$

where the given function f is applied n times.

- Example 2: Given a list s of numbers, compute the list r of running averages:

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

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

These computations involve *mathematical induction*, which we have not yet learned to translate into code in the general case.

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. 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,

$$r_0 = s_0 ; \quad r_i = s_i + r_{i-1} \text{ for } i = 1, 2, 3, \dots$$

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 by 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 by translating mathematical induction into code using recursion.

1.6 Exercises

1.6.1 Aggregation

Exercise 1.6.1.1 Machin's formula⁵ converges to π faster than Example 1.4.1.5:

$$\begin{aligned} \frac{\pi}{4} &= 4 \arctan \frac{1}{5} - \arctan \frac{1}{239} , \\ \arctan \frac{1}{n} &= \frac{1}{n} - \frac{1}{3} \frac{1}{n^3} + \frac{1}{5} \frac{1}{n^5} - \dots = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} n^{-2k-1} . \end{aligned}$$

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 about 12 terms of the series are already sufficient for a full-precision `Double` approximation of π .

⁵<http://turner.faculty.swau.edu/mathematics/materialslibrary/pi/machin.html>

Exercise 1.6.1.2 Using the function `is_prime`, check numerically the Euler product formula⁶ for the Riemann zeta function $\zeta(4)$; it is known⁷ that $\zeta(4) = \frac{\pi^4}{90}$:

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

1.6.2 Transformation

Exercise 1.6.2.1 Define a function `add_20` of type `List[List[Int]] => List[List[Int]]` that adds 20 to every element of every inner list. A sample test:

```
scala> add_20( 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 $n \in [1, \dots, 1000]$.

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 $j \in [1, \dots, n]$ 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 $n \in [1, \dots, 1000]$. 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: finds the element having the largest absolute value and, if that value is nonzero, divides all elements by that factor and returns a new list; otherwise returns the original list.

1.7 Discussion

1.7.1 Functional programming as a paradigm

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

The main principle 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.” Similarly to 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 backed by the vast experience accumulated while 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. Functional programmers are fortunate to have these reasoning tools at their disposal.

As we have seen, the Scala code for certain computational tasks corresponds quite closely to mathematical formulas. (Scala conventions and syntax, of course, require programmers to spell out certain things 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,

⁶https://en.wikipedia.org/wiki/Proof_of_the_Euler_product_formula_for_the_Riemann_zeta_function

⁷<https://tinyurl.com/yxey4tsd>

flexibly composed together, and written independently from each other. 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, building up the specific kind of applied mathematical intuition, familiarizing oneself with concepts adapted to a programmer's needs, and learning how to translate the mathematics into code in various cases. The FP community has discovered a number of specific design patterns founded on mathematical principles but driven by practical necessities of programming (rather than by the needs of academic mathematics). This book explains the required mathematical design patterns in detail, developing them through intuition and examples of Scala code.

1.7.2 Functional programming languages

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 much easier to use in practice. For example, in a language such as Python or Ruby, one can productively use only a limited number of FP idioms, such as the map/reduce operations. More advanced FP constructions 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 ML, F#, Scala, Swift, Elm, and PureScript, 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 ML, OCaml, Haskell, F#, Scala, Swift, Elm, or PureScript do not play a significant role.

1.7.3 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, let us 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 $k \mapsto k^2 + k$. 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 *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
```

because it is clear that this x is an integer. However, it is often helpful to write out 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"
```

1.7.4 Iteration without loops

Another 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 estimated from a sample,

$$\sigma(s) = \sqrt{\frac{1}{n-1} \sum_{i=1}^n \sum_{j=1}^n s_i s_j - \frac{1}{n(n-1)} \left(\sum_{i=1}^n s_i \right)^2} .$$

These expressions are computed by iterating over values of i and j . 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 in a loop, or 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 $\sum_{i=1}^n$ or $\prod_{i=1}^n$ denote iterative computations. Such computations are defined using mathematical induction. The functional programming paradigm has developed rich tools for translating mathematical induction into code. In this chapter, we have seen methods such as `.map`, `.filter`, and `.sum`, which

implement certain kinds of iterative computations. These and other operations can be combined in very flexible ways, which allows programmers to write iterative code without loops.

The programmer can avoid writing loops because the iteration is delegated to the library functions `.map`, `.filter`, `.sum`, and so on. It is the job of the library and the compiler to translate these functions into machine code. The machine code most likely *will* contain loops; but the functional programmer does not need to see that code or to reason about it.

1.7.5 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 $z \Rightarrow \frac{1}{1+z}$, and the integral rewritten as code such as

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

Now consider the traditional mathematical notations for summation, e.g.

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

In that sum, the bound variable k is introduced under the Σ 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:

$$\begin{aligned} &\text{denote summation by } \sum_0^x \left(k \mapsto \frac{1}{1+k} \right) \text{ instead of } \sum_{k=0}^x \frac{1}{1+k} \quad , \\ &\text{denote integration by } \int_0^x \left(z \mapsto \frac{1}{1+z} \right) \text{ instead of } \int_0^x \frac{dz}{1+z} \quad . \end{aligned}$$

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 formulas are more complicated. Simpson's rule⁸ is an approximate algorithm for numerical integration that can be defined by the formula

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

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

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

A straightforward line-by-line translation of this formula into Scala is

```
def integration(a: Double, b: Double, g: Double => Double, eps: Double): Double = {
    // First, we define some helper values and functions that replace
    // the definitions "where n = ..." in the mathematical formula.
    val n: Int = 2 * ((b - a) / eps).toInt
    val delta_x = (b - a) / n
    val s1 = (1 to (n - 1) by 2).map { i => g(a + i * delta_x) }.sum
    val s2 = (2 to (n - 2) by 2).map { i => g(a + i * delta_x) }.sum
    // Now we 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*x, eps = 0.01)      // Exact answer is 625.
res0: Double = 625.0000000004167

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

The entire code is one large *expression*, with a few sub-expressions (s_1 , s_2 , etc.) defined for convenience in the local scope of the function. In other words, this code is written in the FP paradigm.

1.7.6 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 most programming languages today, we can directly write arithmetical expressions such as $(x+123)*y/(2+x)$. Here, x and y are variables with names. Note, however, that the entire expression does not need to have a name. Parts of that expression (such as $x+123$ or $2+x$) also do not have separate names. It would be quite inconvenient if we *needed* to assign a name separately to each sub-expression. The code for $(x+123)*y/(2+x)$ could then look like this:

```
r1 = 123
r2 = x + r1
r3 = r2 * y
r4 = 2
r5 = r4 + x
r6 = r3 / r5
return r6
```

This style of programming resembles assembly languages, where *every* sub-expression – that is, every step of every calculation, – must be named separately (and, in the assembly languages, assigned a 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 similarly useful if data structures can be created without names. For instance, a **dictionary** (also called a “hashmap”) is created in Scala as

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

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

This code is a nameless expression whose value is a dictionary. In programming languages that do not have such a construction, programmers have to write repetitive 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 nameless values of other types: they allow us to build larger programs from simpler parts in a uniform way.

1.7.7 Historical perspective on usage of 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 letter λ is a syntax separator denoting function arguments in nameless functions. For example, the nameless function $x \mapsto x + 1$ could be written as $\lambda x.add\ x\ 1$ in λ -calculus, if it had a function *add* for adding integers (which it does not).

In most programming languages that were in use until around 1990, all functions required names. But by 2015, most languages added support for nameless functions, because programming in the map/reduce style (which invites frequent use of nameless functions) turned out to be immensely productive. Table 1.2 shows the year when nameless functions were introduced in each language.

What this book calls a “nameless function” is also called anonymous function, function expression, function literal, closure, lambda function, lambda expression, or just a “lambda”.

Language	Year	Code for $k \mapsto k + 1$
λ -calculus	1936	$\lambda k. add\ k\ 1$
typed λ -calculus	1940	$\lambda k : \text{int}. add\ k\ 1$
LISP	1958	(lambda (k) (+ k 1))
Standard ML	1973	fn (k:int) => k + 1
Scheme	1975	(lambda (k) (+ k 1))
OCaml	1985	fun k -> 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; }
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.

⁹Although called a “calculus,” it is a (drastically simplified) programming language. It has nothing to do with “calculus” as known in mathematics, such as differential or integral calculus. Also, the letter λ has no particular significance; it plays a purely syntactic role in the λ -calculus. Practitioners of functional programming usually do not need to study any λ -calculus. All practically relevant knowledge related to λ -calculus is explained in Chapter 4 of this book.

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, – such as, 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**: 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:

- destructure 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 the new values are also assigned automatically.

In the example above, the left-hand side of the destructure 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 destructure definition checks whether the structure of the value of `g` “matches” the three pattern variables. (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

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

“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 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> t((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:

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

The only change in the function body, compared to `t1`, is in the pattern matcher. So it is visually clear that `t4` computes the same expression as `t1`.

Sometimes we do not need some of the tuple parts in a pattern match. The following syntax is used to make this intention 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.

A feature of Scala is a short syntax for functions such as `{case (x, y) => y}` that extract elements from tuples. The shorter syntax is `(t => t._2)` or even shorter, `(_.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. 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 `{ ... }` can 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 (Scala's Maps) 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 useful methods that use tuple types, such as `.map` and `.filter` (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 `.map` and `.toMap` methods 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 m1 = Map("apples" -> 3, "pears" -> 2, "lemons" -> 0)

scala> m1.map { case (fruit, count) => count * 2 }
res0: Seq[Int] = ArrayBuffer(6, 4, 0)
```

If we want to transform a dictionary into another dictionary, we first create a sequence of pairs, transform it, and then convert it to a dictionary with the `.toMap` method:

```
scala> m1.map { case (fruit, count) => (fruit, count * 2) }.toMap
res1: Map[String,Int] = Map(apples -> 6, pears -> 4, lemons -> 0)
```

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

```
scala> m1.filter { case (fruit, count) => count > 0 }.toMap
res2: Map[String,Int] = Map(apples -> 6, pears -> 4)
```

The `.zip` and `.zipWithIndex` methods 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 transforms a sequence into a sequence of pairs, where the second part of the pair is the zero-based index:

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

The `.flatten` method 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)
```

The “flattening” operation computes the concatenation of the inner sequences. In Scala, sequences are concatenated using the operation `++`, e.g.:

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

So the `.flatten` method inserts the operation `++` between all the inner sequences.

Keep in mind that `.flatten` removes *only one* level of nesting, which is at the “outside” of the data structure. If applied to a `List[List[List[Int]]]`, the `.flatten` method returns a `List[List[Int]]`:

```
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 `.flatMap` method 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 a potentially different number of new elements.

At first sight, it may be unclear why `.flatMap` is useful. (Should we perhaps combine `.filter` and `.flatten` into a `.flatFilter`, or combine `.zip` and `.flatten` into a `.flatZip`?) However, we will see later in this book that the use of `.flatMap`, which is related to “monads”, is one of the most versatile and powerful design patterns in functional programming. In this chapter, several examples and exercises will illustrate the use of `.flatMap` for working on sequences.

The `.groupBy` method 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", "xanthan", "yoghurt", "zebra").
  groupBy(s => if (s startsWith "y") 1 else 2)
res12: Map[Int,Seq[String]] = Map(1 -> List(yoghurt), 2 -> List(wombat, xanthan, 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 `.sliding` method 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))
```

Usually, this method is used together with an aggregation operation on 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 `.sortBy` method 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 times the predicate was `true` while evaluated on the elements of the sequence:

```
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]`.

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 1 element shorter than `ps`, and the resulting sequence of pairs is also 1 element shorter than `ps`. In other words, 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 $c_i = \max(b_{i-k}, \dots, b_{i+k})$.

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

```
scala> val bs = List(1,2,3,4,5)
bs: List[Int] = List(1, 2, 3, 4, 5)

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

For each b_i , we need to obtain a list of $2k + 1$ nearby elements $(b_{i-k}, \dots, b_{i+k})$. 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`:

```
bs.sliding(2 * k + 1).map(nested => nested.max)
```

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

```
bs.sliding(2 * k + 1).map(_.max)
```

because `_.max` means the nameless function `x => x.max`.

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), \dots, (2,1), (2,2), \dots)`. 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 parentheses and write `(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 completely equivalent to writing the tuple `(key, value)`. The final code is

```
(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`, `.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, 0.01000000000000002, 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, 0.01000000000000002, 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, 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.

Hint: use `.map` and `.toMap`.

Solution Keep in mind that iterating over a dictionary looks like iterating over a list of `(key, value)` pairs, and use `.map` to reverse each pair:

```
dict.map{ case (name, addr) => (addr, name) }.toMap
```

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) }.toMap
```

The body of the function `rev` remains the same as in Example 2.1.5.8; only the type signature changed. 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 words, compute a sequence of type `Seq[Seq[String], Int]`, where each inner sequence contains all the words having the same length, paired with the integer value showing that length. So, the input `Seq("the", "food", "is", "good")` should produce the output

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

The resulting sequence must be ordered by increasing length of words.

Solution It is clear that we need to 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. In our case, we need to group by word length, which is computed with the method `.length` if applied to a string. So the first step is to 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: scala.collection.immutable.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` 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 type.

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 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.
Map[Int, Seq[String]] // Need to sort by word length; can't sort a dictionary!
// Need to convert this dictionary to a sequence:
```

```
Seq[ (Int, Seq[String]) ] // Now sort this! Sorting does not change the type.
// 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 assurance in the correctness of the code.

2.1.7 Exercises: Tuples and collections

Exercise 2.1.7.1 Find all pairs i, j within $(0, 1, \dots, 9)$ 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 $p: \text{Seq[String]}$ and $q: \text{Seq[Boolean]}$ 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 $\text{Seq[(Int, Boolean)]}$ where the `Boolean` value is `true` when the element is followed by a larger value. For example, the input sequence $\text{Seq}(1, 3, 2, 4)$ is to be converted into $\text{Seq}((1, \text{true}), (3, \text{false}), (2, \text{true}), (4, \text{false}))$. (The last element, 4, has no following element.)

Exercise 2.1.7.5 Given $p: \text{Seq[String]}$ and $q: \text{Seq[Int]}$ 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 2.1.7.6 Write the solution of Exercise 2.1.7.5 as a function with type parameters P and Q instead of the fixed types `String` and `Int`. Test it with $P = \text{Boolean}$ and $Q = \text{Set[Int]}$.

Exercise 2.1.7.7 Given $p: \text{Seq[String]}$ and $q: \text{Seq[Int]}$ of equal length, compute a Seq[String] that contains the strings from p ordered according to the corresponding numbers from q . For example, if $p = \text{Seq("a", "b", "c")}$ and $q = \text{Seq}(10, -1, 5)$ then the result must be $\text{Seq("b", "c", "a")}$.

Exercise 2.1.7.8 Write the solution of Exercise 2.1.7.7 as a function with type parameter s instead of the fixed type `String`. The required type signature and a sample test:

```
def reorder[S](p: Seq[S], q: Seq[Int]): Seq[S] = ???

scala> reorder(Seq(6.0, 2.0, 8.0, 4.0), Seq(20, 10, 40, 30))
res0: Seq[Double] = List(2.0, 6.0, 4.0, 8.0)
```

Exercise 2.1.7.9 Given a $\text{Seq[(String, Int)]}$ showing a list of purchased items (where item names may repeat), compute a Map[String, Int] showing the total counts: e.g. for the input

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

the output must be

```
Map("apple" -> 7, "pear" -> 3)
```

Implement this computation as a function with type parameter s instead of `String`.

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

Exercise 2.1.7.10 Given a Seq[List[Int]] , compute a new Seq[List[Int]] where each inner list contains three largest elements from the initial inner list (or fewer than three if the initial 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`; that 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]] = ???

scala> payments(Seq(Map("Tarski" -> 10, "Church" -> 20), Map("Church" -> 100, "Gentzen" -> 40),
  Map("Tarski" -> 50)))
res0: Map[String, Seq[Int]] = Map(Gentzen -> List(40), Church -> List(20, 100), Tarski -> List(10,
  50))
```

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 using `.map`, `.sum`, 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
```

This task is doable because the required computation can be written as the formula

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

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 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()`. If 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 ++ Seq(x)`. So, we need to specify how to compute `f(xs ++ Seq(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. This is how induction works in mathematics, and it works in the same way in functional programming. With this approach, the inductive definition of the method `.sum` looks like this:

- The sum of an empty sequence is 0. That is, `Seq().sum == 0`.
- If the result is already known for a sequence `xs`, and we have a sequence that has one more element `x`, the new result is equal to `xs.sum + x`. In code, this is `(xs ++ Seq(x)).sum == xs.sum + x`.

The inductive definition of the function `digitsToInt` is:

- For an empty sequence of digits, `Seq()`, the result is 0. This is a convenient base case, even if we never call `digitsToInt` on an empty sequence.
- If `digitsToInt(xs)` is already known for a sequence `xs` of digits, and we have a sequence `xs ++ Seq(x)` with one more digit `x`, then

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

Let us write inductive definitions for methods such as `.length`, `.max`, and `.count`:

- The length of a sequence:
 - for an empty sequence, `Seq().length == 0`
 - if `xs.length` is known then `(xs ++ Seq(x)).length == xs.length + 1`
- Maximum element of a sequence (undefined for empty sequences):
 - for a one-element sequence, `Seq(x).max == x`
 - if `xs.max` is known then `(xs ++ Seq(x)).max == math.max(xs.max, x)`
- Count the sequence elements satisfying a predicate `p`:

- for an empty sequence, `Seq().count(p) == 0`
- if `xs.count(p)` is known then `(xs ++ Seq(x)).count(p) == xs.count(p) + c`, where we set `c = 1` when `p(x) == true` and `c = 0` otherwise

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**.

When the body of a recursive function is evaluated, it may repeatedly call itself with different arguments until the result value can be computed *without* any recursive calls. The last recursive call corresponds to the base case of the induction. It is an error 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, and the inductive step returns a value computed from the recursive call:

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

In this example, we use the `if/else` expression to separate the base case from the inductive step. In the inductive step, we split the

given sequence `s` into a single-element sequence `Seq(x)`, the “head” of `s`, and the remainder (“tail”) sequence `xs`. So, we split `s` as `s = Seq(x) ++ xs` rather than as `s = xs ++ Seq(x)`.

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 = Seq(x) ++ xs` or as `s = xs ++ Seq(x)`, according to the task at hand.

Consider the implementation of `digitsToInt` according to the inductive definition shown in the previous subsection:

```
def digitsToInt(s: Seq[Int]): Int = if (s == Seq()) 0 else {
  val x = s.last // To split s = xs ++ Seq(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 into `s = xs ++ Seq(x)` in this order, and not in the order `Seq(x) ++ xs`. The reason is

that digits increase their numerical value from right to left, so we need to multiply the value of the *left* subsequence, `digitsToInt(xs)`, by 10, in order to compute the correct result.

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 the “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 each inductively defined function repeats the code for checking the base case and the code 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 == Seq()) 0
  else 1 + lengthS(s.tail)

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

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)
  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* an 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 function body of `lengthS` will evaluate the inductive step, that is, the “`else`” part of the “`if/else`”, about 100000 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 100000 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, as if 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.

A 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 == Seq()) res
  else lengthT(s.tail, 1 + res)
```

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 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 `1+res` 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 has a feature for checking automatically 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.

```
@tailrec def lengthT(s: Seq[Int], res: Int) =
```

```
if (s == Seq()) res
else lengthT(s.tail, 1 + res)
```

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

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

How did we rewrite the code of `lengths` to obtain 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 ($1 + \text{res}$) 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 an “adapter” 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 adapter function:

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

When `length` is implemented like that, users will not be able to call `lengthT` directly, because it 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 == Seq()) res
  else length(s.tail, 1 + res)
```

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

Using a default argument value, we can define the tail-recursive helper function and the adapter 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 induction step for the accumulator. In the example with the `lengthT` function, the accumulator trick works because of the following mathematical property of the expression being computed:

$$1 + (1 + (1 + (\dots + 1))) = (((1 + 1) + 1) + \dots) + 1 .$$

All sub-expressions such as $1 + 1$ and $1 + 2$ 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.

This property is called the **associativity law** of addition. Due to this law, the computation can be rearranged so that additions associate to the left. In code, it means that intermediate expressions are computed immediately before making the 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, consider a tail-recursive re-implementation 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,

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

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

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

It follows that the digit sequence s must be split into the *leftmost* digit and the rest, $s = s.\text{head} ++ s.\text{tail}$. 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 == Seq()) res
  else fromDigits(s.tail, 10 * res + s.head)
```

Despite a certain similarity between this code and the code of `digitsToInt` from the previous subsection, the implementation `fromDigits` cannot be directly derived from the inductive definition of `digitsToInt`. One needs a separate proof that `fromDigits(s, 0)` computes the same result as `digitsToInt(s)`. The proof follows from the following property.

Statement 2.2.3.1 For any $\text{xs}: \text{Seq[Int]}$ and $\text{r}: \text{Int}$, we have

$$\text{fromDigits}(\text{xs}, \text{r}) = \text{digitsToInt}(\text{xs}) + \text{r} * \text{math.pow}(10, \text{s.length})$$

Proof We prove this by induction. 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 `fromDigits(s, r)`. Then an inductive definition of $f(s, r)$ is

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

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

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

We prove Eq. (2.2) by induction. To prove the base case $s = []$, we have $f([], r) = r$ and $d([]) + r * 10^0 = r$ since $d([]) = 0$ and $|[]| = 0$. 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 we need to prove that

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

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):

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

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,

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

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

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

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 uses induction and begins by writing the code of d in a short notation,

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

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

$$x = d([]++[x]) = d([x]++) = x * 10^0 + d([]) = x .$$

The induction 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 :

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

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

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

This demonstrates Eq. (2.6) and so 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 value may be different from the type of elements in the sequence. To describe this general situation, we introduce type parameters, A and B , so that the input sequence is of type $\text{Seq}[A]$ and the aggregated value is of type B . Then an inductive definition of any aggregation function $f: \text{Seq}[A] \Rightarrow B$ looks like this:

- (Base case.) For an empty sequence, $f(\text{Seq}()) = b0$ where $b0:B$ is a given value.
- (Induction step.) Assuming that $f(xs) = b$ is already computed, we define $f(xs ++ \text{Seq}(x)) = g(x, b)$ where g is a given function with type signature $g: (A, B) \Rightarrow B$.

The code implementing f is written using recursion:

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

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.last ++ s.take(s.length - 1)` 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 == Seq()) 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 induction 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 }
```

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

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.

The method `.foldLeft` is available in the Scala library for all collections, including dictionaries and sets. Since `.foldLeft` is tail-recursive, no stack overflows will occur even for very large sequences.

The Scala library contains several other methods similar to `.foldLeft`, such as `.foldRight` and `.reduce`. (However, `.foldRight` is not tail-recursive!)

2.2.5 Solved examples: using `foldLeft`

It is important to gain experience using the `.foldLeft` method.

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 Write an inductive formulation of the `max` function:

- (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 ++ Seq(x)` is `math.max(b, x)`.

Now we can write the code:

```
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 Implement the `count` method on sequences of type `Seq[A]`.

Solution Using the inductive definition of the function `count` as shown in Section 2.2.1, we can write the code as

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

Example 2.2.5.3 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 2.2.5.4 For a given non-empty sequence `xs: Seq[Double]`, compute the minimum, the maximum, and the mean as a tuple $(x_{\min}, x_{\max}, x_{\text{mean}})$. The sequence should be traversed only once, i.e. the 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 function must update 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 2.2.5.5* 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 these formulas:

- Before the dot character: $g(n, c) = n * 10 + c$.
- After the dot character: $g(n, c) = n + \frac{c}{f}$, where f is 10, 100, 1000, ..., for each new digit.

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 (n, flag, factor) =>
  if (c == '.') (n, true, factor) // Set flag to 'true' after a dot character was seen.
  else {
    val digit = c - '0'
    if (flag) // This digit is after the dot. Update 'factor'.
      (n + digit/factor, flag, factor * 10)
    else // This digit is before the dot.
      (n * 10 + digit, flag, factor)
  }
}
```

Now we can implement `digitsToDouble` as follows,

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

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

The result of calling `d.foldLeft` is a tuple `(n, flag, factor)`, in which only the first part, `n`, is needed. In Scala's pattern matching expressions, the underscore symbol is used to denote the pattern variables whose values are not needed in the subsequent code. We could extract the first part using the accessor method `._1`, but the code will be more readable if we show all parts of the tuple by writing `(n, _, _)`.

Example 2.2.5.6 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. We can append elements to sequences using the `:+` operation:

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

The operation `acc :+ f(x)` is equivalent to `acc ++ Seq(f(x))` but is shorter to write.

Example 2.2.5.7 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 use `.foldLeft` 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 “remainder”: it holds the previous element for odd-numbered iterations and is empty for even-numbered iterations. Initially, the accumulator should be empty. A trace of the accumulator updates is shown in this table:

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 ++ Seq((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 “remainder” 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 ++ Seq((prev, x)), Seq())
  }
  val (result, remainder) = xs.foldLeft(init)(updater)
  remainder match {                    // May need to append the last element to the result.
    case Seq()    => result
    case Seq(x)   => result ++ Seq((x, default))
  }
}
```

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

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

This code works when `remainder` is empty or has length 1 but would fail for longer sequences. Within the implementation of `toPairs`, the value of `remainder` will always be a sequence of length at most 1, so the code is safe.

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 2.2.5.7. The required type signature and a sample test:

```
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:

```
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).getOrDefault("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 or “folds” a sequence into a single value; the opposite operation (“unfold-ing”) 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 requires us to build a sequence whose length is not determined in advance. This kind of sequence is called a **stream**. A stream is a sequence whose elements are computed only when necessary (unlike sequences such as a `List` or an `Array`, whose elements are all computed in advance and stored). The unfolding operation will keep computing the next element; 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 to 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`. 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 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 of the stream until it runs out of memory and crashes.

Streams are similar to sequences, and methods such as `.map`, `.filter`, `.flatMap` are also defined for streams. For instance, we can use the method `.drop` that 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 sequence of digits for, 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 the given initial integer.
- Inductive step: $n_{k+1} = \left\lfloor \frac{n_k}{10} \right\rfloor$ for $k = 1, 2, \dots$

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`. A complete implementation for `digitsOf` is thus

```
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
  }
```

The type signature of the method `Stream.iterate` can be written as

```
def iterate[A](init: A)(next: A => A): Stream[A]
```

and 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.) $b_0 = 0$.
- (Induction step.) Given b_k , we define $b_{k+1} = b_k + x_k$ for $k = 0, 1, 2, \dots$

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)

```

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 work 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)
2   : Seq[B] = {
3   val init: (B, Seq[B]) = (b0, Seq(b0))
4   val (_, result) = xs.foldLeft(init){ case ((b, seq), x) =>
5     val newB = next(b, x)
6     (newB, seq ++ Seq(newB))
7   }
8   result
9 }

```

ond is a value of type `A`. The pattern matching expression `{case ((b, seq), x) => ...}` appears to match a nested tuple. In reality, this expression matches the two arguments of the updater function and, at the same time, destructures the tuple argument as `(b, seq)`.

The first argument of `.scanLeft` is the base case, and the second argument is an updater function describing the induction step. In general, the type of elements of the second sequence is different from that of the first sequence. The updater function takes an

To implement the (nameless) updater function for `.foldLeft` in lines 4–7, we used the 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 sec-

2.5 Summary

We have seen a broad overview of 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
$f([]) = b ; f(s++[x]) = g(f(s), x)$	<code>f(xs) = xs.foldLeft(b)(g)</code>
$x_0 = b ; x_{k+1} = g(x_k)$	<code>xs = Stream.iterate(b)(g)</code>
$y_0 = b ; y_{k+1} = g(y_k, x_k)$	<code>ys = xs.scanLeft(b)(g)</code>

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 any 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 this other sequence,” etc. This is a powerful way of working with sequences, dictionaries, and sets. Many kinds of programming errors (such as an incorrect array index) are avoided from the outset, and the code is shorter and easier to read than conventional code written using loops.

What tasks are not possible 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.)

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 tricks (e.g. the “continuation-passing” or “trampolines”) that convert non-tail-recursive code into iterative code without stack overflows. Those advanced tricks are beyond the scope of this chapter.

2.5.1 Solved examples

Example 2.5.1.1 Compute the smallest n such that $f(f(f(\dots f(1)\dots)) \geq 1000$, 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 We define a stream of values $[1, f(1), f(f(1)), \dots]$ and use `.takeWhile` to stop the stream when the given condition holds. The `.length` method then gives 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){ case (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 tutorial, 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]()) { case (seq, x) =>
    // Sort in the descending order, and take the first k values.
    (seq :+ x).sorted.reverse.take(k)
  }.drop(1) // Skip the useless 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(Seq(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 is fixed – it 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 { case (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 fox")
res0: Map[String, Int] = Map("a" -> 3, "quick" -> 2, "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 and define the type signature of the function to contain a type parameter `A` instead of `String`:

```
def countValues[A](xs: Seq[A]): Map[A, Int] =
  xs.foldLeft(Map[A, Int]()) { (dict, word) =>
    val newCount = dict.getOrElse(word, 0) + 1
    dict.updated(word, 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 algorithm¹ 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 well-known 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 (???are we done???) acc
  else {
    // Determine which half of the sequence contains 'goal'.
    // Then update the accumulator accordingly.
    val newAcc = ???
    binSearch(xs, goal, newAcc) // Tail-recursive call.
  }
}
```

We now decide the type and the initial value of the accumulator, and implement the updater for it.

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 $x_i \leq x_n < x_{j-1}$. 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 $i + 1 = j$. We now write the code, introducing for convenience *two* accumulator values representing (i, j) :

```
@tailrec def binSearch(xs: Seq[Int], goal: Int)(left: Int = 0, right: Int = xs.length): Int = {
  // 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 group** from `xs`. Default values in an argument group may depend on arguments in a *previous* argument group. 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 group 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 sequence 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 make the use of `.takeWhile` more convenient, we add an extra sequence element where the bounds (k, k) are equal. The code becomes

¹https://en.wikipedia.org/wiki/Binary_search_algorithm

```

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) (left, left) // Extra element (k, k) in the stream.
    else if (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)
  .takeWhile{ case (left, right) => right > left }
  .last._1 // Take the 'left' boundary from the last element.
}

```

This code is clearer because recursion is delegated to `Stream.iterate` and 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 $n: \text{Int}$, compute the sequence $[s_0, s_1, s_2, \dots]$ 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`; compute the digits in the reverse order since their sum will be the same.

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 by almost the same code as in Section 2.3:

```

def SD(n: Int): Int = if (n == 0) 0 else
  Stream.iterate(n)(_ / 10).takeWhile(_ != 0).map(_ % 10).sum

```

Now we can try evaluating SD on some numbers to see its behavior:

```

scala> (1 to 15).toList.map(SD)
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6)

```

It is clear that $SD(n) < n$ as long as $n \geq 10$. So the sequence elements s_i will not repeat until they become smaller than 10, and then they will always repeat. This seems to be an easy way of stopping the sequence. Let us try that:

```

scala> Stream.iterate(99)(SD).takeWhile(x => x >= 10).toList
res1: List[Int] = List(99, 18)

```

We are missing the last element of the sequence, $SD(18) = 9$, because `.takeWhile` stops the sequence too early. In order to obtain the correct sequence, we need to compute one more element. To fix this, we can generate a stream of pairs:

```

scala> Stream.iterate((0, 99)){ case (prev, x) => (x, SD(x)) }.
  takeWhile{ case (prev, x) => prev >= 10 || x >= 10 }.toList
res2: 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((0, n)){ case (prev, x) => (x, SD(x)) } // Stream[(Int, Int)]
  .takeWhile{ case (prev, x) => prev >= 10 || x >= 10 } // 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 induction 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 $[s_0, s_1, s_2, \dots]$ of type `Stream[T]`, compute the “half-speed” stream $h = [s_0, s_0, s_1, s_1, s_2, s_2, \dots]$. The half-speed sequence h is defined as $h_{2k} = h_{2k+1} = s_k$ for $k = 0, 1, 2, \dots$

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 $[s_0, s_1, s_2, \dots]$ 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 $s = [1, 3, 5, 7, 9, 3, 5, 7, 9, \dots]$ we obtain the half-speed sequence $h = [1, 1, 3, 3, 5, 5, 7, 7, 9, 9, 3, 3, \dots]$. 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 $k = 1, 2, \dots$ 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 programming² 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 $\{[1,2], [2,5,4], [4,8]\}$. As we add another element, 2, we update the 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' cannot be repeated.  
        val xWasRepeated = current != Seq() && current.last == x  
        val firstIsLongerThanCurrent = first.length > current.length  
        // 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 digitsMapSum(x: Int)(f: Int => Int): Int = ???  
  
scala> digitsMap(123){ x => x * x }
```

²https://en.wikipedia.org/wiki/Dynamic_programming

```
res0: Int = 14
scala> digitsMap(123){ x => x * x * x }
res1: Int = 36
```

Exercise 2.5.2.2 Compute the **Collatz sequence** c_i as a stream defined by

$$c_0 = n \quad ; \quad c_{k+1} = \begin{cases} \frac{c_k}{2} & \text{if } c_k \text{ is even,} \\ 3c_k + 1 & \text{if } c_k \text{ is odd.} \end{cases}$$

Stop the stream when it reaches 1 (as one would expect³ 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))
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, i.e. given a `Set[Set[Int]]` instead of just three sets a, b, c . 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))
```

³https://en.wikipedia.org/wiki/Collatz_conjecture

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")  
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 $\text{revDigits}(n) == n$. 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 $f(n) = \text{revDigits}(n) + n$ 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 $f(n), f(f(n)), \dots$, 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

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`.

The problem is easy to correct: we can give the pattern variable some other name. Since the pattern variable is locally scoped, it can be renamed within its scope without having to change any other code. A completely equivalent code is

```
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 are “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 available immediately.

Generally, there are four possible ways a value could be available:

Availability	Explanation	Example Scala code
“eager”	computed in advance	<code>val z = f(123)</code>
“lazy”	computed upon first request	<code>lazy val z = f(123)</code>
“on-call”	computed each time it is requested	<code>def z = f(123)</code>
“never”	cannot be computed due to errors	<code>val (z, x) = "abc"</code>

A **lazy value** (declared as `lazy val` in Scala) is computed only when used 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, this is the behavior of a `def` declaration.

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, consider the computation

```
scala> (1L to 1000000000L).sum
res0: Long = 500000000500000000
```

We do not actually need to keep a billion numbers in memory if we only want to compute their sum. Indeed, the computation just shown does *not* keep all the numbers in memory. The same computation fails 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
```

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
```

Even though the range `(1 to 10)` is an iterator, a subsequent `.flatMap` operation creates an intermediate collection that is too large

to fit in memory. 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 `broken` Iterator class The Scala library contains a class called `Iterator`, which has methods such as `Iterator.iterate` and other methods similar to `Stream`. However, `Iterator` actually not a correct iterator because it cannot be treated 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, the value `iter` becomes “empty” after the first use.

This situation is impossible in mathematics: if x is some value, such as 100, and f is some 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 **purity** of the function f .

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 called the **value semantics** of numbers. One says 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 semantics if a given 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` has value semantics and is immutable:

```
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 semantics:

```
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 `.view` also have value semantics:

```
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 semantics, 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 expected, but some of the elements of the given sequence are missing (while other elements are present). It is difficult to debug a program when it 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 cannot be used twice in some expression. Intuitively, we expect code such as `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` is mutable, which breaks the usual ways of mathematical reasoning about code.

An `Iterator` can be converted to a `Stream` using the `.toStream` method. This restores the value semantics, since streams are 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.

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)
einstein: 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")
```

3 The logic of types. I. Disjunctive types

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 a generalization 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,

```
scala> val s = MySockX(10.5, "white", 123)
s: MySockX[Int] = MySockX(10.5,white,123)
```

Because 123 has type `Int`, the type parameter `A` in `MySockX[A]` was automatically set to the type `Int`.

Each time we create a value of type `MySockX`, a specific type will have to be used instead of the type parameter `A`. In other words, we can only create values of types `MySockX[Int]`, `MySockX[String]`, etc. If we want to be explicit, we may write

```
scala> val s = MySockX[String](10.5, "white", "last pair")
s: MySockX[String] = MySockX(10.5,white,last pair)
```

However, we can write code working with `MySockX[A]` **parametrically**, that is, keeping the type parameter `A` in the code. For example, a function that checks whether a sock of type `MySockX[A]` fits my foot can be written as

```
def fitsMe[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 automatically when we apply `fitsMe` to an argument:

```
scala> fitsMe(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 `fitsMe`. 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 `fitsMe`:

```
scala> fitsMe[Int](MySockX(10.5, "blue", List(1,2,3)))
<console>:15: error: type mismatch;
 found   : List[Int]
 required: Int
          fitsMe[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...
```

if these types were not 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]` as a tuple with a single part. (This type is occasionally used in practice.)

The tuple with zero parts also exists and is called `Unit` (rather than “`Tuple0`”). The syntax for the value of the `Unit` type is the empty tuple, `()`. It is clear that there is *only one* value, `()`, of this type; this explains the name “unit”.

At first sight, the `Unit` type may appear to be completely useless: it is a tuple that contains *no data*. It turns out, however, that the `Unit` type is important in functional programming, and it is used as a type *guaranteed* to have only a single distinct value. This chapter will show some examples of using the `Unit` type.

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.
```

Scala has a special 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, if needed, a `case class C[x, y, z]()` with type parameters `x`, `y`, `z`.
- 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. In this book, I will not use `case objects` because `case classes` are more general.

Let us summarize 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()

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 will destructure a value and compute a result:

```
def fitsMe(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 expressions with case classes is similar to the syntax for pattern matching of tuples, except for the presence of the *names* of the case classes. For example, 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 exactly 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 `match / case` expressions with case classes than to use destructuring.

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 “the 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 index of x , or it determines that the array does not contain x . It is convenient if the algorithm could return a single value of a type (call it `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 value *or* generate an error and fail to produce a result. It is then convenient to return a value of type (call it `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 provide exactly what we need here – named tuples with zero, one, two and more parts, and 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(Int)` or a value of type `case class NotFound()`
- `Result` is a value of type `case class Value(x: Int)` or a value 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 exactly the three alternatives described 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
```

3 The logic of types. I. Disjunctive types

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** in this book.

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 creating values of disjunctive types and for using the created values.

Consider the disjunctive type `RootsOfQ` having three 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)
```

Given a value `x:RootsOfQ`, how can we use it, say, as a function argument? The main tool for working with values of disjunctive types is pattern matching with `match / case` expressions. In Chapter 2, we used pattern matching to destructure tuples with syntax such as `{ case (x, y) => ... }`. To use `match / case` expressions with disjunctive types, we may have to write *more than one* `case` pattern in a `match` expression, 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"
```

If we only need to recognize a specific case of a disjunctive type, we can match all other cases with an underscore:

```
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(1.0)` is a possible value of type `RootsOfQ`. When used with disjunctive types, `match / case` expressions will usually contain a 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(_, _)
    def g(x: RootsOfQ): String = x match {
        ^
```

This code defines a *partial* function `g` that can be applied only to values of the form `OneRoot(...)` and will fail 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(-0.6180339887498949,1.618033988749895)

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 other two possibilities have been “filtered out” by the preceding `.filter` operation.

Example 3.2.2.3 Implement binary search returning a `SearchResult`. We will 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
```

When the number is not present, 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()
  }
```

To test:

```
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 integer arithmetic”, where a division by zero or a square root of a negative number will give an error message. Define arithmetic operations directly for values of type `Result[Int]`. When errors occur, abandon further computations.

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 error message.
  case (_, Error(m))        => Error(m)
}
```

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 “lifts” operations on integers to operations on `Result[Int]` types:

```
def do2(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 “lift” 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] = do2(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 new “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 used often. We now look at each of them in turn.

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]
final case object None      extends Option[Nothing]
final case class Some[T](t: T) extends Option[T]
```

3 The logic of types. I. Disjunctive types

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` for the empty case instead of an empty case class, such as `None()`. Since Scala's `case objects` 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**).

An alternative (implemented in libraries such as `scalaz`) 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*. The missing-value case is represented by `None`, while `Some(x)` means that a value `x` is present.

Example 3.2.3.1 Suppose that information about subscribers to a certain online service must contain a name and an email address, but a telephone number is optional. To represent this information, we may 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           // Do nothing.
  case Some(number)  => Some(digitsOf(number))
}
```

Here we used the function `digitsOf` 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           // Do nothing.
  case Some(digits)  => Some(digits.length)
}
```

These examples perform a computation when an `Option` value is non-empty, and leave it empty otherwise. This design pattern is used often. To avoid repeating the code, we can implement this design 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           // Do nothing.
  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`:

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  case None          => None           // Do nothing.
  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, `Some[B](f(a))`, is not needed.

The Scala library implements an equivalent function as a method on the `Option` class, with the syntax `x.map(f)` rather than `fmap(f)(x)`. We can concisely rewrite the previous code using the standard library methods as

```
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`, `.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 design 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
```

3 The logic of types. I. Disjunctive types

```
}.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 design 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)
```

Test it:

```
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 design 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 returns the value if it exists for a given key, and returns `None` if the key is not in the dictionary:

```
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 provides safe by-key access to dictionaries, unlike the direct access method 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` provides 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 design pattern was “if value is present, do a computation, otherwise keep the error”. This design pattern is implemented by the `.map` method on `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 `.filter`, `.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 evaluation 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 possibilities (a correct result or an error condition). Users of these functions will *have to* do pattern matching on the result values. This helps programmers to remember and to

¹These methods are available in Scala 2.12 or a later version.

²Just once in the author’s experience, an out-of-memory condition had to be anticipated in an Android app.

handle all relevant error situations.

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]` can be seen as similar 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]` and `Right[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`:

```
import scala.util.{Try, Success, Failure}

scala> Try("xyz".toInt)
res0: Try[Int] = Failure(java.lang.NumberFormatException: For input string: "xyz")

scala> Try("0002".toInt)
res1: 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 = x.map(y => throw new Exception("ouch"))
y: Try[Int] = Failure(java.lang.Exception: ouch)

scala> val z = x.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. Other 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, so the other 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: 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 definition of the case class `N1` refers to the type `NInt` as if it 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? In general, this question is not easy to answer. For instance, the simple definition

```
final case class Bad(x: Bad)
```

is useless: we can create a value of type `Bad` only if we already have a value of type `Bad`. This is an example of an infinite type recursion. We will never be able to create any values of type `Bad`, which

means that the type `Bad` is void (has no values, like the the special type `Nothing`).

Chapter 14 studies recursive types in more detail. For now, we will look at the main examples of recursive disjunctive types that are *known* to be useful. These examples are lists and trees.

3.3.1 Lists

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 `Listn-1`, define a new case class `Listn` describing a list with one more element of type `A`. So we could define `Listn = (Listn-1, A)`.

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](prev: List0[A], x: A) extends ListI[A]
final case class List2[A](prev: List1[A], x: A) extends ListI[A]
???
// Still need an infinitely long definition.
```

To avoid writing an infinitely long type definition, we need to use a trick. Notice that all definitions of `List1`, `List2`, etc., have a similar form (while `List0` is not similar). We can replace all the definitions `List1`, `List2`, etc., by a single definition if we use the type `ListI[A]` recursively inside the case class:

```
sealed trait ListI[A]           // Inductive definition of a list.
final case class List0[A]()     extends ListI[A]
final case class ListN[A](prev: ListI[A], x: 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 definition `ListN[A]`; or else we would have created an infinite type recursion similar to `case class Bad` shown previously.

Since we obtained the type definition of `ListI` 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(prev, x)` where `x` is of type `A` and `prev` is some previously constructed value of type `ListI[A]`. This represents the induction step, because the case class `ListN` is a named tuple containing `ListI[A]` and `A`. Now, the same consideration recursively applies to constructing the value `prev`, which must be either an empty list or a pair containing another list and an element of type `A`. The assumption that the value `prev: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(List0(), x)`, and a two-element list `ListN(ListN(List0(), x), y)`.

To illustrate writing pattern-matching code using this type, let us implement the method `headOption`:

```
@tailrec def headOption[A]: ListI[A] => Option[A] = {
  case List0()          => None
```

```

    case ListN(List0(), x)      => Some(x)
    case ListN(prev, _)        => headOption(prev)
}

```

The Scala library already defines the type `List[A]` in an equivalent but different way: its case classes are named differently, and the second case class (with the special name `::`) places the value of type `A` before the previously constructed list,

```

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 `head :: tail` instead of `::(head, tail)`. Pattern matching with the standard `List` class 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 use in the syntax `List(1, 2, 3)`. Pattern matching can also use that syntax when convenient:

```

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 induction 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.

3 The logic of types. I. Disjunctive types

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 induction 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)
```

The result is a reversed `.map(x => x*x)`, so we need 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 implementation is shown only as an example. The Scala library implements `.map` for `List` using low-level tricks in order to achieve 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

3 The logic of types. I. Disjunctive types

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:

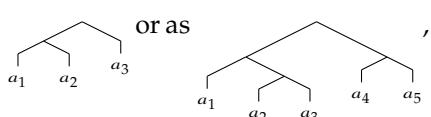
```
def concat[A](xs: NEL[A], ys: NEL[A]): NEL[A] = ???

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, regular-shaped trees, and abstract syntax trees.

Examples of a **binary tree** with leaves of type `A` can be drawn as

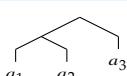


where a_i are some values of type `A`.

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]
```

With this definition, the tree



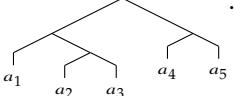
is created by the code expression

```
Branch(Branch(Leaf("a1"), Leaf("a2")), Leaf("a3"))
```

while the expression

```
Branch(Branch(Leaf("a1"), Branch(Leaf("a2"), Leaf("a3"))), Branch(Leaf("a4"), Leaf("a5")))
```

creates the tree



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.
    foldLeft(t2)(r1)(f) // Starting from 'r1', 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 test:

```

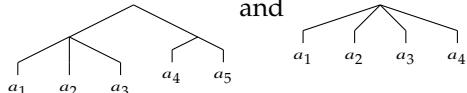
val t: Tree2[String] = Branch(Branch(Leaf("a1"), Leaf("a2")), Leaf("a3"))

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,



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]

```

Exercise 3.3.4.1 Define the function `foldLeft` for a rose tree, using `foldLeft` for the type `NEL`. Type signature and a test:

```

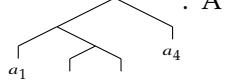
def foldLeft[A, R](t: TreeN[A])(init: R)(f: (R, A) => R): R = ???

scala> foldLeft(Branch(More(Leaf(1), More(Leaf(2), Last(Leaf(3))))))(0)(_ + _)
res0: Int = 6

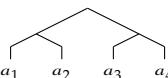
```

3.3.5 Regular-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



regular-shaped tree always branches in the same way at every node until a chosen total depth, e.g.



2 do not branch. The branching number is fixed for a given type of a regular-shaped tree; in this example, the branching number is 2, so it is a regular-shaped *binary* tree.

How can we define a data type representing a regular-shaped binary tree? We need a tree 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 RTree[A]
final case class Leaf[A](x: A) extends RTree[A]
final case class Branch1[A](xs: (A, A)) extends RTree[A]
final case class Branch2[A](xs: ((A, A), (A, A))) extends RTree[A]
??? // Need an infinitely long definition.
```

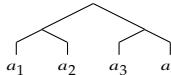
The case `Branch1` describes a regular-shaped tree with total depth 1, the case `Branch2` has total depth 2, and so on. Now, we cannot rewrite this definition as a recursive type because the case classes do not have the same structure. The non-trivial trick is to notice that each `Branchn` case class 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 RTree[A]
final case class Leaf[A](x: A) extends RTree[A]
final case class Branch1[A](xs: Leaf[(A, A)]) extends RTree[A]
final case class Branch2[A](xs: Branch1[(A, A)]) extends RTree[A]
??? // Need an infinitely long definition.
```

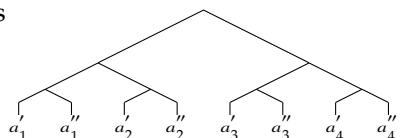
We can now apply the type recursion trick: replace the type `Branchn-1[(A, A)]` in the definition of `Branchn` by the type `RTree[(A, A)]`. This gives the type definition for a regular-shaped binary tree:

```
sealed trait RTree[A]
final case class Leaf[A](x: A) extends RTree[A]
final case class Branch[A](xs: RTree[(A, A)]) extends RTree[A]
```

Since we used some tricks to figure out the definition of `RTree[A]`, let us verify that this definition actually describes the recursive disjunctive type we wanted. The only way to create a structure of type `RTree[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 `RTree` with values of type `(A, A)` instead of A. By the inductive assumption, the previously created `RTree[A]` would have the correct shape. Now, it is clear that if we replace the type parameter A by the pair `(A, A)`, a regular-shaped tree such as



remains regular-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 `RTree[A]` is the correct definition of a regular-shaped binary tree.

Example 3.3.5.1 Define a (non-tail-recursive) `map` function for a regular-shaped binary tree. The required type signature and a test:

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = ???

scala> map(Branch(Branch(Leaf(((1,2),(3,4))))))(_ * 10)
res0: RTree[Int] = Branch(Branch(Leaf(((10,20),(30,40)))))
```

Solution Begin by pattern-matching on the tree:

```
def map[A, B](t: RTree[A])(f: A => B): RTree[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: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)      => Leaf(f(x))
  case Branch(xs)   => ???
}
```

In the inductive step, we are given a previous tree value `xs:RTree[(A, A)]`. It is clear that we need to apply `map` recursively to `xs`. Let us try:

```
def map[A, B](t: RTree[A])(f: A => B): RTree[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 `RTree[(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: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)      => Leaf(f(x))
  case Branch(xs)   => Branch(map(xs){ case (x, y) => (f(x), f(y)) })
}
```

Exercise 3.3.5.2 Using tail recursion, compute the depth of a regular-shaped binary tree of type `RTree`. (An `RTree` of depth n has 2^n leaf values.) The required type signature and a test:

```
@tailrec def depth[A](t: RTree[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 regular-shaped binary tree. The required type signature and a test:

```
@tailrec def foldLeft[A, R](t: RTree[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 $\sqrt{16} * (1 + 2) = 12$; $20 + 1/0 = \text{error}$; and $10 + \sqrt{-1} = \text{error}$.

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 a data structure for an AST is to use a disjunctive type whose cases 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 for “arithmetic sub-language expressions” 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
```

3 The logic of types. I. Disjunctive types

```
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

```
sealed trait Tree
final case class Leaf(x: Int) extends Tree
final case class Branch(x: Tree, y: Tree) extends Tree
```

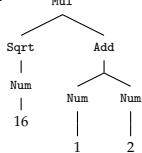
if we rename `Leaf` to `Num` and `Branch` to `Add`. However, the `Arith` type contains four different types of “branches”, some with branching number 1 and others with branching number 2.

This example illustrates the structure of an AST: it is a tree of a general shape, where leaves and branches are chosen from a specified set of allowed possibilities. In this 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 of “safe arithmetic”. 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 destructure 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) }
```

while 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

```
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 methods defined for them.
- 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.

Solution Since there is no information other than the label on each day, we use empty case classes:

```
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.

The often-used code pattern of the form `x => x match { case ... }` can be shortened to the nameless function syntax `{ 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** `:Either[A, B]` only for clarity; it is not required here because 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

3 The logic of types. I. Disjunctive types

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) => ???  
}
```

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 }
```

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:

```
1 def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = {  
2     case None          => Right(None)  
3     case Some(eab)     => eab match {  
4         case Left(a)   => Left(a)  
5         case Right(b)  => Right(Some(b))  
6     }  
7 }
```

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     ...
```

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     ...
```

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,

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`,

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.

Exercise 3.4.2.2 Define a function from `Seq[Seq[CellState]]` to `Int`, counting the total number of cells with zero neighbor bombs shown.

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 $ax + b = 0$, produce 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 ??? // etc.
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(proc2, proc1), proc2)
hello1
hello2
hello2
```

³[https://en.wikipedia.org/wiki/Minesweeper_\(video_game\)](https://en.wikipedia.org/wiki/Minesweeper_(video_game))

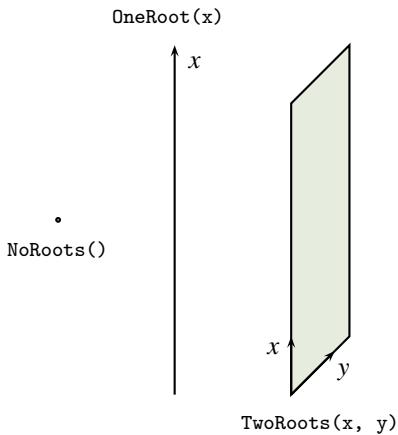


Figure 3.1: The disjoint domain represented by the RootsOfQ type.

Exercise 3.4.2.7 Implement functions that have a given type signature, 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 is guaranteed to have an even number of elements (zero, two, four, etc.). Implement functions `foldLeft` and `map` for `EvenList`.

3.5 Discussion

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 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 (see Figure 3.1). The point, the line, and the plane do not intersect (have no common points); together, they form the domain of the possible roots. Such domains are called **disjoint**. So, the set of real roots of a quadratic equation $x^2 + bx + c = 0$ is a disjoint domain containing three parts.

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

$$\mathbb{R}^0 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2 = \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2 .$$

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 $(\text{OneRoot}, x)_{x \in \mathbb{R}^1}$. Then the domain `Roots0fQ` is expressed as

$$\text{Roots0fQ} = (\text{NoRoots}, u)_{u \in \mathbb{R}^0} \cup (\text{OneRoot}, x)_{x \in \mathbb{R}^1} \cup (\text{TwoRoots}, (x, y))_{(x, y) \in \mathbb{R}^2} .$$

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 labeled union is called a **disjoint union**. Each element of the disjoint union is a pair of the form $(\text{label}, \text{data})$, 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 disjoint unions, we cannot confuse different parts of the union even if their data have the same type, because labels are required to be distinct.

Disjoint unions are not often used in mathematics, but they are needed in software engineering because real-life data often belongs to disjoint domains.

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 `Roots0fQ`. 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 $(\text{NoRoots}, u)_{u \in \mathbb{R}^0}$. 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 $(\text{NoRoots}, u)_{u \in \mathbb{R}^0}$.

We see that case classes with no parts are quite similar to `Unit` except for an added name. For this reason, they can be viewed as “named unit” types.

3.5.2 Disjunctive types in other programming languages

Disjunctive types and the associated pattern matching turns out to be one of the defining features of functional programming 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 the 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 have disjunctive types without other FP features.⁴)

⁴[https://en.wikipedia.org/wiki/Comparison_of_programming_languages_\(basic_instructions\)#Other_types](https://en.wikipedia.org/wiki/Comparison_of_programming_languages_(basic_instructions)#Other_types)

3 The logic of types. I. Disjunctive types

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; } di;
```

The problem is that we cannot determine whether a given value `di` represents an `int`, a `double`, or a `long`. This leads 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. 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 `enum` types were “enriched” with extra data, so that the tuples could be non-empty, we would 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 shorter syntax for disjunctive types⁵ that indeed looks like an “enriched `enum`”:

```
enum RootsOfQ {
  case NoRoots
  case OneRoot(x: Double)
  case TwoRoots(x: Double, y: Double)
}
```

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 define a mathematical notation designed for efficient reasoning about types.

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 Code Has a value of type A ”, where “this code” refers to a particular function in a program. So, the proposition “the function *can* return a value of type `RootsOfQ`” is denoted by $\mathcal{CH}(\text{RootsOfQ})$. We can then write the above sentence about `RootsOfQ` as the logical formula

$$\mathcal{CH}(\text{RootsOfQ}) = \mathcal{CH}(\text{NoRoots}) \vee \mathcal{CH}(\text{OneRoot}) \vee \mathcal{CH}(\text{TwoRoots}) \quad . \quad (3.1)$$

⁵<https://dotty.epfl.ch/docs/reference/enums/adts.html>

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

$$\mathcal{CH}(\text{TwoRoots}) = \mathcal{CH}(\text{Double}) \wedge \mathcal{CH}(\text{Double}) .$$

Formal logic admits the simplification

$$\mathcal{CH}(\text{Double}) \wedge \mathcal{CH}(\text{Double}) = \mathcal{CH}(\text{Double}) .$$

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

$$\mathcal{CH}(\text{Data3}) = \mathcal{CH}(\text{Int}) \wedge \mathcal{CH}(\text{String}) \wedge \mathcal{CH}(\text{Double}) . \quad (3.2)$$

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 a logical negation $\neg A$ (“ A is not true”) or, equivalently, a logical implication $A \Rightarrow B$ (“if A is true than B is true”). It turns out that the logical implication $A \Rightarrow B$ is related to the function type `A => B`. In Chapter 4, we will study function types in depth.

⁶These types are also called “variants”, “sum types”, “co-product types”, and “tagged union types”.

Part II

Intermediate level

4 The logic of types. II. Higher-order 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 logger functions 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 so 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` were captured.

4.1.2 Curried and uncurried functions

Reasoning mathematically about the 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, both function applications `logWith("INFO")` and `logWith("INFO")("hello")` are 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 $(A \Rightarrow B)$, 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") }
```

We will sometimes enclose nameless functions in parentheses or curly braces to improves clarity.

Line 3 in the above code shows that the arrow symbol `=>` groups to the right within the *code* of nameless functions: `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 adopted for two reasons. First, the code `x => y => z` visually corresponds to the curried function's type signature `A => B => C`, which uses the same syntax convention. Second, the syntax `(x => y) => z` could not work for a nameless function because `(x => y)` is not a valid pattern expression for the function's argument; patterns cannot match arbitrary *functions* of type `A => B`. So `x => (y => z)` is the only possible way of inserting parentheses into `x => y => z`.

Although the code `(x => y) => z` is invalid, a type expression `(A => B) => C` is valid. A nameless function of type `(A => B) => C` is written as `f => expr` where `f: A => B` is its argument and `expr` its 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 the functions `f1` and `f2` is different:

```
scala> f1(20)(4)
res0: Int = 16

scala> f2(20, 4)
res1: Int = 16
```

The main difference between the usage of `f1` and `f2` is that `f2` must be applied *at once* to both arguments, while `f1` applied to just the first argument, `20`. 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 some but not all possible arguments is called a **partial application**. Applying a curried function to all possible arguments is called a **saturated application**.

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) }
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 functions `f1` and `f2` are **computationally equivalent** in the sense that given `f1` we can reconstruct `f2` and vice versa:

```
def f2new(x: Int, y: Int): Int = f1(x)(y)           // f2new is equal to f2
def finew: Int => Int = { x => y => f2(x, y) } // finew 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 computational equivalence of the functions `f1` and `f2` is not *equality* – these functions are *different*; but one of them can be easily reconstructed from the other if necessary.

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 computationally 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 easy to implement, as we will show in Section 4.2.1.

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 the `map` and `filter` methods with type signatures

```
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 only because we need to work 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 definition 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 setting type parameters equal to specific types (e.g. `A=Int, B=String`) in the type signature of the function, similarly to how setting value parameters means substituting specific values into the body of the function.

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* types in the type signature of a function by type parameters.

A function is **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 type signature or in their body. A fully parametric function does not use any information about its argument types, other than assuming that the arguments' types correctly match the type signature.

What kind of functions are fully parametric? To build an intuition for that, let us compare these two functions having 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.
}

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, with no changes in the code of the function:

```
def swap[A, B](p: (A, B)): (B, A) = p match {
  case (x, y) => (y, x)
}
```

Generalizing `swap` to a fully parametric function is possible because the operation of swapping two parts of a tuple `(A, B)` works in the same way for all types `A, B`. 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` and cannot be generalized to an arbitrary type parameter `A` instead of `Double`. So, `cos_sin` cannot be generalized to a fully parametric function.

The `swap` operation for pairs is already defined in the Scala library:

```
scala> (1, "abc").swap
res0: (String, Int) = (abc,1)
```

Other swapping functions can be defined for tuples with more elements, e.g.

```
def swap12[A,B,C]: ((A, B, C)) => (B, A, C) = {
  case (x, y, z) => (y, x, z)
}
```

The Scala syntax requires the 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 of type

```
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 curry / uncurry conversions.

The identity function (available in the Scala library as `identity[T]`) is

```
def id[T]: T => T = (t => t)
```

The constant function (available in the Scala library as `Function.const`) takes an argument `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 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:Int` and get a result `f(x)` of type `Double`. We can then apply `g` to that result and obtain a `String` value `g(f(x))`. The transformation from the original integer `x:Int` to the 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, this operation 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: Int => String = <function1>

scala> h(40)
res36: String = Result x = 45.67
```

The Scala compiler derives the type of `h` automatically as `Int => String`.

The forward composition is denoted by `;` (pronounced “before”) and can be defined as

$$f ; g \triangleq (x \Rightarrow g(f(x))) \quad . \quad (4.1)$$

The symbol `△` means “is defined as”.

We could write the 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` is applied to the result. Using the symbol `o` (pronounced “after”) for this operation, we can write

$$f \circ g \triangleq (x \Rightarrow f(g(x))) \quad . \quad (4.2)$$

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, let us write our own code for converting curried functions to uncurried:

```
def uncurry[A, B, R](f: A => B => R): ((A, B)) => R = {
  case (a, b) => f(a)(b)
}
```

We conclude from these examples 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 $A \Rightarrow B \Rightarrow R$, which are type expressions made up from type parameters. But fully parametric functions do not use values of specific types such as `Int` or `String`.

The property of being fully parametric is also called **parametricity**. In some programming languages, functions with type parameters are called “generic”.

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 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 mathematical operation. Let us write these laws rigorously as mathematical equations and prove them.

Proofs in the forward notation The composition of the identity function with an arbitrary function f can be $\text{id} ; f$ with the identity function to the left of f , or $f ; \text{id}$ with the identity function to the right of f . In both cases, the result must be equal to the function f . The resulting two laws are

$$\begin{aligned}\text{left identity law of composition : } & \text{id} ; f = f , \\ \text{right identity law of composition : } & f ; \text{id} = f .\end{aligned}$$

To show that these laws always 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 the laws to have consistently matching types.

The laws must hold for an arbitrary function f . So we may assume that f has the type signature $A \Rightarrow B$, where A and B are arbitrary type parameters. Consider the left identity law. The function $(\text{id} ; f)$ is, by definition (4.1), a function that takes an argument x , applies id to that x , and then applies f to the result:

$$\text{id} ; f = (x \Rightarrow f(\text{id}(x))) .$$

If f has type $A \Rightarrow B$, its argument must be of type A , or else the types will not match. Therefore, the identity function must have type $A \Rightarrow A$, and the argument x must have type A . With these choices of the type parameters, the function $(x \Rightarrow f(\text{id}(x)))$ will have type $A \Rightarrow B$, as it must since the right-hand side of the law is f . We add type annotations to the code as *superscripts*,

$$\text{id}^A ; f^{A \Rightarrow B} = (x^A \Rightarrow f(\text{id}(x)))^{A \Rightarrow B} .$$

In the Scala syntax, this formula may be written as

```
id[A] andThen (f: A => B) == { x: A => f(id(x)) }: A => B
```

It is quicker to write code in the mathematical notation than in the Scala syntax. 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 $A \Rightarrow A$, we can write id^A or equivalently (but more verbosely) $\text{id}^{A \Rightarrow A}$ to denote that function.

Since $\text{id}(x) = x$ by definition of the identity function, we find that

$$\text{id} ; f = (x \Rightarrow f(\text{id}(x))) = (x \Rightarrow f(x)) = f .$$

The last step works since $x \Rightarrow f(x)$ is a function taking an argument x and applying f to that argument; i.e. $x \Rightarrow f(x)$ is the same function as f .

Now consider the right identity law:

$$f \circ \text{id} = (x \Rightarrow \text{id}(f(x))) \quad .$$

To make the types match, assume that $f:A \Rightarrow B$. Then x must have type A , and the identity function must have type $B \Rightarrow B$. The result of $\text{id}(f(x))$ will also have type B . With these choices of type parameters, all types match:

$$f:A \Rightarrow B \circ \text{id}^B = (x:A \Rightarrow \text{id}(f(x)))^{A \Rightarrow B} \quad .$$

Since $\text{id}(f(x)) = f(x)$, we find that

$$f \circ \text{id} = (x \Rightarrow f(x)) = f \quad .$$

In this way, we have demonstrated that both identity laws hold.

The associativity law is written as an equation like this:

$$\text{associativity law of composition : } (f \circ g) \circ h = f \circ (g \circ h) \quad . \quad (4.3)$$

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 exist. If f has type $A \Rightarrow B$ for some type parameters A and B , then the argument of g must be of type B ; so we can choose $g:B \Rightarrow C$, where C is another type parameter. The composition $f \circ g$ has type $A \Rightarrow C$, so h must have type $C \Rightarrow D$ for some type D . Assuming the types as $f:A \Rightarrow B$, $g:B \Rightarrow C$, and $h:C \Rightarrow D$, we find that the types in all the compositions $f \circ g$, $g \circ h$, $(f \circ g) \circ h$, and $f \circ (g \circ h)$ match. We can rewrite Eq. (4.3) with type annotations,

$$(f:A \Rightarrow B \circ g:B \Rightarrow C) \circ h:C \Rightarrow D = f:A \Rightarrow B \circ (g:B \Rightarrow C \circ h:C \Rightarrow D) \quad . \quad (4.4)$$

Having checked the types, we are ready to prove the associativity law. We note that both sides of the law (4.4) are functions of type $A \Rightarrow D$. To prove that two functions are equal means to prove that they always return the same results when applied to the same arguments. So we need to apply both sides of Eq. (4.4) to an arbitrary value $x:A$. Using the definition (4.1) of the forward composition,

$$(f \circ g)(x) = g(f(x)) \quad ,$$

we find

$$\begin{aligned} ((f \circ g) \circ h)(x) &= h((f \circ g)(x)) = h(g(f(x))) \quad , \\ (f \circ (g \circ h))(x) &= (g \circ h)(f(x)) = h(g(f(x))) \quad . \end{aligned}$$

Both sides of the law are now equal when applied to an arbitrary value x .

Because of the associativity law, we do not need parentheses when writing the expression $f \circ g \circ h$. The function $(f \circ g) \circ h$ is the same as $f \circ (g \circ h)$.

In the proof, we have omitted the type annotations since we already checked that the types match. Checking the types beforehand allows us to write shorter proofs.

Proofs in the backward notation This book uses the **forward notation** $f \circ g$ for writing function compositions, rather than the backward notation $g \circ f$. If necessary, all equations can be automatically converted from one notation to the other by reversing the order of compositions because

$$f \circ g = g \circ f$$

for any functions $f:A \Rightarrow B$ and $g:B \Rightarrow C$. 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

$$f^{A \Rightarrow B} \circ \text{id}^{A \Rightarrow A} = f^{A \Rightarrow B} \quad , \quad \text{id}^{B \Rightarrow B} \circ f^{A \Rightarrow B} = f^{A \Rightarrow B} \quad .$$

We can apply both sides of the laws to an arbitrary value x^A . For the left identity law, we find from definition (4.2) that

$$f \circ \text{id} = (x \Rightarrow f(\text{id}(x))) = (x \Rightarrow f(x)) = f \quad .$$

Similarly for the right identity law,

$$\text{id} \circ f = (x \Rightarrow \text{id}(f(x))) = (x \Rightarrow f(x)) = f \quad .$$

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 $f^{A \Rightarrow B}$. The types in $g \circ f$ match only when $g^{B \Rightarrow C}$, and then $g \circ f$ is of type $A \Rightarrow C$. The type of h must be $h^{C \Rightarrow D}$ for the types in $h \circ (g \circ f)$ to match. We can write the associativity law with type annotations as

$$h^{C \Rightarrow D} \circ (g^{B \Rightarrow C} \circ f^{A \Rightarrow B}) = (h^{C \Rightarrow D} \circ g^{B \Rightarrow C}) \circ f^{A \Rightarrow B} \quad . \quad (4.5)$$

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 easier to check that types match in Eq. (4.4) than in Eq. (4.5).

4.2.3 Example: A function that violates parametricity

Fully parametric functions should not make any decisions based on the actual types of arguments. As an example of an *incorrect* implementation of a fully parametric function, consider the following “fake identity” function:

```
def fakeId[A]: A => A = { // Special code for A = Int:
  case x: Int    => (x - 1).asInstanceOf[A]
  case x          => x           // Common code for all other types A.
}
```

This function’s type signature is the same as that of `id[A]`, and its behavior is the same for all types `A` except for `A = Int`:

```
scala> fakeId("abc")
res0: String = abc

scala> fakeId(true)
res1: Boolean = true

scala> fakeId(0)
res2: Int = -1
```

While Scala allows us to write this kind of code, the resulting function does not appear to be useful. In any case, `fakeId` is not a fully parametric function.

The identity laws of composition will not hold if we use `fakeId[A]` instead of the correct function `id[A]`. For example, consider the composition of `fakeId` with a simple function `f_1` defined by

```
def f_1: Int => Int = { x => x + 1 }
```

The composition (`f_1 andThen fakeId`) will have type `Int => Int`. Since `f_1` has type `Int => Int`, Scala will automatically set the type parameter `A = Int` in `fakeId[A]`,

```
scala> def f_2 = f_1 andThen fakeId
f_2: Int => Int
```

The identity law says that $f_2 = f_1 \circ id = f_1$. But we can check that `f_1` and `f_2` are not the same:

```
scala> f_1(0)
res3: Int = 1
```

```
scala> f_2(0)
res4: Int = 0
```

It is important that we are able to detect a violation of parametricity by checking whether some equation holds, without need to examine the code of the function `fakeId`. In this book, we will always formulate the desired properties of functions through equations or “laws”. To prove the laws, we will need to perform symbolic calculations similar to the proofs in Section 4.2.2. These calculations are **symbolic** in the sense that we were manipulating symbols such as x , f , g , and h without substituting any specific values for these symbols but using only the general properties of functions. In the next section, we will get more experience with such calculations.

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. Nameless functions are evaluated in the same way. For example, applying the nameless function $x \Rightarrow x + 10$ to an integer 2, we substitute 2 instead of x in “ $x + 10$ ” and get “ $2 + 10$ ”, which we then evaluate to 12. The computation is written like this,

$$(x \Rightarrow x + 10)(2) = 2 + 10 = 12 \quad .$$

To run this computation in Scala, we need to add a type annotation:

```
scala> ((x: Int) => x + 10)(2)
res0: Int = 12
```

Curried function applications such as $f(x)(y)$ are rarely used in mathematics, so we need to gain some experience working with them.

Consider a curried nameless function being applied to arguments, such as $(x \Rightarrow y \Rightarrow x - y)(20)(4)$, and compute the result of this function application. Begin with the argument 20; applying a nameless function of the form $(x \Rightarrow \dots)$ to 20 means to substitute $x = 20$ into the body of the function. After that substitution, we obtain the expression $y \Rightarrow 20 - y$, which is again a nameless function. Applying that function to the remaining argument 4 means substituting $y = 4$ into the body of $y \Rightarrow 20 - y$. We get the expression $20 - 4$, which equals 16. Check the result with Scala:

```
scala> ((x: Int) => (y: Int) => x - y)(20)(4)
res1: Int = 16
```

Applying a curried function such as $x \Rightarrow y \Rightarrow z \Rightarrow expr(x,y,z)$ to three curried arguments 10, 20, and 30 means to substitute $x = 10$, $y = 20$, and $z = 30$ into the expression `expr`. 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 the arguments.

We see that the right grouping of the function arrow \Rightarrow 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: $x:\text{Int} \Rightarrow x + 10$ instead of the code $((x:\text{Int}) \Rightarrow x + 10)$.

The symbolic evaluation of the Scala code $((x:\text{Int}) \Rightarrow (y:\text{Int}) \Rightarrow x - y)(20)(4)$ can be written as

$$\begin{aligned} & (\underline{x:\text{Int}} \Rightarrow \underline{y:\text{Int}} \Rightarrow \underline{x - y})(20)(4) \\ \text{apply function and substitute } x = 20 : & = (\underline{y:\text{Int}} \Rightarrow 20 - \underline{y})(4) \\ \text{apply function and substitute } y = 4 : & = 20 - 4 = 16 . \end{aligned}$$

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 so 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 $f \Rightarrow f(2)$ to a nameless function $x \Rightarrow x + 4$:

$$\begin{aligned} & (\underline{f \Rightarrow f(2)})(\underline{x \Rightarrow x + 4}) \\ \text{substitute } f = (x \Rightarrow x + 4) : & = (\underline{x \Rightarrow x + 4})(2) \\ \text{substitute } x = 2 : & = 2 + 4 = 6 . \end{aligned}$$

In the nameless function $f \Rightarrow f(2)$, the argument f has to be itself a function, otherwise the expression $f(2)$ 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 $f(2)$ is 4, an integer. We conclude that f must have type $\text{Int} \Rightarrow \text{Int}$, or else the types will not match. To verify this result in Scala, we need to specify a type annotation for f :

```
scala> ((f: Int => Int) => f(2))(x => x + 4)
res2: Int = 6
```

No type annotation is needed for $x \Rightarrow x + 4$ since the Scala compiler already knows the type of f and can figure out that x in $x \Rightarrow x + 4$ must have type Int .

To summarize the syntax conventions for curried nameless functions:

- Function expressions group everything to the right: $x \Rightarrow y \Rightarrow z \Rightarrow e$ means $x \Rightarrow (y \Rightarrow (z \Rightarrow e))$.
- Function applications group everything to the left, so $f(x)(y)(z)$ means $((f(x))(y))(z)$.
- Function applications group stronger than infix operations, so $x + f(y)$ means $x + (f(y))$, as usual in mathematics.

Here are some more examples of performing function applications symbolically. Types are omitted for brevity; every non-function value is of type Int .

$$\begin{aligned} (x \Rightarrow x * 2)(10) &= 10 * 2 = 20 . \\ (p \Rightarrow z \Rightarrow z * p)(t) &= (z \Rightarrow z * t) . \\ (p \Rightarrow z \Rightarrow z * p)(t)(4) &= (z \Rightarrow z * t)(4) = 4 * t . \end{aligned}$$

Some results of these computation are integer values such as 20; other results are nameless functions such as $z \Rightarrow z * t$. 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 ($g \Rightarrow g(2)$) as an argument:

$$(f \Rightarrow p \Rightarrow f(p))(g \Rightarrow g(2)) \quad (4.6)$$

substitute $f = (g \Rightarrow g(2))$: $= p \Rightarrow (g \Rightarrow g(2))(p)$

substitute $g = p$: $= p \Rightarrow p(2)$.

The result of this expression is a function $p \Rightarrow p(2)$ that will apply its argument p to the value 2. A possible value for p is the function $x \Rightarrow x + 4$. So, let us apply Eq. (4.6) to that function:

$$(f \Rightarrow p \Rightarrow f(p))(g \Rightarrow g(2))(x \Rightarrow x + 4)$$

use Eq. (4.7) : $= (p \Rightarrow p(2))(x \Rightarrow x + 4)$

substitute $p = (x \Rightarrow x + 4)$: $= (x \Rightarrow x + 4)(2)$

substitute $x = 2$: $= 2 + 4 = 6$.

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 $f \Rightarrow p \Rightarrow f(p)$ is being applied to the arguments $f = (g \Rightarrow g(2))$ and $p = (x \Rightarrow x + 4)$. So, the argument f in $f \Rightarrow p \Rightarrow f(p)$ must be a function that takes p as an argument.

The variable x in $x \Rightarrow x + 4$ must be of type `Int`, or else we cannot add x to 4. Thus, the type of the expression $x \Rightarrow x + 4$ is `Int` \Rightarrow `Int`, and so must be the type of the argument p . We write $p: \text{Int} \Rightarrow \text{Int}$.

Finally, we need to make sure that the types match in the function $f \Rightarrow p \Rightarrow f(p)$. Types match in $f(p)$ if the type of f 's argument is the same as the type of p , which is `Int` \Rightarrow `Int`. So f 's type must be $(\text{Int} \Rightarrow \text{Int}) \Rightarrow A$ for some type A . Since in our example $f = (g \Rightarrow g(2))$, types match only if g has type `Int` \Rightarrow `Int`. But then $g(2)$ has type `Int`, and so we must have $A = \text{Int}$. Thus, the type of f is $(\text{Int} \Rightarrow \text{Int}) \Rightarrow \text{Int}$. We know enough to write the Scala code now:

```
scala> ((f: (\text{Int} \Rightarrow \text{Int}) \Rightarrow \text{Int}) \Rightarrow p \Rightarrow f(p))(g \Rightarrow g(2))(x \Rightarrow x + 4)
res6: \text{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 just makes the code longer.

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 $f \circ g \circ h$. This worked because the composition operation `andThen` (denoted by the symbol `;`) is fully parametric. Given a fully parametric function, it is often possible to derive the most general type signature that matches the body of that function. The same type-matching 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:

$$\begin{aligned}\text{const}^{C,X} &\triangleq (c:C \Rightarrow _)^X \Rightarrow c) \quad , \\ \text{id}^A &\triangleq (a:A \Rightarrow a) \quad .\end{aligned}$$

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 $A \Rightarrow A$, where A is an arbitrary type so far. So, the type parameter C in $\text{const}^{C,X}$ must be equal to $A \Rightarrow A$:

$$C = A \Rightarrow A \quad .$$

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 $X \Rightarrow C$, which equals $X \Rightarrow A \Rightarrow A$. In this way, we find

$$\text{const}^{A \Rightarrow A, X}(\text{id}^A) : X \Rightarrow A \Rightarrow A \quad .$$

The types A and X can be arbitrary. The type $X \Rightarrow A \Rightarrow A$ 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:

$$\begin{aligned}\text{const}(\text{id}) \\ \text{definition of const : } &= (c \Rightarrow x \Rightarrow c)(\text{id}) \\ \text{apply function, substitute } c = \text{id} : &= (x \Rightarrow \text{id}) \\ \text{definition of id : } &= (x \Rightarrow a \Rightarrow a) \quad .\end{aligned}$$

The function $(x \Rightarrow a \Rightarrow a)$ takes an argument $x:X$ and returns the identity function $a:A \Rightarrow a$. It is clear that the argument x is ignored by this function, so we can rewrite it equivalently as

$$\text{const}(\text{id}) = (_ \Rightarrow a \Rightarrow a) \quad .$$

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 example, if the function `f` is `{ x => x + 3 }`, the result of `twice(f)` should be equal to the function `x => x + 6`. 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. To test:

```
scala> val g = twice(x => x + 3)
g: Int => Int = <function1>

scala> g(10)
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, \dots , we need to determine the most general type that matches the function body. The function body is the expression $x \Rightarrow f(f(x))$. Assume that x has type A ; for types to match in the sub-expression $f(x)$, we need f to have type $A \Rightarrow B$ 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 $f:A \Rightarrow B$. This is consistent only if $A = B$. In this way, $x:A$ implies $f:A \Rightarrow A$, and the expression $x \Rightarrow f(f(x))$ has type $A \Rightarrow A$. 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

$$\text{twice}^A \triangleq f^{A \Rightarrow A} \Rightarrow x^A \Rightarrow f(f(x)) = (f^{A \Rightarrow A} \Rightarrow f \circ f) . \quad (4.8)$$

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 $(A \Rightarrow A) \Rightarrow A \Rightarrow A$ have been “inferred” from the code $f \Rightarrow x \Rightarrow f(f(x))$.

Example 4.3.2.3 Consider the fully parametric function `twice` defined in Example 4.3.2.2. What is the type of `twice(twice)`, and what computation does it perform? Test your answer on the expression `twice(twice[Int])(x => x+3)(10)`. What are the type parameters in that expression?

Solution Begin by figuring out the required type of `twice(twice)`. We introduce unknown type parameters as `twice[A](twice[B])`. The function `twice[A]` of type $(A \Rightarrow A) \Rightarrow A \Rightarrow A$ can be applied to the argument `twice[B]` only if `twice[B]` has type $A \Rightarrow A$. But `twice[B]` is of type $(B \Rightarrow B) \Rightarrow B \Rightarrow B$. Since the symbol \Rightarrow groups to the right, we have

$$(B \Rightarrow B) \Rightarrow B \Rightarrow B = (B \Rightarrow B) \Rightarrow (B \Rightarrow B) .$$

This can match with $A \Rightarrow A$ only if we set $A = (B \Rightarrow B)$. So the most general type of `twice(twice)` is

$$\text{twice}^{B \Rightarrow B}(\text{twice}^B) : (B \Rightarrow B) \Rightarrow B \Rightarrow B . \quad (4.9)$$

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 will use the curried definition of `twice` given by Eq. (4.8), which shows that `twice(f) = f ∘ f`. Substituting that into the expression `twice(twice)`, we find

$$\begin{aligned} \text{twice}(\text{twice}) &= \text{twice} \circ \text{twice} \\ \text{expand function composition} : &= f \Rightarrow \text{twice}(\text{twice}(f)) . \\ \text{definition of } \text{twice}(f) : &= f \Rightarrow \text{twice}(f \circ f) \\ \text{definition of } \text{twice} : &= f \Rightarrow f \circ f \circ f \circ f . \end{aligned}$$

So `twice(twice)` is a function that applies its (function-typed) argument *four* times.

The type parameters follow from Eq. (4.9) with $A = \text{Int}$ and can be written as `twice[Int] → Int(twice[Int])`, or in Scala syntax, `twice[Int] => Int(twice[Int])`. To test, we need to write at least one type parameter in the code, or else Scala cannot infer the types correctly:

```
scala> twice(twice[Int])(x => x + 3)(100) // _ + 3 + 3 + 3 + 3
res0: Int = 112

scala> twice[Int => Int](twice)(x => x + 3)(100)
res1: Int = 112
```

Example 4.3.2.4 Infer the type signature for the fully parametric function

```
def p[...]: ... = { f => f(2) }
```

Can the types possibly match in the expression $p(p)$?

Solution In the nameless function $f \Rightarrow f(2)$, the argument f must be itself a function with an argument of type `Int`, otherwise the sub-expression $f(2)$ makes no sense. So, types will match if f has type `Int` \Rightarrow `Int` or `Int` \Rightarrow `String` or similar. The most general case is when f has type `Int` \Rightarrow A , where A is an arbitrary type (i.e. a type parameter). The type A will then be the (so far unknown) type of the value $f(2)$. Since nameless function $f \Rightarrow f(2)$ has an argument f of type `Int` \Rightarrow A and the result of type A , we find that the type of p must be $(\text{Int} \Rightarrow A) \Rightarrow A$. 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]: (\text{Int} \Rightarrow A) \Rightarrow A = { f => f(2) }
```

To answer the question about the expression $p(p)$, we begin by writing that expression with new type parameters as $p[A](p[B])$. Then we try to choose A and B so that the types match in that expression. Does the type of $p[B]$, which is $(\text{Int} \Rightarrow B) \Rightarrow B$, match the type of the argument of $p[A]$, which is $\text{Int} \Rightarrow A$, with some choice of A and B ? A function type $P \Rightarrow Q$ matches $x \Rightarrow y$ only if $P = x$ and $Q = y$. So $(\text{Int} \Rightarrow B) \Rightarrow B$ can match $\text{Int} \Rightarrow A$ only if $\text{Int} \Rightarrow B$ matches Int and if $B = A$. But it is impossible for $\text{Int} \Rightarrow B$ to match Int , no matter how we choose B .

We conclude that types cannot be chosen consistently in $p[A](p[B])$. Such expressions contain a type error and are rejected by the Scala compiler. One also says that the expression $p(p)$ is **not well-typed**, or does not **typecheck**.

For any given code expression containing only function expressions, one can always find the most general type that makes all functions match their arguments, unless the expression does not typecheck. The Damas-Hindley-Milner algorithm¹ 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 code 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 applications symbolically.
- Infer the most general type for a given code expression.
- Convert functions to a fully parametric form when possible.

The following solved 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`:

¹https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system#Algorithm_W

Code notation	Scala syntax	Comments
$x:A$	<code>x:A</code>	a value or an argument of type A
$f:A \Rightarrow B$	<code>f: A => B</code>	a function of type A \Rightarrow B
$x:\text{Int} \Rightarrow f(x)$	<code>{ x: Int => f(x)}</code>	nameless function with argument x
$f^{A,B} \triangleq \dots$	<code>def f[A, B] = ...</code>	define a function with type parameters
id^A , also $\text{id}^{A \Rightarrow A}$	<code>identity[A]</code>	the standard identity function
$A \Rightarrow B \Rightarrow C$	<code>A => B => C</code>	type of a curried function
$f \circ g$	<code>f andThen g</code>	forward composition of functions
$g \circ f$	<code>g compose f</code>	backward composition of functions

Table 4.1: Mathematical notation for symbolic computations with code.

```
def converge[X](f: X => X, x0: X, cond: X => Boolean): X = ???
```

Solution We create an iterator that keeps applying the function f , and use `.find` to stop the sequence when the condition first holds:

```
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. But here it is safe to call `.get`, 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 4.4.2.7 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. The iteration function f is

$$f(x) = \frac{1}{2} \left(x + \frac{q}{x} \right).$$

We iterate $f(x)$ starting with $x_0 = q/2$ until we obtain a given precision:

```
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 a function that we can write as $\{ z \Rightarrow z + x \}$. Since functions are values, we can directly 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 as a `val`, we first convert the type signature of `add_x` to the equivalent type expression $\text{Int} \Rightarrow \text{Int} \Rightarrow \text{Int}$. 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`. It is not necessary to specify the type of the arguments x and z because we already specified the type $\text{Int} \Rightarrow \text{Int} \Rightarrow \text{Int}$ for the value `add_x_v`.

Example 4.4.1.3 Using both `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 a prime number. Use the function `is_prime` defined in 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 $\text{Int} \Rightarrow \text{Int}$. 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 `is_prime` applied to $f(x)$. A simple solution is

```
def prime_f(f: Int => Int)(x: Int): Boolean = is_prime(f(x))
```

To implement the same function as a `val`, rewrite its type signature as

```
val prime_f: (Int => Int) => Int => Boolean = ???
```

(The parentheses around $\text{Int} \Rightarrow \text{Int}$ are mandatory since $\text{Int} \Rightarrow \text{Int} \Rightarrow \text{Int} \Rightarrow \text{Boolean}$ would be a completely different type.) The implementation is

```
val prime_f: (Int => Int) => Int => Boolean = { f => x => is_prime(f(x)) }
```

The code `is_prime(f(x))` is a forward composition of the functions f and `is_prime`, so we can write

```
val prime_f: (Int => Int) => Int => Boolean = (f => f andThen is_prime)
```

A nameless function of the form $f \Rightarrow f.\text{something}$ 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 is_prime)
```

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 $A \Rightarrow \text{Boolean}$. 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 return $f(x)$ 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 $f:A \Rightarrow B$ and $g:A \Rightarrow C$, yielding the type signature

$$\text{choice}(x:A, p:A \Rightarrow \text{Boolean}, f:A \Rightarrow B, g:A \Rightarrow C) .$$

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 $g \Rightarrow g(f)$, the curried argument g must itself be a function, because it is being applied to f as $g(f)$. So we assign types as $f:A \Rightarrow g:A \Rightarrow B \Rightarrow g(f)$, 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 will probably 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 $C \Rightarrow (C \Rightarrow D) \Rightarrow D$. So we must set the type parameter A as

$$A = C \Rightarrow (C \Rightarrow D) \Rightarrow D .$$

The type of `q(q)` becomes

$$q^{A,B}(q^{C,D}) : ((C \Rightarrow (C \Rightarrow D) \Rightarrow D) \Rightarrow B) \Rightarrow B , \\ \text{where } A = C \Rightarrow (C \Rightarrow D) \Rightarrow D .$$

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 $r \triangleq q(q)$ for brevity; then, as we just found, r has type $((C \Rightarrow (C \Rightarrow D) \Rightarrow D) \Rightarrow B) \Rightarrow B$. 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

$$E = ((C \Rightarrow (C \Rightarrow D) \Rightarrow D) \Rightarrow B) \Rightarrow B .$$

Other than that, the type parameters are arbitrary. The type of the expression `q(q(q))` is $(E \Rightarrow F) \Rightarrow F$. We conclude that the most general type of `q(q(q))` is

$$q^{E,F}(q^{A,B}(q^{C,D})) : (((C \Rightarrow (C \Rightarrow D) \Rightarrow D) \Rightarrow B) \Rightarrow B) \Rightarrow F \Rightarrow F , \\ \text{where } A = C \Rightarrow (C \Rightarrow D) \Rightarrow D \\ \text{and } E = ((C \Rightarrow (C \Rightarrow D) \Rightarrow D) \Rightarrow B) \Rightarrow B .$$

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 = 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

$$(f \Rightarrow g \Rightarrow g(f))(f \Rightarrow g \Rightarrow g(f))(f \Rightarrow f(10))$$

and simplify the code by symbolic calculations.

Solution The given expression is a curried function $f \Rightarrow g \Rightarrow g(f)$ 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:

$$(f \Rightarrow g \Rightarrow g(f))(x \Rightarrow y \Rightarrow y(x))(h \Rightarrow h(10)) . \quad (4.10)$$

As we have seen in Example 4.4.1.5, the sub-expression $f \Rightarrow g \Rightarrow g(f)$ is typed as $f:A \Rightarrow g:A \Rightarrow B \Rightarrow g(f)$, where A and B are some type parameters. The sub-expression $x \Rightarrow y \Rightarrow y(x)$ is the same function as $f \Rightarrow g \Rightarrow g(f)$ but with possibly different type parameters, say, $x:C \Rightarrow y:C \Rightarrow D \Rightarrow y(x)$. The types A, B, C, D are so far unknown.

Finally, the variable h in the sub-expression $h \Rightarrow h(10)$ must have type $\text{Int} \Rightarrow E$, where E is another type parameter. So, the sub-expression $h \Rightarrow h(10)$ is a function of type $(\text{Int} \Rightarrow E) \Rightarrow E$.

The types must match in the entire expression (4.10):

$$(f:A \Rightarrow g:A \Rightarrow B \Rightarrow g(f))(x:C \Rightarrow y:C \Rightarrow D \Rightarrow y(x))(h:\text{Int} \Rightarrow E \Rightarrow h(10)) . \quad (4.11)$$

It follows that f must have the same type as $x \Rightarrow y \Rightarrow y(x)$, while g must have the same type as $h \Rightarrow h(10)$. The type of g , which we know as $A \Rightarrow B$, will match the type of $h \Rightarrow h(10)$, which we know as $(\text{Int} \Rightarrow E) \Rightarrow E$, only if $A = (\text{Int} \Rightarrow E)$ and $B = E$. It follows that f has type $\text{Int} \Rightarrow E$. At the same time, the type of f must match the type of $x \Rightarrow y \Rightarrow y(x)$, which is $C \Rightarrow (C \Rightarrow D) \Rightarrow D$. This can work only if $C = \text{Int}$ and $E = (C \Rightarrow D) \Rightarrow D = (\text{Int} \Rightarrow D) \Rightarrow D$.

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 undetermined (i.e. arbitrary), while the type parameters A, B, C, E are expressed as

$$A = \text{Int} \Rightarrow (\text{Int} \Rightarrow D) \Rightarrow D , \quad (4.12)$$

$$B = E = (\text{Int} \Rightarrow D) \Rightarrow D , \quad (4.13)$$

$$C = \text{Int} .$$

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 $B = (\text{Int} \Rightarrow D) \Rightarrow D$.

Having established that types match, we can now omit the type annotations and rewrite the code:

$$\begin{aligned} & (f \Rightarrow g \Rightarrow g(f))(x \Rightarrow y \Rightarrow y(x))(h \Rightarrow h(10)) \\ \text{substitute } f, g : &= (h \Rightarrow h(10))(x \Rightarrow y \Rightarrow y(x)) \\ \text{substitute } h : &= (x \Rightarrow y \Rightarrow y(x))(10) \\ \text{substitute } x : &= y \Rightarrow y(10) . \end{aligned}$$

The type of this expression is $(\text{Int} \Rightarrow D) \Rightarrow D$ with a type parameter D . Since the argument y is an arbitrary function, we cannot simplify $y(10)$ or $y \Rightarrow y(10)$ any further. We conclude that $y:\text{Int} \Rightarrow D \Rightarrow y(10)$ is the final simplified form of Eq. (4.10).

To test this, we first define the function $f \Rightarrow g \Rightarrow g(f)$ as in Example 4.4.1.5,

```
def q[A, B]: A => (A => B) => B = { f => g => g(f) }
```

We also define the function $h \Rightarrow h(10)$ with a general type $(\text{Int} \Rightarrow E) \Rightarrow E$,

```
def r[E]: (\text{Int} \Rightarrow 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 => (\text{Int} => D) => D, (\text{Int} => D) => D](q)(r)
s: [D]=> (\text{Int} => D) => D
```

To verify that the function s^D indeed equals $y^{\text{Int} \Rightarrow D} \Rightarrow y(10)$, we apply s^D to some functions of type $\text{Int} \Rightarrow D$, 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
```

4.4.2 Exercises

Exercise 4.4.2.1 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 4.4.2.2 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 4.4.2.3 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 4.4.2.4 Define a function `ence` similarly to `twice` except it should apply a given function n times, where n is an additional curried argument.

Exercise 4.4.2.5 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 4.4.2.6 Revise the function from Exercise 1.6.2.4, implementing it 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 4.4.2.7 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(5.00000000053722)
```

Exercise 4.4.2.8 Write a function `curry2` converting an uncurried function of type `(Int, Int) => Int` into an equivalent curried function of type `Int => Int => Int`.

Exercise 4.4.2.9 Apply the function $(x \Rightarrow _ \Rightarrow x)$ to the value $(z \Rightarrow z(q))$ where q is a given value of type Q . Infer types in these expressions.

Exercise 4.4.2.10 Infer types in the following expressions and test in Scala:

- (a) $p \Rightarrow q \Rightarrow p(t \Rightarrow t(q))$.
- (b) $p \Rightarrow q \Rightarrow q(x \Rightarrow x(p(q)))$.

Exercise 4.4.2.11 Show that the following expressions cannot be well-typed:

- (a) $p \Rightarrow p(q \Rightarrow q(p))$.
- (b) $p \Rightarrow q \Rightarrow q(x \Rightarrow p(q(x)))$.

Exercise 4.4.2.12 Infer types and simplify the following code expressions by symbolic calculations:

- (a) $q \Rightarrow (x \Rightarrow y \Rightarrow z \Rightarrow x(z)(y(z))) (a \Rightarrow a) (b \Rightarrow b(q))$.
- (b) $(f \Rightarrow g \Rightarrow h \Rightarrow f(g(h))) (x \Rightarrow x)$.
- (c) $(x \Rightarrow y \Rightarrow x(y)) (x \Rightarrow y \Rightarrow x)$.
- (d) $(x \Rightarrow y \Rightarrow x(y)) (x \Rightarrow y \Rightarrow y)$.
- (e) $x \Rightarrow (f \Rightarrow y \Rightarrow f(y)(x)) (z \Rightarrow _ \Rightarrow z)$.
- (f) $z \Rightarrow (x \Rightarrow y \Rightarrow x) (x \Rightarrow x(z)) (y \Rightarrow y(z))$.

4.5 Discussion

4.5.1 Higher-order functions

The **order** of a function is the number of function arrows (\Rightarrow) 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 as result values.

The methods `andThen`, `compose`, `curried`, and `uncurried` are examples of higher-order functions that take other functions as arguments *and* return a new function.

The following example illustrates the concept of a function's order. Consider

```
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.

Although `f2` is a higher-order function, its "higher-orderness" comes from the fact that the 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
```

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 an argument of a function type.

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 curried nameless function $x \Rightarrow y \Rightarrow x + y$, the bound variables are the curried arguments x and y . The variable y is only defined within the scope ($y \Rightarrow x + y$) of the inner function; the variable x is defined within the entire scope of $x \Rightarrow y \Rightarrow x + y$.

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
} // 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

$$f(x) = \int_0^x \frac{dx}{1+x} ,$$

two bound variables named x are defined in two scopes: one in the scope of f , another in the scope of the nameless function $x \mapsto \frac{1}{1+x}$. 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 closest inner scope. The outer definition of x is then **shadowed**, i.e. hidden, by the definition of the inner x . For this reason, mathematicians expect that evaluating $f(10)$ will give

$$f(10) = \int_0^{10} \frac{dx}{1+x} = \log_e(11) \approx 2.398 ,$$

rather than $\int_0^{10} \frac{dx}{1+10} = \frac{10}{11}$, because the outer definition $x = 10$ is shadowed within the expression $\frac{1}{1+x}$ by the definition of x in the local scope of $x \mapsto \frac{1}{1+x}$.

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) is invisible outside that scope but 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

$$x \Rightarrow x \Rightarrow x ,$$

and let us decipher this confusing syntax. The symbol \Rightarrow groups to the right, so $x \Rightarrow x \Rightarrow x$ is the same as $x \Rightarrow (x \Rightarrow x)$. It is a function that takes x and returns $x \Rightarrow x$. Since the nameless function $(x \Rightarrow x)$ may be renamed to $(y \Rightarrow y)$ without changing its value, we can rewrite the code to

$$x \Rightarrow (y \Rightarrow y) .$$

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 $y \Rightarrow y$). So we can rewrite $(x \Rightarrow x \Rightarrow x)$ as $(_ \Rightarrow y \Rightarrow y)$, 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.²

The conventions for nameless functions in the operator syntax become:

²The 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.

- Function expressions group to the right, so $x \Rightarrow y \Rightarrow z \Rightarrow e$ means $x \Rightarrow (y \Rightarrow (z \Rightarrow e))$.
- Function applications group to the left, so $f x y z$ means $((f x) y) z$.
- Function applications group stronger than infix operations, so $x + f y$ means $x + (f y)$, just as in mathematics “ $x + \cos y$ ” groups “ $\cos y$ ” stronger than the infix “ $+$ ” operation.

Thus, $x \Rightarrow y \Rightarrow a b c + p q$ means $x \Rightarrow (y \Rightarrow ((a b) c) + (p q))$. When this notation becomes hard to read correctly, one needs to add parentheses, e.g. to write $f(x \Rightarrow g h)$ instead of $f x \Rightarrow g h$.

This book will avoid using the “operator syntax” when reasoning about code. Scala does not support the parentheses-free operator syntax; parentheses are needed around each curried argument.

From the point of view of programming language theory, curried functions are “simpler” because they always have a *single* argument (and 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 used e.g. in OCaml and Haskell, 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 first define short names for complicated expressions and then use these 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 for curried functions in Scala is the ability to specify a curried argument using the curly brace syntax. Compare the two definitions of the function `summation` described in Section 1.7.5:

```
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
```

To apply these functions to arguments, we need to use slightly different syntax:

```
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
```

bodies to contain their own local definitions (`val` or `def`).

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

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 some other languages.

4.5.4 Deriving a function’s code from its type signature

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,³ with a Scala implementation available.⁴

³https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system

⁴<http://dysphoria.net/2009/06/28/hindley-milner-type-inference-in-scala/>

It is remarkable that one can sometimes derive the 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 $f \circ g$ 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=>B`, and `g:B=>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 derived 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)) }
```

In Chapter 5 and in Appendix C, we will see how a function's code can be derived from type signatures for a wide range of fully parametric functions.

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 does not depend on values of any specific data types such as `Int` or `String`. 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 also seen in Section 4.5.4 that for certain functions of this kind, the code can be derived unambiguously from the type signature.

There exists a mathematical theory (called the **Curry-Howard correspondence**) that provides precise conditions for the possibility of deriving a function's code from its type and a systematic derivation algorithm. Technical details about the algorithm are found in Appendix C. This chapter describes the main results and applications of this theory to functional programming.

5.1 Values computed by fully parametric functions

5.1.1 Motivation

Consider the Scala code of a fully parametric function,

```
def f[A, B, ...]: ... = {
  val x: Either[A, B] = ... // Some expression here.
  ...
}
```

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!
  ...
}
```

function that returns values of type `A`. In `fmap`, no values of type `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.
    ...
}
```

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 in the following code:

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]`.

The reason is that a fully parametric function cannot compute values of type `A` from scratch without using previously given values of type `A` and without applying a function

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

```

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.
}

```

$\Rightarrow A$, 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 its return value of type A . Since a fully parametric function cannot create values of an arbitrary type A from scratch, we see no possibility of computing h within the body of `before`.

Can we prove rigorously that a value of type $C \Rightarrow A$ 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 $C \Rightarrow A$* " into mathematical formulas, with rigorous rules for proving them true or false.

In Section 3.5.3, we denoted by $\mathcal{CH}(A)$ the proposition "the Code \mathcal{H} has a value of type A ". By "the code" we now mean the body of a given fully parametric function. So, the notation $\mathcal{CH}(A)$ is not completely adequate because the validity of the proposition $\mathcal{CH}(A)$ depends not only on the choice of the type A but also on the place in the code fragment where the value of type A needs to be computed. What exactly is this additional dependency? In the above examples, we used the *types* of a function's arguments when reasoning about getting a value of a given type A . Thus, a precise description of the proposition $\mathcal{CH}(A)$ is

\mathcal{CH} -proposition : a fully parametric function having arguments of types X, Y, \dots, Z can compute a value of type A (5.1)

Here X, Y, \dots, Z, A may be either type parameters or more complicated type expressions such as $B \Rightarrow C$ or $(C \Rightarrow D) \Rightarrow E$, built from other type parameters.

If arguments of types X, Y, \dots, Z are given, it means we already have values of these types. So, the propositions $\mathcal{CH}(X), \mathcal{CH}(Y), \dots, \mathcal{CH}(Z)$ will be true. Thus, proposition (5.1) is equivalent to " $\mathcal{CH}(A)$ assuming $\mathcal{CH}(X), \mathcal{CH}(Y), \dots, \mathcal{CH}(Z)$ ". In mathematical logic, a statement of this form is called a **sequent** and is denoted by

$\mathcal{CH}(X), \mathcal{CH}(Y), \dots, \mathcal{CH}(Z) \vdash \mathcal{CH}(A)$ (5.2)

The assumptions $\mathcal{CH}(X), \mathcal{CH}(Y), \dots, \mathcal{CH}(Z)$ are called **premises** and the proposition $\mathcal{CH}(A)$ is called the **goal**. Showing rigorously the possibility of computing values in functions means proving sequents of the form (5.2). Our previous examples are denoted by the following sequents:

fmap for Option : $\mathcal{CH}(A \Rightarrow B) \vdash \mathcal{CH}(\text{Option}[A] \Rightarrow \text{Option}[B])$
the function before : $\mathcal{CH}(A \Rightarrow B), \mathcal{CH}(B \Rightarrow C) \vdash \mathcal{CH}(A \Rightarrow C)$
value of type A within fmap : $\mathcal{CH}(A \Rightarrow B), \mathcal{CH}(\text{Option}[A]) \vdash \mathcal{CH}(A)$
value of type $C \Rightarrow A$ within before : $\mathcal{CH}(A \Rightarrow B), \mathcal{CH}(B \Rightarrow C) \vdash \mathcal{CH}(C \Rightarrow A)$

Calculations in formal logic are called **proofs**. So, 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 such sequents rigorously.

A proposition $\mathcal{CH}(A)$ may be true for one set of premises such as $\mathcal{CH}(X), \mathcal{CH}(Y), \dots, \mathcal{CH}(Z)$ but false for 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 use the shorter notation $\mathcal{CH}(A)$.

5.1.2 Type notation and \mathcal{CH} -propositions for standard type constructions

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 pro-

The body of `before` may only use the arguments f and g . We can compute a value of type $A \Rightarrow C$ by composing f and g , but it is impossible to compute a value h of type C

grams could use. A special **type notation** explained in this section will help us write type expressions more concisely. (See Appendix A for reference on the type notation.)

There are six **standard type constructions** supported by all functional languages: primitive types (including `Unit` type and the void type, called `Nothing` in Scala), product (tuple) types, co-product (disjunctive) 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 Unit type The `Unit` type has only a single value `()`, and this value (an “empty tuple”) 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 $\mathcal{CH}(\text{Unit})$ is always true. In the type notation, the `Unit` type is denoted by `1`.

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

```
val x: N1 = N1()
```

So, the proposition $\mathcal{CH}(N1)$ is always true. Named unit types are denoted by `1`, just as the `Unit` type itself.

1b) Rule for the void type The Scala type `Nothing` has no values, so the proposition $\mathcal{CH}(\text{Nothing})$ is always false. The type `Nothing` is denoted by `0` in the type notation.

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 any 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 the rule for the `Unit` type.

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 $\mathcal{CH}((A, B)) = \mathcal{CH}(A) \wedge \mathcal{CH}(B)$. 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 $A \times B \times \dots \times C$. For example, the Scala definition

```
case class Person(firstName: String, lastName: String, age: Int)
```

is written in the type notation as $\text{String} \times \text{String} \times \text{Int}$. So, the rule for tuple types is

$$\mathcal{CH}(A \times B \times \dots \times C) = \mathcal{CH}(A) \wedge \mathcal{CH}(B) \wedge \dots \wedge \mathcal{CH}(C) .$$

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 $\mathcal{CH}(\text{Either}[A, B]) = \mathcal{CH}(A) \vee \mathcal{CH}(B)$. In the type notation, the Scala type `Either[A, B]` is written as $A + B$. A longer 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

$$\text{RootsOfQ} = 1 + \text{Double} + \text{Double} \times \text{Double} .$$

The type notation is significantly shorter because it omits all case class names and part names from the type definitions. In this notation, the rule for disjunctive types is

$$CH(A + B + \dots + C) = CH(A) \vee CH(B) \vee \dots \vee CH(C) .$$

4) Rule for function types Consider now a function type such as $A \Rightarrow B$. (This type is written in the type notation as $A \Rightarrow B$.) To compute a value of that type, we need to write code such as

```
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, $CH(A \Rightarrow B)$ 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, $CH(A) \Rightarrow CH(B)$, which means that $CH(B)$ can be proved if $CH(A)$ already holds. So the rule for function types is

$$CH(A \Rightarrow B) = CH(A) \Rightarrow CH(B) .$$

5) Rule for parameterized types Consider a function with type parameters, e.g.

```
def f[A, B]: A => (A => B) => B = { x => g => g(x) }
```

Being able to define the body of such a function is equivalent to being able to compute a value of type $A \Rightarrow (A \Rightarrow B) \Rightarrow B$ for *all* possible types A and B . In the notation of formal logic, this is written as

$$CH(\forall(A, B). A \Rightarrow (A \Rightarrow B) \Rightarrow B)$$

and is equivalent to

$$\forall(A, B). CH(A \Rightarrow (A \Rightarrow B) \Rightarrow B) .$$

The code notation for the parameterized function f is

$$f^{A,B} : A \Rightarrow (A \Rightarrow B) \Rightarrow B ,$$

and its type can be written as

$$\forall(A, B). A \Rightarrow (A \Rightarrow B) \Rightarrow B .$$

The symbol \forall means “for all” and is known as the **universal quantifier** in logic.

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 as

$$\begin{aligned} F^{A,B} &\triangleq A \Rightarrow (A \Rightarrow B) \Rightarrow B , \\ f^{A,B} : F^{A,B} &\triangleq x:A \Rightarrow g:A \Rightarrow B \Rightarrow g(x) , \end{aligned}$$

or equivalently (although somewhat less readability)

$$f : (\forall(A, B). F^{A,B}) \triangleq \forall(A, B). x:A \Rightarrow g:A \Rightarrow B \Rightarrow g(x) .$$

In Scala 3, the function f can be written as a value via the syntax

```
val f: [A, B] => A => (A => B) => B = {   // Valid only in Scala 3.
  [A, B] => (x: A) => (g: A => B) => g(x)
}
```

Type construction	Scala syntax	Type notation	\mathcal{CH} -proposition
type parameter	<code>[A]</code>	A	$\mathcal{CH}(A)$
product type (tuple)	<code>(A, B)</code>	$A \times B$	$\mathcal{CH}(A) \wedge \mathcal{CH}(B)$
disjunctive type	<code>Either[A, B]</code>	$A + B$	$\mathcal{CH}(A) \vee \mathcal{CH}(B)$
function type	<code>A => B</code>	$A \Rightarrow B$	$\mathcal{CH}(A) \Rightarrow \mathcal{CH}(B)$
unit or a “named unit” type	<code>Unit</code>	$\mathbb{1}$	$\mathcal{CH}(\mathbb{1}) = \text{True}$
primitive type	<code>Int, String, ...</code>	<code>Int, String, ...</code>	$\mathcal{CH}(\text{Int}) = \text{True}$
void type	<code>Nothing</code>	\emptyset	$\mathcal{CH}(\emptyset) = \text{False}$
value parameterized by type	<code>def f[A]: F[A]</code>	$f^A : F^A$	$\forall A. \mathcal{CH}(F^A)$
type with quantifier	<code>[A] => F[A]</code> (Scala 3)	$\forall A. F^A$	$\forall A. \mathcal{CH}(F^A)$

Table 5.1: The correspondence between type constructions and \mathcal{CH} -propositions.

This syntax corresponds more closely to the mathematical notation shown above.

So, the rule for parameterized types with the type notation F^A is

$$\mathcal{CH}(\forall A. F^A) = \forall A. \mathcal{CH}(F^A) .$$

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. So, the type notation allows us to use 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 $\mathcal{CH}(\dots)$.

The precedence of operators in the type notation is chosen to have fewer parentheses in the type expressions that are frequently used. Here are the rules of precedence:

- The type product operators (\times) group stronger than the disjunctive operators ($+$), 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 arrows (\Rightarrow) group weaker than the operators $+$ and \times , so that often-used types such as $A \Rightarrow \mathbb{1} + B$ (representing `A => Option[B]`) or $A \times B \Rightarrow C$ (representing `((A, B)) => C`) can be written without any parentheses. Type expressions such as $(A \Rightarrow B) \times C$ will require parentheses but are used less often.
- The type quantifiers group weaker than all other operators, so we can write types such as $\forall A. A \Rightarrow A \Rightarrow A$ without parentheses. 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 $(\forall A. A \Rightarrow A) \Rightarrow \mathbb{1}$.

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, in particular, makes the structure of disjunctive types and their parts more transparent, compared with the Scala syntax. Names of types and parts of types are, of course, helpful for the programmer because they show the significance of data in the application. 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 expressions can be straightforwardly translated from the type notation into Scala code.

Let us get some experience converting between type notation and Scala code.

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 the type parameter 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 $A \Rightarrow (A, A)$. We also note that there is only one way of implementing a fully parametric function with type signature $A \Rightarrow (A, A)$: the function must duplicate its given argument.

It is convenient to use the letter Δ for the function `delta`. In the type notation, the type signature of Δ is written as

$$\Delta^A : A \Rightarrow A \times A .$$

So the proposition $\mathcal{CH}(\Delta)$ (meaning “the function Δ can be implemented”) is

$$\mathcal{CH}(\Delta) = \forall A. \mathcal{CH}(A \Rightarrow A \times A) .$$

In the type expression $A \Rightarrow A \times A$, the product symbol (\times) binds stronger than the function arrow (\Rightarrow), so the parentheses in $A \Rightarrow (A \times A)$ may be omitted.

Using the rules for transforming \mathcal{CH} -propositions, we rewrite

$$\begin{aligned} & \mathcal{CH}(A \Rightarrow A \times A) \\ \text{rule for function types : } &= \mathcal{CH}(A) \Rightarrow \mathcal{CH}(A \times A) \\ \text{rule for tuple types : } &= \mathcal{CH}(A) \Rightarrow (\mathcal{CH}(A) \wedge \mathcal{CH}(A)) . \end{aligned}$$

Thus the proposition $\mathcal{CH}(\Delta)$ is equivalent to

$$\mathcal{CH}(\Delta) = \forall A. \mathcal{CH}(A) \Rightarrow (\mathcal{CH}(A) \wedge \mathcal{CH}(A)) .$$

Example 5.1.3.2 The standard disjunctive types `Either[A, B]` and `Option[A]` are written in the type notation as

$$\text{Either}^{A,B} \triangleq A + B , \quad \text{Opt}^A \triangleq \mathbb{1} + A .$$

The type `Either[A, B]` is written as $A + B$ by definition of the disjunctive type operator (+). The type `Option[A]` has two disjoint cases, `None` and `Some[A]`. The case class `None` is a “named `Unit`” and is denoted by `1`. The case class `Some[A]` contains a single value of type `A`. So, the type notation for `Option[A]` is $\mathbb{1} + A$.

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

$$\text{UserAction} \triangleq \text{String} \times \text{String} + \text{String} + \text{Long} . \quad (5.3)$$

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 $(\text{String}, \text{String})$, (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

$$\text{Either3}^{A,B,C} \triangleq A + B + C .$$

Example 5.1.3.5 Define a Scala type constructor `F[A]` corresponding to the type notation

$$F^A \triangleq \mathbb{1} + \text{Int} \times A \times A + \text{Int} \times (\text{Int} \Rightarrow A) .$$

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 $\text{Int} \times (\text{Int} \Rightarrow A)$. To define these case classes, we need to name their parts. The final code is

```
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 function

```
def fmap[A, B](f: A => B): Option[A] => Option[B]
```

in the type notation.

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

$$\begin{aligned} \text{fmap}^{A,B} : (A \Rightarrow B) \Rightarrow \mathbb{1} + A \Rightarrow \mathbb{1} + B , \\ \text{or equivalently : } \text{fmap} : \forall(A, B). (A \Rightarrow B) \Rightarrow \mathbb{1} + A \Rightarrow \mathbb{1} + B . \end{aligned}$$

We do not put parentheses around $\mathbb{1} + A$ and $\mathbb{1} + B$ because the function arrows (\Rightarrow) group weaker than the other type operations. Parentheses around $(A \Rightarrow B)$ are required.

We will usually prefer to write type parameters in superscripts rather than under type quantifiers. So, we will prefer to write $\text{id}^A = x:A \Rightarrow x$ rather than $\text{id} : (\forall A. A \Rightarrow A) = x:A \Rightarrow x$.

5.1.4 Exercises: Type notation

Exercise 5.1.4.1 Define a Scala disjunctive type $Q[T, A]$ corresponding to the type notation

$$Q^{T,A} \triangleq \mathbb{1} + T \times A + \text{Int} \times (T \Rightarrow T) + \text{String} \times A \quad .$$

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 `OptE[A, B]` written in the type notation as $\text{OptE}^{A,B} \triangleq \mathbb{1} + A + B$.

Exercise 5.1.4.4 Write a Scala type signature for the fully parametric function

$$\text{flatMap}^{A,B} : \mathbb{1} + A \Rightarrow (A \Rightarrow \mathbb{1} + B) \Rightarrow \mathbb{1} + B$$

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 of the form $\mathcal{CH}(A)$ 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. A simple example of a true formula is “any proposition α is equivalent to itself”,

$$\forall \alpha. \alpha = \alpha \quad .$$

In logic, equivalence of propositions is usually understood as **implication** (\Rightarrow) in both directions: $\alpha = \beta$ means $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$. So, the above formula is the same as

$$\forall \alpha. \alpha \Rightarrow \alpha \quad .$$

If the proposition α is a \mathcal{CH} -proposition, $\alpha \triangleq \mathcal{CH}(A)$ for some type A , we obtain the formula

$$\forall A. \mathcal{CH}(A) \Rightarrow \mathcal{CH}(A) \quad . \tag{5.4}$$

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. (5.4) as

$$\begin{aligned} & \forall A. \underline{\mathcal{CH}(A)} \Rightarrow \mathcal{CH}(A) \\ \text{rule for function types : } &= \underline{\forall A. \mathcal{CH}(A \Rightarrow A)} \\ \text{rule for parameterized types : } &= \mathcal{CH}(\forall A. A \Rightarrow A) \quad . \end{aligned}$$

The last line shows the \mathcal{CH} -proposition that corresponds to the function type $\forall A. A \Rightarrow A$. Translating the type notation into a Scala type signature, we get

```
def f[A]: A => A
```

This type signature can be easily implemented,

```
def f[A]: A => A = { x => x }
```

So, in this example we see how we converted a true formula in logic into the type of a value `f` that can be implemented.

While the formula $\forall \alpha. \alpha = \alpha$ may be self-evident, the point of using formal logic is to have a set of axioms and proof rules that allow us to deduce *all* correct formulas systematically, without need for intuition or guessing. What axioms and proof rules are suitable for proving \mathcal{CH} -propositions?

A well-known set of logical rules is called Boolean logic. In that logic, each proposition is either *True* or *False*, and the implication operation (\Rightarrow) is defined by

$$(\alpha \Rightarrow \beta) \triangleq ((\neg\alpha) \vee \beta) \quad . \quad (5.5)$$

To verify a formula, substitute *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 basic operations (disjunction, conjunction, negation, and implication) have the following truth tables:

α	β	$\alpha \vee \beta$	$\alpha \wedge \beta$	$\neg\alpha$	$\alpha \Rightarrow \beta$
<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>True</i>
<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>True</i>

So the formula $\alpha \Rightarrow \alpha$ has the value *True* 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 formula $\forall(\alpha, \beta). (\alpha \wedge \beta) \Rightarrow \alpha$; that formula is true in Boolean logic since all values in the last column are *True*:

α	β	$\alpha \wedge \beta$	$(\alpha \wedge \beta) \Rightarrow \alpha$
<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>False</i>	<i>True</i>
<i>False</i>	<i>False</i>	<i>False</i>	<i>True</i>

The formula $\forall(\alpha, \beta). \alpha \Rightarrow (\alpha \wedge \beta)$ is not true in Boolean logic, which we can see from the following truth table (one value in the last column is *False*):

α	β	$\alpha \wedge \beta$	$\alpha \Rightarrow (\alpha \wedge \beta)$
<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>False</i>	<i>True</i>	<i>False</i>	<i>True</i>
<i>False</i>	<i>False</i>	<i>False</i>	<i>True</i>

Table 5.2 shows more examples of logical formulas that are true in Boolean logic. Each formula is first given in terms of \mathcal{CH} -propositions (we denoted $\alpha \triangleq \mathcal{CH}(A)$ and $\beta \triangleq \mathcal{CH}(B)$ for brevity) and then into a Scala type signature of a function that can be implemented.

Table 5.3 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.

At first sight, it appears 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 suitable for reasoning about types in a functional language. Below we will see some examples of formulas that are true in Boolean logic but yield unimplementable type signatures.

Logic formula	Type formula	Scala code
$\forall\alpha. \alpha \Rightarrow \alpha$	$\forall A. A \Rightarrow A$	<code>def id[A](x: A): A = x</code>
$\forall\alpha. \alpha \Rightarrow True$	$\forall A. A \Rightarrow \mathbb{1}$	<code>def toUnit[A](x: A): Unit = ()</code>
$\forall(\alpha, \beta). \alpha \Rightarrow (\alpha \vee \beta)$	$\forall(A, B). A \Rightarrow A + B$	<code>def toL[A, B](x: A): Either[A, B] = Left(x)</code>
$\forall(\alpha, \beta). (\alpha \wedge \beta) \Rightarrow \alpha$	$\forall(A, B). A \times B \Rightarrow A$	<code>def first[A, B](p: (A, B)): A = p._1</code>
$\forall(\alpha, \beta). \alpha \Rightarrow (\beta \Rightarrow \alpha)$	$\forall(A, B). A \Rightarrow (B \Rightarrow A)$	<code>def const[A, B](x: A): B => A = (_ => x)</code>

Table 5.2: Examples of logical formulas that are true theorems in Boolean logic.

Logic formula	Type formula	Scala type signature
$\forall\alpha. True \Rightarrow \alpha$	$\forall A. \mathbb{1} \Rightarrow A$	<code>def f[A](x: Unit): A</code>
$\forall(\alpha, \beta). (\alpha \vee \beta) \Rightarrow \alpha$	$\forall(A, B). A + B \Rightarrow A$	<code>def f[A, B](x: Either[A, B]): A</code>
$\forall(\alpha, \beta). \alpha \Rightarrow (\alpha \wedge \beta)$	$\forall(A, B). A \Rightarrow A \times B$	<code>def f[A, B](p: A): (A, B)</code>
$\forall(\alpha, \beta). (\alpha \Rightarrow \beta) \Rightarrow \alpha$	$\forall(A, B). (A \Rightarrow B) \Rightarrow A$	<code>def f[A, B](x: A => B): A</code>

Table 5.3: Examples of logical formulas that are *not* true in Boolean logic.

5.2.2 Example: Failure of Boolean logic for type reasoning

To see an explicit example of obtaining an incorrect result when using Boolean logic to reason about values computed by fully parametric functions, consider the following type,

$$\forall(A, B, C). (A \Rightarrow B + C) \Rightarrow (A \Rightarrow B) + (A \Rightarrow C) , \quad (5.6)$$

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 as a fully parametric function. To see why, consider that the only available data is a function $g: A \Rightarrow B + C$, 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 $A \Rightarrow B$ or a function of type $A \Rightarrow C$. How can the code of `bad` make this 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 $A \Rightarrow B$ or $A \Rightarrow C$ must be independent of the function g ; that decision must be hard-coded in the function `bad`.

Suppose we hard-coded the decision to return a function of type $A \Rightarrow B$. How can we create a function of type $A \Rightarrow B$ 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 have type `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 $A \Rightarrow B$ out of the function g . Similarly, we cannot build a function of type $A \Rightarrow C$ out of g .

Whether we decide to return $A \Rightarrow B$ or $A \Rightarrow C$, 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 $A \Rightarrow B$ and $A \Rightarrow C$ depending on a given value of type A . This idea, however, means that we are working with a different type signature:

$$\forall(A, B, C). (A \Rightarrow B + C) \Rightarrow A \Rightarrow (A \Rightarrow B) + (A \Rightarrow C) .$$

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 (5.6).

Now let us convert the type signature (5.6) into a \mathcal{CH} -proposition:

$$\begin{aligned} \forall(\alpha, \beta, \gamma). (\alpha \Rightarrow (\beta \vee \gamma)) &\Rightarrow ((\alpha \Rightarrow \beta) \vee (\alpha \Rightarrow \gamma)) , \\ \text{where } \alpha &\triangleq \mathcal{CH}(A), \quad \beta \triangleq \mathcal{CH}(B), \quad \gamma \triangleq \mathcal{CH}(C) . \end{aligned} \quad (5.7)$$

It turns out that this formula is true *in Boolean logic*. To prove this, we need to show that Eq. (5.7) is equal to *True* for any Boolean values of the variables α, β, γ . One way is to rewrite the expression (5.7) using the rules of Boolean logic, such as Eq. (5.5):

$$\begin{aligned} \alpha \Rightarrow (\beta \vee \gamma) &= (\neg\alpha) \vee (\beta \vee \gamma) , \\ \text{definition of } \Rightarrow \text{ via Eq. (5.5)} : &= (\alpha \Rightarrow \beta) \vee (\alpha \Rightarrow \gamma) \\ &= (\neg\alpha) \vee (\beta \vee (\neg\alpha)) \vee \gamma \\ \text{definition of } \Rightarrow \text{ via Eq. (5.5)} : &= (\neg\alpha) \vee (\beta \vee \gamma) , \\ \text{property } x \vee x = x \text{ in Boolean logic} : &= (\neg\alpha) \vee \beta \vee \gamma , \end{aligned}$$

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 $X = \text{True}$ and $Y = \text{False}$. So, Eq. (5.7) can be *False* only if $(\alpha \Rightarrow (\beta \vee \gamma)) = \text{True}$ and $(\alpha \Rightarrow \beta) \vee (\alpha \Rightarrow \gamma) = \text{False}$. A disjunction can be false only when both parts are false; so we must have both $(\alpha \Rightarrow \beta) = \text{False}$ and $(\alpha \Rightarrow \gamma) = \text{False}$. This is only possible if $\alpha = \text{True}$ and $\beta = \gamma = \text{False}$. But, with these value assignments, we find $(\alpha \Rightarrow (\beta \vee \gamma)) = \text{False}$ rather than *True* as we assumed. It follows that we cannot ever make Eq. (5.7) equal to *False*. This proves Eq. (5.7) to be true in Boolean logic.

5.2.3 The rules of proof for \mathcal{CH} -propositions

Section 5.2.2 shows that some true formulas in Boolean logic do not correspond to types of *implementable* fully parametric functions. However, we have also seen several other examples where Boolean logic does provide correct results: some true formulas correspond to implementable type signatures, while some false formulas correspond to non-implementable type signatures.

Instead of guessing whether the rules of Boolean logic are suitable, let us derive the suitable logical axioms and proof rules systematically.

The proposition $\mathcal{CH}(A)$ is true when a value of type A can be computed by a fully parametric function with a given type signature. To describe all possible ways of computing a value of type A , we need to enumerate all possible ways of writing code within a fully parametric function. The requirement of parametricity means that we are not allowed to use any specific types such as `Int` or `String`. We are only allowed to work with values of unknown types described by the given type parameters. We cannot use any concrete values such as `123` or `"hello"`, or any library functions that work with specific (non-parametric) types; however, we are permitted to use fully parametric types, such as `Either[A, B]` or `Option[A]`. The allowed eight code constructs are illustrated in this code fragment:

```

def f[A, B, ...](a: A, b: B) = { // (A given type signature.)
  val x1: Unit = ()           // 1) Create a value of type Unit.
  val x2: A = a               // 2) Use a given argument.
  val x3 = { x: A => ... }    // 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.
}

```

A value of type X can be computed (i.e. $\mathcal{CH}(X)$ is true) if and only if we can create a sequence of computed values such as x_1, x_2, \dots , each being the result of one of these eight code constructs, ending with a value of type X . So, each of the eight code constructs should correspond to a logical rule for proving a \mathcal{CH} -proposition.

A set of axioms and proof rules defines a **formal logic**. So, we will now write the proof rules that will define *the logic appropriate for reasoning about \mathcal{CH} -propositions*.

Because each proof rule will be obtained from a specific code construct, any \mathcal{CH} -proposition such as $\mathcal{CH}(X)$ proved by applying a sequence of these rules will automatically correspond to a code fragment that combines the relevant code constructs to compute a value of type X . Conversely, any code computing a value of type X must be a combination of some of the eight code constructs, and that combination can be automatically translated into a sequence of applications of proof rules in the logic to produce a proof of the proposition $\mathcal{CH}(X)$.

Let us now write down the proof rules that follow from the eight code constructs. We will need to consider the full formulation (5.1) of \mathcal{CH} -propositions and write them as sequents such as Eq. (5.2). For brevity, we define $\alpha \triangleq \mathcal{CH}(A)$, $\beta \triangleq \mathcal{CH}(B)$, etc. It is also customary to use the letter Γ to denote a set of premises, such as $\mathcal{CH}(X), \mathcal{CH}(Y), \dots, \mathcal{CH}(Z)$ in Eq. (5.2). So, we can write a shorter formula $\Gamma \vdash \alpha$ instead of the sequent (5.2).

With these notations, we will now enumerate all the possible ways of proving that a \mathcal{CH} -proposition is true. We assume that the set of premises Γ is known.

1) Create a 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:

$$\text{Proof}(\Gamma \vdash \mathcal{CH}(\mathbb{1})) = 1 ,$$

where we denoted by 1 the value $()$.

In formal logic, a sequent that is found to be always true, such as our $\Gamma \vdash \mathcal{CH}(\mathbb{1})$, is called an **axiom** and is written in the following notation,

$$\frac{}{\Gamma \vdash \mathcal{CH}(\mathbb{1})} \quad (\text{create unit}) .$$

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 $\alpha \triangleq \mathcal{CH}(A)$ 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 $x:A$),

$$\text{Proof}(\Gamma, \alpha \vdash \alpha) = x:A .$$

This sequent is an axiom since its proof requires no previous sequents. The formal logic notation for this axiom is

$$\frac{}{\Gamma, \alpha \vdash \alpha} \text{ (use arg)} .$$

3) Create a function At any place in the code, we may compute a nameless function of type, say, $A \Rightarrow B$, 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

$$\Gamma \vdash \mathcal{CH}(A \Rightarrow B) = \Gamma \vdash \mathcal{CH}(A) \Rightarrow \mathcal{CH}(B) \triangleq \Gamma \vdash \alpha \Rightarrow \beta$$

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

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \Rightarrow \beta} \text{ (create function)} .$$

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; the 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 $A \Rightarrow B$ and requires us to already have a suitable `exprB`. This exactly corresponds to the proof rule “create function”. We may write the corresponding code as

$$\text{Proof}(\Gamma \vdash \mathcal{CH}(A) \Rightarrow \mathcal{CH}(B)) = x^{:A} \Rightarrow \text{Proof}(\Gamma, x^{:A} \vdash \mathcal{CH}(B)) .$$

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 5.2.4 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 $A \Rightarrow B$ to an already computed value of type A . The result will be a value of type B . This corresponds to assuming $\mathcal{CH}(A \Rightarrow B)$ and $\mathcal{CH}(A)$, and then deriving $\mathcal{CH}(B)$. The formal logic notation for this proof rule is

$$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \alpha \Rightarrow \beta}{\Gamma \vdash \beta} \text{ (use function)} .$$

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,

$$\text{Proof}(\Gamma \vdash \beta) = \text{Proof}(\Gamma \vdash \alpha \Rightarrow \beta)(\text{Proof}(\Gamma \vdash \alpha)) .$$

5) Create a tuple If we have already computed some values `x:A` and `y:B`, we may write the expression `(x, y)` and so compute a value of the tuple type `(A, B)`. The proof rule is

$$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta}{\Gamma \vdash \alpha \wedge \beta} \text{ (create tuple)} .$$

We can write the corresponding code expression as

$$\text{Proof}(\Gamma \vdash \alpha \wedge \beta) = \text{Proof}(\Gamma \vdash \alpha) \times \text{Proof}(\Gamma \vdash \beta) ,$$

where we write $a \times b$ to represent a pair of two values.

This rule describes creating a pair of 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 is sufficient to have a sequent rule for creating pairs; this rule can express the sequent rules for creating all other tuples, and 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

$$\frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \alpha} \quad (\text{use tuple-1}) \qquad \frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \beta} \quad (\text{use tuple-2}) .$$

The code can be written as

$$\text{Proof}(\Gamma \vdash \alpha) = \nabla_1(\text{Proof}(\Gamma \vdash \alpha \wedge \beta)) ,$$

$$\text{Proof}(\Gamma \vdash \beta) = \nabla_2(\text{Proof}(\Gamma \vdash \alpha \wedge \beta)) ,$$

where we introduced the notation ∇_1 and ∇_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 is sufficient 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

$$\frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \vee \beta} \quad (\text{create Left}) \qquad \frac{\Gamma \vdash \beta}{\Gamma \vdash \alpha \vee \beta} \quad (\text{create Right}) .$$

The corresponding code can be written as

$$\text{Proof}(\Gamma \vdash \alpha \vee \beta) = \text{Left}(\text{Proof}(\Gamma \vdash \alpha)) ,$$

$$\text{Proof}(\Gamma \vdash \alpha \vee \beta) = \text{Right}(\text{Proof}(\Gamma \vdash \beta)) .$$

8) Use a disjunctive value The only way we may use a value of disjunctive type `Either[A, B]` is by pattern matching on it:

```
val result: C = (e: Either[A, B]) match {
  case Left(x:A)    => expr1(x)
  case Right(y:B)   => expr2(y)
}
```

Here, `expr1(x)` must be an expression of some type c , computed using $x:A$ and any previously available arguments (i.e. the premises Γ). Similarly, `expr2(y)` must be an expression of type c computed using $y:B$ and previous arguments. It is clear that `expr1(x)` represents a

proof of a sequent with an additional premise of type A , i.e. $\Gamma, \alpha \vdash \gamma$, where we denoted $\gamma \triangleq CH(C)$. Similarly, `expr2(y)` is a proof of the sequent $\Gamma, \beta \vdash \gamma$. So, we can write the proof rule corresponding to the `match/case` expression as a rule with three previous sequents:

$$\frac{\Gamma \vdash \alpha \vee \beta \quad \Gamma, \alpha \vdash \gamma \quad \Gamma, \beta \vdash \gamma}{\Gamma \vdash \gamma} \quad (\text{use Either}) .$$

The code can be written as

$$\text{Proof}(\Gamma \vdash \gamma) = \text{Proof}(\Gamma \vdash \alpha \vee \beta) \text{ match } \begin{cases} \text{have } x:A : \text{ Proof}(\Gamma, x:A \vdash \gamma) \\ \text{have } y:B : \text{ Proof}(\Gamma, y:B \vdash \gamma) \end{cases} .$$

Table 5.4 summarizes the eight proof rules derived in this section. These proof rules define a logic known as the **intuitionistic propositional logic** or **constructive propositional logic**. We will call this logic “constructive” for short.

axioms :	$\frac{}{\Gamma \vdash \mathcal{CH}(1)}$ (create unit)	$\frac{}{\Gamma, \alpha \vdash \alpha}$ (use arg)
derivation rules :	$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \Rightarrow \beta}$ (create function)	
	$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \alpha \Rightarrow \beta}{\Gamma \vdash \beta}$ (use function)	
	$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta}{\Gamma \vdash \alpha \wedge \beta}$ (create tuple)	
	$\frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \alpha}$ (use tuple-1)	$\frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \beta}$ (use tuple-2)
	$\frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \vee \beta}$ (create Left)	$\frac{\Gamma \vdash \beta}{\Gamma \vdash \alpha \vee \beta}$ (create Right)
	$\frac{\Gamma \vdash \alpha \vee \beta \quad \Gamma, \alpha \vdash \gamma \quad \Gamma, \beta \vdash \gamma}{\Gamma \vdash \gamma}$ (use Either)	

Table 5.4: Proof rules for the constructive logic.

5.2.4 Example: Proving a \mathcal{CH} -proposition and deriving code

The task is to 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

$$F \triangleq \forall(A, B). ((A \Rightarrow A) \Rightarrow B) \Rightarrow B \quad .$$

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

$$\forall(\alpha, \beta). \emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta \quad , \quad (5.8)$$

where we denoted $\alpha \triangleq \mathcal{CH}(A)$ and $\beta \triangleq \mathcal{CH}(B)$.

The next step is to prove the sequent (5.8) using the logic proof rules of Section 5.2.3. 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. (5.8), 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 $\Gamma = \emptyset$, $p = (\alpha \Rightarrow \alpha) \Rightarrow \beta$, and $q = \beta$:

$$\frac{(\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta}{\emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta} \quad .$$

We now need to prove the sequent $(\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta$, which we can write as $\Gamma_1 \vdash \beta$ where we defined $\Gamma_1 \triangleq [(\alpha \Rightarrow \alpha) \Rightarrow \beta]$ to be the set containing the single premise $(\alpha \Rightarrow \alpha) \Rightarrow \beta$.

There are no proof rules that derive a sequent with an explicit premise of 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. We would be able to use that rule,

$$\frac{\Gamma_1 \vdash \alpha \Rightarrow \alpha \quad \Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta}{\Gamma_1 \vdash \beta} \quad ,$$

if we could prove the two 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 $\Gamma_1 \vdash \alpha \Rightarrow \alpha$ as follows,

$$\frac{\Gamma_1, \alpha \vdash \alpha}{\Gamma_1 \vdash \alpha \Rightarrow \alpha} \quad .$$

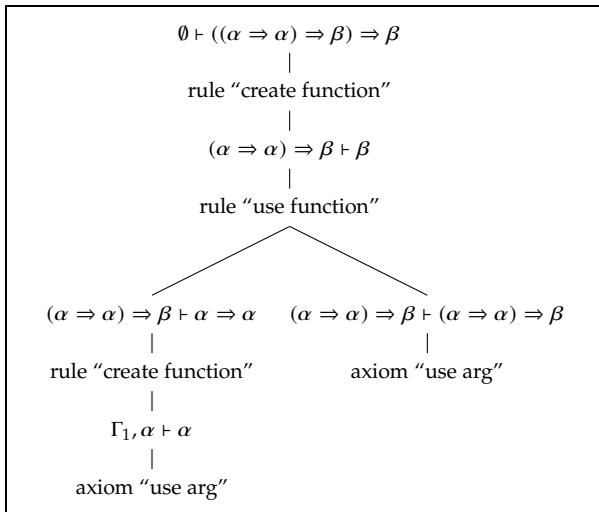


Figure 5.1: Proof tree for the sequent (5.8).

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 (5.8) is now complete and can be visualized as a tree (Figure 5.1). The next step is to derive the code from this proof.

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, and to assemble the Proof (...) code expressions one by one.

Begin with the left-most leaf “use arg”. This rule corresponds to the code x^A ,

$$\text{Proof}(\Gamma_1, \alpha \vdash \alpha) = x^A .$$

Here x^A must be a proof of the premise α in the sequent $\Gamma_1, \alpha \vdash \alpha$. So, we need to use the same x^A when we write the code for the previous rule, “create function”:

$$\text{Proof}(\Gamma_1 \vdash \alpha \Rightarrow \alpha) = (x^A \Rightarrow \text{Proof}(\Gamma_1, \alpha \vdash \alpha)) = (x^A \Rightarrow x) .$$

The right-most leaf “use arg” corresponds to the code $f^{:(A \Rightarrow A) \Rightarrow B}$, where f is the premise contained in Γ_1 . So we can write

$$\text{Proof}(\Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta) = f^{:(A \Rightarrow A) \Rightarrow B} .$$

The previous rule, “use function”, combines the two preceding proofs:

$$\begin{aligned} &\text{Proof}((\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta) \\ &= \text{Proof}(\Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta) (\text{Proof}(\Gamma_1 \vdash \alpha \Rightarrow \alpha)) \\ &= f(x^A \Rightarrow x) . \end{aligned}$$

Keep going backwards and find that the rule applied before “use function” is “create function”. We need to provide the same $f^{:(A \Rightarrow A) \Rightarrow B}$ as in the premise above, and so we obtain the code

$$\begin{aligned} &\text{Proof}(\emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta) \\ &= f^{:(A \Rightarrow A) \Rightarrow B} \Rightarrow \text{Proof}((\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta) \\ &= f^{:(A \Rightarrow A) \Rightarrow B} \Rightarrow f(x^A \Rightarrow x) . \end{aligned}$$

This is the final code expression that implements the type $((A \Rightarrow A) \Rightarrow B) \Rightarrow B$. In this way, we have systematically derived the code from the type signature of a function. This function can be implemented 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. If we *somewhat* 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 combine the proof rules for a given initial sequent. This is so because the rules of Table 5.7 do not provide an algorithm for finding a proof tree automatically for a given initial sequent. It turns out that such an algorithm exists (the “LJT algorithm”, see Appendix C). That algorithm can find proofs and derive code from type signatures containing tuples, disjunctive types, and function types (if the given type signature can be implemented).

The library `curryhoward`¹ implements the LJT algorithm. Here are some examples of using this library. We will run the `ammonite`² shell to load the library more easily.

Consider the type signature

$$\forall(A, B). (((A \Rightarrow B) \Rightarrow A) \Rightarrow B) \Rightarrow B .$$

It is not immediately clear whether it is even possible to implement a function with this type signature. It turns out that it *is* possible, and the code can be derived automatically with help of the LJT algorithm. The library does this via the method `implement`:

```
@ import $ivy.`io.chymyst:curryhoward:0.3.7`, io.chymyst.ch._

@ def f[A, B]: (((A => B) => A) => A) => B = implement
defined function f

@ println(f.lambdaTerm.prettyPrint)
a => a (b => b (c => a (d => c)))
```

The code $a \Rightarrow a (b \Rightarrow b (c \Rightarrow a (d \Rightarrow c)))$ was derived automatically for the function `f`. The function `f` was compiled and is ready to be used in any subsequent code.

A compile-time error occurs when trying to use a type signature that cannot be implemented as a fully parametric function:

```
@ def g[A, B]: ((A => B) => A) => A = implement
cmd3.sc:1: type ((A => B) => A) => A cannot be implemented
def g[A, B]: ((A => B) => A) => A = implement
^
Compilation Failed
```

The logical formula corresponding to this type signature is

$$\forall(\alpha, \beta). ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta . \quad (5.9)$$

This formula is known as “Peirce’s law”.³ It is another example showing that the logic of types in functional programming languages is not Boolean. Peirce’s law is a true theorem in Boolean logic but does not hold in the constructive logic (i.e. it cannot be derived using the proof rules of Table 5.7). If we try to implement `g[A, B]` with the type signature shown above, we will fail to write fully parametric code for `g` that compiles without type errors. We will fail 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 5.2.2 cannot be implemented as a fully parametric function:

```
@ 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
def bad[A, B, C](g: A => Either[B, C]): Either[A => B, A => C] = implement
^
Compilation Failed
```

¹<https://github.com/Chymyst/curryhoward>

²<http://ammonite.io/#Ammonite-Shell>

³https://en.wikipedia.org/wiki/Peirce%27s_law

5.3 Solved examples: 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

$$\forall(\alpha, \beta). \alpha \Rightarrow (\beta \Rightarrow \alpha) \quad (5.10)$$

corresponds to the code of the function

```
def f[A, B]: A => (B => A) = { x => _ => x }
```

With the CH correspondence in mind, we may say that the function `f`'s code “is” the proof of the proposition (5.10). In this sense, the *existence* of the code `x => _ => x` with the type $A \Rightarrow (B \Rightarrow A)$ “is” a proof of the logical formula (5.10).

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 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 logic formulas (such as $A \vee B$ and $A \wedge B$).

The most important use of the type notation is for writing equations with types. Can we write type equations using the arithmetic intuition, such as

$$(A + B) \times C = A \times C + B \times C \quad ? \quad (5.11)$$

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. (5.11) into the logic formula

$$\forall(A, B, C). (A \vee B) \wedge C = (A \wedge C) \vee (B \wedge C) \quad . \quad (5.12)$$

This formula is the well-known “distributive law”⁴ 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 (5.12) means that the two formulas hold,

$$\forall(A, B, C). (A \vee B) \wedge C \Rightarrow (A \wedge C) \vee (B \wedge C) \quad , \quad (5.13)$$

$$\forall(A, B, C). (A \wedge C) \vee (B \wedge C) \Rightarrow (A \vee B) \wedge C \quad . \quad (5.14)$$

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) = ???
```

In the type notation, these type signatures are written as

$$f_1 : \forall(A, B, C). (A + B) \times C \Rightarrow A \times C + B \times C \quad ,$$

$$f_2 : \forall(A, B, C). A \times C + B \times C \Rightarrow (A + B) \times C \quad .$$

Since the two logical formulas (5.13)–(5.14) 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. (5.13)–(5.14). 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 do pattern matching on the argument:

⁴https://en.wikipedia.org/wiki/Distributive_property#Rule_of_replacement

```
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.
}
```

Orously as a requirement that an arbitrary value $x: \text{Either}[\text{A}, \text{B}], \text{C}$ is mapped by `f1` to some value $y: \text{Either}[(\text{A}, \text{C}), (\text{B}, \text{C})]$ 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,

$$\forall x^{:(A+B)\times C}. f_2(f_1(x)) = x , \quad \forall y^{:A\times C+B\times C}. f_1(f_2(y)) = y .$$

If we show that these equations hold, it will follow that all the information in a value $x^{:(A+B)\times C}$ is completely preserved inside the value $y \triangleq f_1(x)$; the original value x can be recovered as $x = f_2(y)$. Conversely, all the information in a value $y^{:A\times C+B\times C}$ is preserved inside $x \triangleq f_2(y)$ and can be recovered by applying `f1`. Since the values $x^{:(A+B)\times C}$ and $y^{:A\times C+B\times C}$ are arbitrary, it will follow 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 $f_1: P \Rightarrow Q$ and $f_2: Q \Rightarrow P$ that are inverses of each other. We can write these conditions using the notation $(f_1 \circ f_2)(x) \triangleq f_2(f_1(x))$ as

$$f_1 \circ f_2 = \text{id} , \quad f_2 \circ f_1 = \text{id} .$$

(We omit type annotations since we already checked that the types match. In Scala, the forward composition $f_1 \circ f_2$ is the function `f1 andThen f2`.) If these conditions hold, there is a one-to-one correspondence between the 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 $f_1 \circ f_2 = \text{id}$. Applying $f_1 \circ f_2$ means to apply `f1` and then to apply `f2` to the result. Begin by applying `f1` 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 `f2` to those values, which maps them back to a tuple `(Left(a), c)` or a tuple `(Right(b), c)` respectively, according to the code of `f2`. These tuples are exactly the value x we started with. So, applying $f_1 \circ f_2$ to an arbitrary $x^{:(A+B)\times C}$ does not change the value x ; this is the same as to say that $f_1 \circ f_2 = \text{id}$.

To check whether $f_2 \circ f_1 = \text{id}$, we apply `f2` 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 `f2` and then `f1` will always recover the initial value y . In other words, $f_2 \circ f_1 = \text{id}$.

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 then visually clear that no information is lost and that the original values are restored by function compositions $f_1 \circ f_2$ or $f_2 \circ f_1$.

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 rig-

We find that the logical identity (5.12) leads to an equivalence of the corresponding types,

$$(A + B) \times C \cong A \times C + B \times C . \quad (5.15)$$

To get Eq. (5.15) from Eq. (5.12), we just need to mentally replace the disjunction operations \vee by $+$ and the conjunctions \wedge by \times , as if we were converting to an arithmetic expression.

Consider another example of a logical identity: the associativity law for conjunction,

$$(\alpha \wedge \beta) \wedge \gamma = \alpha \wedge (\beta \wedge \gamma) . \quad (5.16)$$

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

$$\text{True} \vee \alpha = \text{True} . \quad (5.17)$$

Since the CH correspondence maps the logical constant *True* into the unit type $\mathbb{1}$, the type equivalence corresponding to Eq. (5.17) 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 $g_1 : \mathbb{1} + A \Rightarrow \mathbb{1}$ and $g_2 : \mathbb{1} \Rightarrow \mathbb{1} + A$ will not satisfy $g_1 \circ g_2 = \text{id}$. To verify this, we note that $g_2 : \mathbb{1} \Rightarrow \mathbb{1} + A$ must have type signature

```
def g2[A]: Unit => Option[A] = ???
```

Such a function must always return `None`, since a fully parametric function cannot produce values of an arbitrary type `A` from scratch. Therefore, $g_1 \circ g_2$ is also a function that always returns `None`. The function $g_1 \circ g_2$ has type signature $\mathbb{1} + A \Rightarrow \mathbb{1} + A$ 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

$$\forall(A, B, C). (A \wedge B) \vee C = (A \vee C) \wedge (B \vee C) , \quad (5.18)$$

which is “dual” to the law (5.12), i.e. it is obtained from Eq. (5.12) by swapping all conjunctions (\wedge) with disjunctions (\vee). In logic, a dual formula to an identity is often also an identity. The CH correspondence maps Eq. (5.18) into the type equation

$$\forall(A, B, C). (A \times B) + C = (A + C) \times (B + C) . \quad (5.19)$$

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 the function $g_3 : (A + C) \times (B + C) \Rightarrow A \times B + C$:

```
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 c_1, c_2 . After evaluating g_3 , we will not be able to restore *both* c_1 and c_2 – no matter what code we write for g_4 . So, the composition $g_3 \circ g_4$ cannot be equal to the identity function. The type equation (5.19) is incorrect.

We conclude 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 $P \Rightarrow Q$ and $Q \Rightarrow P$. However, it is not guaranteed that these functions are inverses of each other, i.e. that the type conversions $P \Rightarrow Q$ or $Q \Rightarrow P$ 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 we can compute *some* value $x:X$ of a given type X when the proposition $\mathcal{CH}(X)$ holds. 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. (5.12) leads to the type equivalence (5.15), which looks like a standard identity of arithmetic, such as

$$(1 + 10) \times 20 = 1 \times 20 + 10 \times 20 .$$

The logical identity in Eq. (5.18), which does *not* yield a type equivalence, leads to an incorrect arithmetic equation 5.19, e.g. $(1 \times 10) + 20 \neq (1 + 20) \times (10 + 20)$. Similarly, the associativity law (5.16) leads to a type equivalence and to the arithmetic identity

$$(a \times b) \times c = a \times (b \times c) ,$$

while the logical identity in Eq. (5.17), 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, with the corresponding type equivalences. In all rows, quantifiers such as $\forall a$ or $\forall(A, B)$ are implied when 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 `Unit`, defined in Scala as `Nothing`. Reasoning about the void type needs a special technique that we will now develop while verifying the type isomorphisms $\text{Unit} \times A \cong \text{Unit}$ and $\text{Unit} + A \cong A$.

Example 5.3.2.1 Verify the type equivalence $\text{Unit} \times A \cong \text{Unit}$.

Solution Recall that the type notation $\text{Unit} \times A$ 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 other words, the set of values of the type `(Nothing, A)` is empty; but that is the definition of the type `Nothing`. The types `(Nothing, A)` (denoted by $\text{Unit} \times A$) and `Nothing` (denoted by `Unit`) are both void and therefore equivalent.

Example 5.3.2.2 Verify the type equivalence $\text{Unit} + A \cong A$.

Logical identity	Type equivalence (if it holds)
$\text{True} \vee \alpha = \text{True}$	$\mathbb{1} + A \not\cong \mathbb{1}$
$\text{True} \wedge \alpha = \alpha$	$\mathbb{1} \times A \cong A$
$\text{False} \vee \alpha = \alpha$	$\mathbb{0} + A \cong A$
$\text{False} \wedge \alpha = \text{False}$	$\mathbb{0} \times A \cong \mathbb{0}$
$\alpha \vee \beta = \beta \vee \alpha$	$A + B \cong B + A$
$\alpha \wedge \beta = \beta \wedge \alpha$	$A \times B \cong B \times A$
$(\alpha \vee \beta) \vee \gamma = \alpha \vee (\beta \vee \gamma)$	$(A + B) + C \cong A + (B + C)$
$(\alpha \wedge \beta) \wedge \gamma = \alpha \wedge (\beta \wedge \gamma)$	$(A \times B) \times C \cong A \times (B \times C)$
$(\alpha \vee \beta) \wedge \gamma = (\alpha \wedge \gamma) \vee (\beta \wedge \gamma)$	$(A + B) \times C \cong A \times C + B \times C$
$(\alpha \wedge \beta) \vee \gamma = (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$	$(A \times B) + C \not\cong (A + C) \times (B + C)$

Table 5.5: Logic identities with disjunction and conjunction, and the corresponding equivalences of types.

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 $f_1 : \mathbb{0} + A \Rightarrow A$ and $f_2 : A \Rightarrow \mathbb{0} + A$ such that $f_1 \circ f_2 = \text{id}$ and $f_2 \circ f_1 = \text{id}$.

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 $x:\text{Nothing}$, 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; so 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 be given there.
}
```

For the same reason, there is only one implementation of the function f_2 ,

```
def f2[A]: A => Either[Nothing, A] = { y => Right(y) }
```

It is clear from the code that the functions f_1 and f_2 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:

We can write the functions `toLeft` and `toRight` in a code notation as

$$\begin{aligned} \text{toLeft}^{A,B} &= x:A \Rightarrow x:A + \mathbb{0}:B , \\ \text{toRight}^{A,B} &= y:B \Rightarrow \mathbb{0}:A + y:B . \end{aligned}$$

```
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")
```

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 is left unspecified and should be determined by matching the types of other expressions.

In the notation $0 + y^B$, we use the symbol 0 rather than an ordinary zero (0), to avoid suggesting that 0 is a value of type 0. The void type 0 has no values, unlike the `Unit` type, 1, which has a value denoted by 1 in the code notation.

Example 5.3.2.3 Verify the type equivalence $A \times 1 \cong A$.

Solution The corresponding Scala types are the tuple `(A, Unit)` and the type `A`. We need to implement functions $f_1 : \forall A. A \times 1 \Rightarrow A$ and $f_2 : \forall A. A \Rightarrow A \times 1$ 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

$$\begin{aligned} f_1 &= a:A \times 1 \Rightarrow a , \\ f_2 &= a:A \Rightarrow a \times 1 , \end{aligned}$$

where we denoted by 1 the value () of the `Unit` type. We find

$$\begin{aligned} (f_1 \circ f_2)(a:A \times 1) &= f_2(f_1(a \times 1)) = f_2(a) = a \times 1 , \\ (f_2 \circ f_1)(a:A) &= f_1(f_2(a)) = f_1(a \times 1) = a . \end{aligned}$$

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 $a:A$,

$$\begin{aligned} f_1 \circ f_2 &= (a \times 1 \Rightarrow a) \circ (a \Rightarrow a \times 1) = (a \times 1 \Rightarrow a \times 1) = \text{id}^{A \times 1} , \\ f_2 \circ f_1 &= (a \Rightarrow a \times 1) \circ (a \times 1 \Rightarrow a) = (a \Rightarrow a) = \text{id}^A . \end{aligned}$$

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]
```

by using only 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 f_1 and f_2 are inverses of each other. To verify that rigorously, we need show that $f_1 \text{ andThen } f_2$ is equal to an identity function. The function $f_1 \text{ andThen } f_2$ applies f_2 to the result of f_1 . The code of f_1 contains two `case ...` lines, each returning a result. So, we need to apply f_2 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 functions f_1 and f_2 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]`.

5 The logic of types. III. The Curry-Howard correspondence

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` contains a pair of functions with types `A => Either[B, A]` and `B => Either[B, A]`. One of these functions is chosen depending on whether the argument of `f1` has type $A + \emptyset$ or $\emptyset + B$. So, we may write the code of `f1` as

$$f_1 \triangleq x^{A+B} \Rightarrow \begin{cases} \text{if } x = a^{A+0} + 0^B & : 0^B + a^A \\ \text{if } x = 0^A + b^B & : b^B + 0^A \end{cases}$$

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)
}
```

$$f_1 \triangleq \begin{array}{c|cc} & B & A \\ \hline A & 0 & a^A \Rightarrow a \\ B & b^B \Rightarrow b & 0 \end{array} .$$

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 marks the input types of the functions.

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(b)    => Right(b)
  case Right(a)   => Left(a)
}
```

$$f_2 \triangleq \begin{array}{c|cc} & A & B \\ \hline B & 0 & y^B \Rightarrow y \\ A & x^A \Rightarrow x & 0 \end{array} .$$

The forward composition $f_1 \circ f_2$ is computed by the standard rules of row-by-column matrix multiplication.⁵ The \emptyset terms are omitted, and the remaining expressions are composed:

$$\begin{aligned} f_1 \circ f_2 &= \begin{array}{c|cc} & B & A \\ \hline A & 0 & a^A \Rightarrow a \\ B & b^B \Rightarrow b & 0 \end{array} \circ \begin{array}{c|cc} & A & B \\ \hline B & 0 & y^B \Rightarrow y \\ A & x^A \Rightarrow x & 0 \end{array} \\ &\quad \text{use matrix multiplication :} \\ &= \begin{array}{c|cc} & A & B \\ \hline A & (a^A \Rightarrow a) \circ (x^A \Rightarrow x) & 0 \\ B & 0 & (b^B \Rightarrow b) \circ (y^B \Rightarrow y) \end{array} \\ &\quad \text{function composition :} \\ &= \begin{array}{c|cc} & A & B \\ \hline A & \text{id} & 0 \\ B & 0 & \text{id} \end{array} = \text{id}^{A+B \Rightarrow A+B} . \end{aligned}$$

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 $\parallel B \quad A \parallel$ of

f_1 with the input column $\begin{array}{c|cc} B \\ \hline A \end{array}$ of f_2 . Once we verified that all types match, we may omit the type

⁵https://en.wikipedia.org/wiki/Matrix_multiplication

annotations and write the derivation as

$$\begin{aligned}
 f_1 \circ f_2 &= \left\| \begin{array}{cc} \emptyset & a:A \Rightarrow a \\ b:B \Rightarrow b & \emptyset \end{array} \right\| \left\| \begin{array}{cc} \emptyset & y:B \Rightarrow y \\ x:A \Rightarrow x & \emptyset \end{array} \right\| \\
 \text{use matrix multiplication : } &= \left\| \begin{array}{cc} (a:A \Rightarrow a) \circ (x:A \Rightarrow x) & \emptyset \\ \emptyset & (b:B \Rightarrow b) \circ (y:B \Rightarrow y) \end{array} \right\| \\
 \text{function composition : } &= \left\| \begin{array}{cc} \text{id} & \emptyset \\ \emptyset & \text{id} \end{array} \right\| = \text{id} .
 \end{aligned}$$

The identity function is represented by the diagonal matrix $\left\| \begin{array}{cc} \text{id} & \emptyset \\ \emptyset & \text{id} \end{array} \right\|$, just as in standard matrix notation.

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 $(A + B) + C \cong A + (B + C)$. The equivalences $(A + B) + C \cong A + (B + C)$ and $(A \times B) \times C \cong A \times (B \times C)$, which was verified in Section 5.3.1, permit us to write the type notation as $A + B + C$ and $A \times B \times C$ without the parentheses.

Exercise 5.3.2.7 Verify the type equivalence

$$(A + B) \times (A + B) = A \times A + 2 \times A \times B + B \times B ,$$

where 2 denotes the Boolean type.

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`, and so the total number of possible different strings will be finite (at least, for any given 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, and since we may assume that we are already working with the largest possible computer, then 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 $|\text{String}|$.

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

$$\{(a, b) : a \in A, b \in B\}$$

is also known as 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 disjoint 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 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 such as `Either[A, B]` 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 the pairs and $A + B$ for the disjunctive types, 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 $1 + 1 + A$ and $2 + A$ respectively (the type Boolean is denoted by 2 since it has only two distinct values).

One could easily write code to convert 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))
}
```

A sign of trouble is the presence of an arbitrary choice in this code. In `f1`, we could map `None` to `Left(false)` or to `Left(true)`, and adjust the rest of the code accordingly; the type equivalence would still hold. So, formally speaking, 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 is application-dependent. This type equivalence is “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 $1 + A$, and `Either[Unit, A]` is written also as $1 + A$. This already indicates, by looking at the notation alone, that the types are equivalent. But let us verify explicitly that the type notation is not misleading here.

To establish 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] = ???
```

such that $f_1 \circ f_2 = \text{id}$ and $f_2 \circ f_1 = \text{id}$. 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 f_1 and f_2 are inverses of each other; this verifies the type equivalence.

The logic identity is $\text{True} \vee A = \text{True} \vee A$ and holds trivially. It remains to check the arithmetic identity, which relates the number of distinct values of types `Option[A]` and `Either[Unit, A]`. Assume that the number of distinct values 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 number of distinct values 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 solution was found immediately when we wrote the type notation, $1 + A$, for the given types.

5.3.4 Type equivalence involving function types

Until now, we have looked at product types and disjunctive types. Let us now consider type constructions involving function types.

Consider two types A and B , whose cardinalities are known as $|A|$ and $|B|$. What is the cardinality of the set of all maps between given sets A and B ? In other words, how many distinct values does the function type $A \Rightarrow B$ have? A function $f: A \Rightarrow B$ needs to select a value of type B for each possible value of type A . Therefore, the number of different functions $f: A \Rightarrow B$ is $|B|^{|A|}$ (the arithmetic exponent, $|B|$ to the power $|A|$).

For the types $A = B = \text{Int}$, we have $|A| = |B| = 2^{32}$, and so the estimate will give

$$|A \Rightarrow B| = (2^{32})^{(2^{32})} = 2^{32 \times 2^{32}} = 2^{2^{37}} \approx 10^{4.1 \times 10^{10}} .$$

In fact, most of these functions will map integers to integers in a complicated (and practically useless) way and will be impossible to implement on a realistic computer because their code will be much longer than the available memory. So, the number of practically implementable functions of type $A \Rightarrow B$ is often much smaller than $|B|^{|A|}$. Nevertheless, the estimate $|B|^{|A|}$ is useful since it shows the number of distinct functions that are possible in principle.

Let us now see what logic identities and arithmetic identities are available for type expressions involving function types. Table 5.6 lists the available identities and the corresponding type equivalences. (In the last column, we defined $a \triangleq |A|$, $b \triangleq |B|$, and $c \triangleq |C|$ 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 $A \Rightarrow B + C$ (although there is an identity for $A \Rightarrow B \times C$). The presence of type expressions of the form $A \Rightarrow B + C$ makes type reasoning more complicated because they cannot be transformed into equivalent formulas with simpler parts.

We will now prove some of the type identities in Table 5.6.

Example 5.3.4.1 Verify the type equivalence $1 \Rightarrow A \cong A$.

Solution Recall that the type notation $1 \Rightarrow A$ means the Scala function type `Unit => A`. There is only one value of type `Unit`, so the choice of a function of the type `Unit => A` is the same as the choice of a value of type `A`. Thus, the number of distinct values of the type $1 \Rightarrow A$ is $|A|$, and the arithmetic identity holds.

To verify the type equivalence explicitly, we need to implement two functions

```
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 `()`:

Logical identity (if holds)	Type equivalence	Arithmetic identity
$(True \Rightarrow \alpha) = \alpha$	$\mathbb{1} \Rightarrow A \cong A$	$a^1 = a$
$(False \Rightarrow \alpha) = True$	$\mathbb{0} \Rightarrow A \cong \mathbb{1}$	$a^0 = 1$
$(\alpha \Rightarrow True) = True$	$A \Rightarrow \mathbb{1} \cong \mathbb{1}$	$1^\alpha = 1$
$(\alpha \Rightarrow False) \neq False$	$A \Rightarrow \mathbb{0} \not\cong \mathbb{0}$	$0^\alpha \neq 0$
$(\alpha \vee \beta) \Rightarrow \gamma = (\alpha \Rightarrow \gamma) \wedge (\beta \Rightarrow \gamma)$	$A + B \Rightarrow C \cong (A \Rightarrow C) \times (B \Rightarrow C)$	$c^{a+b} = c^a \times c^b$
$(\alpha \wedge \beta) \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)$	$A \times B \Rightarrow C \cong A \Rightarrow B \Rightarrow C$	$c^{a \times b} = (c^b)^a$
$\alpha \Rightarrow (\beta \wedge \gamma) = (\alpha \Rightarrow \beta) \wedge (\alpha \Rightarrow \gamma)$	$A \Rightarrow B \times C \cong (A \Rightarrow B) \times (A \Rightarrow C)$	$(b \times c)^a = b^a \times c^a$

Table 5.6: Logical identities with implication, and the corresponding type equivalences and arithmetic identities.

```
def f1[A]: (Unit => A) => A = (h: Unit => A) => h()
```

Implementing f_2 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 f_1 and f_2 are inverses of each other. Let us perform the proof using Scala code and then using the code notation.

Writing Scala code, compute $f_1(f_2(x))$ for an arbitrary $x:A$. Substituting the code, we get

```
f1(f2(x)) == f1(_ => x) == (_ => x)() == x
```

Now compute $f_2(f_1(h))$ for arbitrary $h: \text{Unit} \Rightarrow A$ in Scala code:

```
f2(f1(h)) == f2(h()) == { _ => h() }
```

How can we show that the function $\{ _ \Rightarrow h() \}$ 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 $\{ _ \Rightarrow h() \}$ 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

$$\begin{aligned} f_1 &\triangleq h : \mathbb{1} \Rightarrow A \Rightarrow h(1) , \\ f_2 &\triangleq x : A \Rightarrow 1 \Rightarrow x . \end{aligned}$$

Now write the function compositions in both directions:

$$f_1 ; f_2 = (h : \mathbb{1} \Rightarrow A \Rightarrow h(1)) ; (x : A \Rightarrow 1 \Rightarrow x)$$

$$\text{compute composition : } = (h \Rightarrow 1 \Rightarrow h(1))$$

$$\text{note that } 1 \Rightarrow h(1) \text{ is the same as } h : = (h \Rightarrow h) = \text{id} .$$

$$f_2 ; f_1 = (x : A \Rightarrow 1 \Rightarrow x) ; (h : \mathbb{1} \Rightarrow A \Rightarrow h(1))$$

$$\text{compute composition : } = x \Rightarrow (1 \Rightarrow x)(1)$$

$$\text{apply function : } = (x \Rightarrow x) = \text{id} .$$

The type $\mathbb{1} \Rightarrow A$ is equivalent to the type A , but these types are not the same. The most important difference between these types is that a value of type A is available immediately, while a value of

type $\mathbb{1} \Rightarrow A$ 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 $\mathbb{1} \Rightarrow A$ may represent an “on-call” value of type A ; that is, a value computed on demand every time. (See Section 2.6.3 for more details about “on-call” values.)

The void type $\mathbb{0}$ needs special reasoning.

Example 5.3.4.2 Verify the type equivalence $\mathbb{0} \Rightarrow A \cong \mathbb{1}$.

Solution What could be a function $f: \mathbb{0} \Rightarrow A$ 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; or at least it 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> ???
scala.NotImplementedError: an implementation is missing
  scala.Predef$.qmark$qqmark$qqmark(Predef.scala:288)
```

Since the function `absurd` is impossible to apply to an argument, this error will never happen. So, we can pretend that the result value (which will never be computed or returned) has any required type, e.g. the type A .

Let us now verify that there exists *only one* function of type $\mathbb{0} \Rightarrow A$. Suppose there are two such functions, $f: \mathbb{0} \Rightarrow A$ and $g: \mathbb{0} \Rightarrow A$. Are f and g different functions? We would see that f and g are different only if we had a value x such that $f(x) \neq g(x)$, where x must have type $\mathbb{0}$. However, there are *no* values of type $\mathbb{0}$, and so we will never be able to find such x . It follows that any two functions f and g of type $\mathbb{0} \Rightarrow A$ are equal. In other words, there exists only *one* distinct value of type $\mathbb{0} \Rightarrow A$; i.e. the cardinality of the type $\mathbb{0} \Rightarrow A$ is 1. So, the type $\mathbb{0} \Rightarrow A$ is equivalent to the type $\mathbb{1}$.

Example 5.3.4.3 Show that $A \Rightarrow \mathbb{0} \not\cong \mathbb{0}$ and $A \Rightarrow \mathbb{0} \not\cong \mathbb{1}$.

Solution To prove that two types are *not* equivalent, it is sufficient to show that their type cardinalities are different. Let us determine the cardinality of the type $A \Rightarrow \mathbb{0}$, assuming that the cardinality of A is known. We note that a function of type, say, $\text{Int} \Rightarrow \mathbb{0}$ is impossible to implement. (If we had such a function $f: \text{Int} \Rightarrow \mathbb{0}$, we could evaluate, say, $x \triangleq f(123)$ and obtain a value x of type $\mathbb{0}$, which is impossible by definition of the type $\mathbb{0}$. It follows that $|\text{Int} \Rightarrow \mathbb{0}| = 0$. However, Example 5.3.4.2 shows that $\mathbb{0} \Rightarrow \mathbb{0}$ has cardinality 1. So, the cardinality $|A \Rightarrow \mathbb{0}| = 1$ if the type A is itself $\mathbb{0}$ but $|A \Rightarrow \mathbb{0}| = 0$ for all other types A . We conclude that the type $A \Rightarrow \mathbb{0}$ is not equivalent to $\mathbb{0}$ or $\mathbb{1}$ for all A ; it is equivalent to $\mathbb{0}$ only for non-void types A .

Example 5.3.4.4 Verify the type equivalence $A \Rightarrow \mathbb{1} \cong \mathbb{1}$.

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 that type. 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

$$f: A \Rightarrow \mathbb{1} \triangleq (_ \Rightarrow \mathbb{1}) \quad .$$

We can show that there exist only *one* distinct function of type $A \Rightarrow \mathbb{1}$ (that is, the type $A \Rightarrow \mathbb{1}$ 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 $f(x) = \mathbb{1}$ and $g(x) = \mathbb{1}$ for all x . So, any two functions f and g of type $A \Rightarrow \mathbb{1}$ must be equal to each other. Any type having cardinality 1 is equivalent to the `Unit` type, $\mathbb{1}$. So $A \Rightarrow \mathbb{1} \cong \mathbb{1}$.

Example 5.3.4.5 Verify the type equivalence

$$A + B \Rightarrow C \cong (A \Rightarrow C) \times (B \Rightarrow C) .$$

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 $(A \Rightarrow C, B \Rightarrow C)$. Can we produce the first part of that pair? Computing a function of type $A \Rightarrow C$ 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 $\text{Either}[A, B] \Rightarrow C$ called, say, h . We can apply that function to $\text{Left}(a)$ and obtain a value of type C as required. So, a function of type $A \Rightarrow C$ is computed as $a \Rightarrow h(\text{Left}(a))$. Similarly, we produce a function of type $B \Rightarrow C$. 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)))
```

A code notation for this function is

$$\begin{aligned} f_1 : (A + B \Rightarrow C) &\Rightarrow (A \Rightarrow C) \times (B \Rightarrow C) , \\ f_1 \triangleq h^{A+B \Rightarrow C} &\Rightarrow (a^A \Rightarrow h(a + 0^B)) \times (b^B \Rightarrow h(0^A + b)) . \end{aligned}$$

For the function f_2 , 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]: ((A => C, B => C)) => Either[A, B] => C = { case (f, g) =>  
  {  
    case Left(a) => f(a)  
    case Right(b) => g(b)  
  }  
}
```

A code notation for this function can be written as

$$\begin{aligned} f_2 : (A \Rightarrow C) \times (B \Rightarrow C) &\Rightarrow A + B \Rightarrow C , \\ f_2 \triangleq f^{A \Rightarrow C} \times g^{B \Rightarrow C} &\Rightarrow \left| \begin{array}{c|c} & C \\ \hline A & a \Rightarrow f(a) \\ B & b \Rightarrow g(b) \end{array} \right| . \end{aligned}$$

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 simplify the functions, e.g. $a \Rightarrow f(a)$ into f , and write

$$f_2 \triangleq f^{A \Rightarrow C} \times g^{B \Rightarrow C} \Rightarrow f^{A \Rightarrow C} \times g^{B \Rightarrow C} \Rightarrow \left| \begin{array}{c|c} & C \\ \hline A & f \\ B & g \end{array} \right| .$$

It remains to verify that $f_1 \circ f_2 = \text{id}$ and $f_2 \circ f_1 = \text{id}$. To compute the composition $f_1 \circ f_2$, we write (omitting types)

$$\begin{aligned} f_1 \circ f_2 &= (h \Rightarrow (a \Rightarrow h(a + 0)) \times (b \Rightarrow h(0 + b))) \circ (f \times g \Rightarrow \left| \begin{array}{c} f \\ g \end{array} \right|) \\ \text{compute composition : } &= h \Rightarrow \left| \begin{array}{c} a \Rightarrow h(a + 0) \\ b \Rightarrow h(0 + b) \end{array} \right| . \end{aligned}$$

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 $A + B \Rightarrow C$) in the matrix notation:

$$h \triangleq \left| \begin{array}{c|c|c} & & C \\ \hline A & a \Rightarrow p(a) \\ B & b \Rightarrow q(b) \end{array} \right| = \left| \begin{array}{c|c|c} & & C \\ \hline A & p \\ B & q \end{array} \right| ,$$

where $p : A \Rightarrow C$ and $q : B \Rightarrow C$ are new arbitrary functions. Since we already checked the types, we can omit all type annotations and express h as

$$h \triangleq \left| \begin{array}{c} p \\ q \end{array} \right| .$$

To evaluate expressions such as $h(a + \mathbb{0})$ and $h(\mathbb{0} + b)$, we need to use one of the rows of the column matrix h . The correct row will be selected *automatically* by the standard rules of matrix multiplication if we place a row vector to the left of the matrix, and if we use the convention of omitting any terms containing $\mathbb{0}$:

$$\begin{aligned} \left| \begin{array}{c} a \\ \mathbb{0} \end{array} \right| \triangleright \left| \begin{array}{c} p \\ q \end{array} \right| &= a \triangleright p , \\ \left| \begin{array}{c} \mathbb{0} \\ b \end{array} \right| \triangleright \left| \begin{array}{c} p \\ q \end{array} \right| &= b \triangleright q . \end{aligned}$$

Here we used the symbol \triangleright to separate an argument from a function when the argument is written to the *left* of the function. The symbol \triangleright (pronounced “pipe”) is defined by $x \triangleright f \triangleq f(x)$. In Scala, this operation is available as `x.pipe(f)` as of Scala 2.13.

We write values of disjunctive types, such as $a + \mathbb{0}$, as if they were row vectors:

$$h(a + \mathbb{0}) = (a + \mathbb{0}) \triangleright h = \left| \begin{array}{c} a \\ \mathbb{0} \end{array} \right| \triangleright h . \quad (5.20)$$

With these notations, we can compute further:

$$\begin{aligned} h(a + \mathbb{0}) &= \left| \begin{array}{c} a \\ \mathbb{0} \end{array} \right| \triangleright \left| \begin{array}{c} p \\ q \end{array} \right| = a \triangleright p = p(a) , \\ h(\mathbb{0} + b) &= \left| \begin{array}{c} \mathbb{0} \\ b \end{array} \right| \triangleright \left| \begin{array}{c} p \\ q \end{array} \right| = b \triangleright q = q(b) . \end{aligned}$$

Now we can complete the proof of $f_1 \circ f_2 = \text{id}$:

$$\begin{aligned} f_1 \circ f_2 = h &\Rightarrow \left| \begin{array}{c} a \Rightarrow h(a + \mathbb{0}) \\ b \Rightarrow h(\mathbb{0} + b) \end{array} \right| \\ \text{previous equations : } &= \left| \begin{array}{c} p \\ q \end{array} \right| \Rightarrow \left| \begin{array}{c} a \Rightarrow p(a) \\ b \Rightarrow q(b) \end{array} \right| \\ \text{simplify functions : } &= \left(\left| \begin{array}{c} p \\ q \end{array} \right| \Rightarrow \left| \begin{array}{c} p \\ q \end{array} \right| \right) = \text{id} . \end{aligned}$$

To prove that $f_2 \circ f_1 = \text{id}$, use the notation (5.20):

$$f_2 \circ f_1 = \left(f \times g \Rightarrow \begin{array}{|c|} \hline f \\ \hline g \\ \hline \end{array} \right) \circ (h \Rightarrow (a \Rightarrow h(a + 0)) \times (b \Rightarrow h(0 + b)))$$

compute composition : $= f \times g \Rightarrow (a \Rightarrow | a \ 0 | \triangleright \begin{array}{|c|} \hline f \\ \hline g \\ \hline \end{array}) \times (b \Rightarrow | 0 \ b | \triangleright \begin{array}{|c|} \hline f \\ \hline g \\ \hline \end{array})$

matrix notation : $= f \times g \Rightarrow (a \Rightarrow \underline{a \triangleright f}) \times (b \Rightarrow \underline{b \triangleright g})$

definition of \triangleright : $= f \times g \Rightarrow (\underline{a \Rightarrow f(a)}) \times (\underline{b \Rightarrow g(b)})$

simplify functions : $= (f \times g \Rightarrow f \times g) = \text{id} .$

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. Given this notation, the proof for $f_1 \circ f_2 = \text{id}$ can be written as

$$f_1 \circ f_2 = (h \Rightarrow (a \Rightarrow (a + 0) \triangleright h) \times (b \Rightarrow (0 + b) \triangleright h)) \circ \left(f \times g \Rightarrow \begin{array}{|c|} \hline f \\ \hline g \\ \hline \end{array} \right)$$

compute composition : $= h \Rightarrow \left\| \begin{array}{|c|} \hline a \Rightarrow | a \ 0 | \triangleright h \\ \hline b \Rightarrow | 0 \ b | \triangleright h \\ \hline \end{array} \right\| = \left\| \begin{array}{|c|} \hline p \\ \hline q \\ \hline \end{array} \right\| \Rightarrow \left\| \begin{array}{|c|} \hline a \Rightarrow | a \ 0 | \triangleright \begin{array}{|c|} \hline p \\ \hline q \\ \hline \end{array} \\ \hline b \Rightarrow | 0 \ b | \triangleright \begin{array}{|c|} \hline p \\ \hline q \\ \hline \end{array} \\ \hline \end{array} \right\|$

matrix notation : $= \left\| \begin{array}{|c|} \hline p \\ \hline q \\ \hline \end{array} \right\| \Rightarrow \left\| \begin{array}{|c|} \hline a \Rightarrow a \triangleright p \\ \hline b \Rightarrow b \triangleright q \\ \hline \end{array} \right\| = \left(\left\| \begin{array}{|c|} \hline p \\ \hline q \\ \hline \end{array} \right\| \Rightarrow \left\| \begin{array}{|c|} \hline p \\ \hline q \\ \hline \end{array} \right\| \right) = \text{id} .$

The code notation makes proofs shorter than they would be if we were manipulating code in Scala syntax. From now on, we will prefer to use the code notation in the proofs, keeping in mind that Scala code can be unambiguously recovered from the code notation, and vice versa.

Example 5.3.4.6 Verify the type equivalence

$$A \times B \Rightarrow C \cong A \Rightarrow B \Rightarrow C .$$

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 without ambiguity:

```
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:

$$f_1 = g^{:A \times B \Rightarrow C} \Rightarrow a^{:A} \Rightarrow b^{:B} \Rightarrow g(a \times b) ,$$

$$f_2 = h^{:A \Rightarrow B \Rightarrow C} \Rightarrow (a \times b)^{:A \times B} \Rightarrow h(a)(b) .$$

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 notation involving tupled arguments.

Compute the function composition $f_1 \circ f_2$:

$$\begin{aligned} f_1 \circ f_2 &= (g \Rightarrow a \Rightarrow b \Rightarrow g(a \times b)) \circ (h \Rightarrow a \times b \Rightarrow h(a)(b)) \\ \text{substitute } h = a \Rightarrow b \Rightarrow g(a \times b) : &= g \Rightarrow a \times b \Rightarrow g(a \times b) \\ \text{simplify function :} &= (g \Rightarrow g) = \text{id} \end{aligned}$$

Compute the function composition $f_2 \circ f_1$:

$$\begin{aligned} f_2 \circ f_1 &= (h \Rightarrow a \times b \Rightarrow h(a)(b)) \circ (g \Rightarrow a \Rightarrow b \Rightarrow g(a \times b)) \\ \text{substitute } g = a \times b \Rightarrow h(a)(b) : &= h \Rightarrow a \Rightarrow b \Rightarrow h(a)(b) \\ \text{simplify function } b \Rightarrow h(a)(b) : &= h \Rightarrow a \Rightarrow h(a) \\ \text{simplify function } a \Rightarrow h(a) \text{ to } h : &= (h \Rightarrow h) = \text{id} \end{aligned}$$

Exercise 5.3.4.7 Verify the type equivalence

$$(A \Rightarrow B \times C) \cong (A \Rightarrow B) \times (A \Rightarrow C) \quad .$$

5.4 Summary

What tasks can we perform now?

- Use the matrix notation and the pipe notation to write code that works on disjunctive types.
- 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.
- Convert a fully parametric type signature into a logical formula to:
 - Decide whether the type signature can be implemented in code.
 - If possible, derive the code using the CH correspondence.

What tasks cannot be performed with these tools?

- Automatically generate code for a recursive function. (The CH correspondence is based on propositional logic, which cannot describe recursion.)
- Automatically generate code satisfying a property (e.g. isomorphism). We may generate the code, but it is not guaranteed that properties will hold. The workaround is 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 depend on run-time values in an arbitrary way) – an advanced technique that Scala does not fully support. Programming languages such as Coq, Agda, and Idris support full dependent types, but cannot automatically generate code from dependent type signatures.
- Generate code using type constructors with known properties (e.g. `.map`).

As an example of using type constructors with properties, 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` type constructor, so the implementation of `q` is simple,

```
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 the logical formula

$$\mathcal{CH}(\text{Array}^A) \Rightarrow \mathcal{CH}(A \Rightarrow \text{Opt}^B) \Rightarrow \mathcal{CH}(\text{Array}^{\text{Opt}^B}) . \quad (5.21)$$

To derive an implementation, the algorithm would need to use the available `.map` method for `Array`. That method has a type signature such as

$$\text{map} : \forall(A, B). \text{Array}^A \Rightarrow (A \Rightarrow B) \Rightarrow \text{Array}^B .$$

To derive the \mathcal{CH} -proposition (5.21), the algorithm will need to assume that the \mathcal{CH} -proposition

$$\mathcal{CH}(\forall(A, B). \text{Array}^A \Rightarrow (A \Rightarrow B) \Rightarrow \text{Array}^B) \quad (5.22)$$

already holds, i.e. that Eq. (5.22) is one of the premises of a sequent to be proved. Reasoning about propositions such as Eq. (5.22) 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 guarantee finding a proof or showing the absence of a proof in all cases.

The constructive propositional logic (with the rules listed in Section 5.2.3) is **decidable** and has an algorithm that either finds a proof or disproves any given formula. However, that logic cannot handle premises containing type quantifiers such as $\forall(A, B)$ inside, because all the available rules have the quantifiers implicitly 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 functional 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 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:

$$|\text{Opt}^{\text{Bool}}| = |\mathbb{1} + \text{Bool}| = 1 + |\text{Bool}| = 3 .$$

In the type notation, `Boolean` is denoted by the symbol 2, and the type `Option[Boolean]` by $\mathbb{1} + 2$. So, the type notation $\mathbb{1} + 2$ is consistent with the cardinality 3 of that type,

$$|\mathbb{1} + \text{Bool}| = |\mathbb{1} + 2| = 1 + 2 = 3 .$$

The function type `Option[Boolean] => Boolean` is denoted by $\mathbb{1} + 2 \Rightarrow 2$. Its cardinality is computed as the arithmetic power

$$|\text{Opt}^{\text{Bool}} \Rightarrow \text{Bool}| = |\mathbb{1} + 2 \Rightarrow 2| = |2|^{\mathbb{1}+2} = 2^3 = 8 .$$

Finally, the we write `P` in the type notation as

$$P = \mathbb{1} + (\mathbb{1} + 2 \Rightarrow 2)$$

and find

$$|P| = |\mathbb{1} + (\mathbb{1} + 2 \Rightarrow 2)| = 1 + |\mathbb{1} + 2 \Rightarrow 2| = 1 + 8 = 9 .$$

Example 5.4.1.2 Implement a Scala type `P[A]` for the type notation

$$P^A \triangleq \mathbb{1} + A + \text{Int} \times A + (\text{String} \Rightarrow A) .$$

Solution To translate type notation into Scala code, begin by defining the disjunctive types as case classes (with names chosen for convenience). In this case, P^A is a disjunctive type with four parts, so we will 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 $P = (\text{Either}[A, B], \text{Either}[C, D])$.

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:

$$P \triangleq (A + B) \times (C + D) .$$

We can use the usual rules of arithmetic to expand brackets in this type expression and to obtain an equivalent type:

$$P \cong A \times C + A \times D + B \times C + B \times D .$$

This type is disjunctive, with 4 parts.

Example 5.4.1.4 Show that the following type equivalences do *not* hold: $A + A \not\cong A$ and $A \times A \not\cong A$, 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 $a^A + 0$ or $0 + a^A$. So, a value of type $A + A$ contains a value of type A with an additional information about whether it is the first or the second part of the disjunctive type. We cannot represent all that information in a single value of type A .

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

$$\begin{aligned} \alpha \vee \alpha &\Rightarrow \alpha , & \alpha &\Rightarrow \alpha \vee \alpha , \\ \alpha \wedge \alpha &\Rightarrow \alpha , & \alpha &\Rightarrow \alpha \wedge \alpha , \end{aligned}$$

using the proof rules of Section 5.2.3. Alternatively, we may use the CH correspondence and show that the type signatures

$$\begin{aligned} \forall A. A + A &\Rightarrow A , & \forall A. A &\Rightarrow A + A , \\ \forall A. A \times A &\Rightarrow A , & \forall A. A &\Rightarrow A \times A \end{aligned}$$

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

$$f_1 \triangleq \begin{array}{|c|c|} \hline & A \\ \hline A & a \Rightarrow a \\ \hline A & a \Rightarrow a \\ \hline \end{array} = \begin{array}{|c|c|} \hline & A \\ \hline A & \text{id} \\ \hline A & \text{id} \\ \hline \end{array},$$

$$f_2 \triangleq a:A \Rightarrow a + 0:A = \begin{array}{|c|c|c|} \hline & A & A \\ \hline A & a \Rightarrow a & 0 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline & A & A \\ \hline A & \text{id} & 0 \\ \hline \end{array}.$$

The composition of these functions is not equal to identity:

$$f_1 ; f_2 = \begin{array}{|c|c|} \hline \text{id} \\ \hline \text{id} \\ \hline \end{array} ; \begin{array}{|c|c|} \hline \text{id} & 0 \\ \hline \text{id} & 0 \\ \hline \end{array} = \begin{array}{|c|c|} \hline \text{id} & 0 \\ \hline \text{id} & 0 \\ \hline \end{array} \neq \text{id} = \begin{array}{|c|c|} \hline \text{id} & 0 \\ \hline 0 & \text{id} \\ \hline \end{array}.$$

For the second pair of type signatures, the code is

```
def f1[A]: ((A, A)) => A = { case (a1, a2) => a1 } // Can be also 'a2'.
def f2[A]: A => (A, A) = { x => (x, x) } // No other choice here.
```

It is clear that the first function loses information when it returns a_1 and discards a_2 (or vice versa).

The code notation for these functions is

$$f_1 \triangleq a_1^A \times a_2^A \Rightarrow a_1, \\ f_2 \triangleq a^A \Rightarrow a \times a.$$

The composition of these functions is not equal to identity:

$$f_1 ; f_2 = (a_1 \times a_2 \Rightarrow a_1) ; (a \Rightarrow a \times a) \\ = (a_1 \times a_2 \Rightarrow a_1 \times a_1) \neq \text{id} = (a_1 \times a_2 \Rightarrow a_1 \times a_2).$$

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, although the equality holds in Boolean logic. This is another example of the failure of Boolean logic to provide correct reasoning for types.

Solution Begin by rewriting the logical equality as two implications,

$$(A \wedge B \Rightarrow C) \Rightarrow (A \Rightarrow C) \vee (B \Rightarrow C), \\ ((A \Rightarrow C) \vee (B \Rightarrow C)) \Rightarrow ((A \wedge B) \Rightarrow C).$$

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 would need to demonstrate that *no* proof tree exists), let us use the CH correspondence. So the task is to implement fully parametric functions with the type signatures

$$(A \times B \Rightarrow C) \Rightarrow (A \Rightarrow C) + (B \Rightarrow C), \\ (A \Rightarrow C) + (B \Rightarrow C) \Rightarrow A \times B \Rightarrow C.$$

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 $g: A \Rightarrow C$, or a `Right(h)` with $h: B \Rightarrow C$. The only given data is a function k of type $A \times B \Rightarrow C$, 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 $A \Rightarrow C$? 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 $g: A \Rightarrow C$. Similarly, we cannot produce a function h of type $B \Rightarrow C$.

To repeat the same argument in the type notation: Obtaining a value of type $(A \Rightarrow C) + (B \Rightarrow C)$ means to compute either $g^{A \Rightarrow C} + \emptyset$ or $\emptyset + h^{B \Rightarrow C}$. This decision must be hard-coded since the only data is a function $k^{A \times B \Rightarrow C}$. We can compute a $g^{A \Rightarrow C}$ only by partially applying $k^{A \times B \Rightarrow C}$ to a value of type B . However, we cannot obtain any values of type B . Similarly, we cannot get an $h^{B \Rightarrow C}$.

The inverse type signature is implementable:

```
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) }
}
```

$$f2 \triangleq \begin{array}{|c|c|} \hline & A \times B \Rightarrow C \\ \hline A \Rightarrow C & g^{A \Rightarrow C} \Rightarrow a \times b \Rightarrow g(a) \\ \hline B \Rightarrow C & h^{B \Rightarrow C} \Rightarrow a \times b \Rightarrow h(b) \\ \hline \end{array} .$$

Let us now show that the identity

$$((\alpha \wedge \beta) \Rightarrow \gamma) = ((\alpha \Rightarrow \gamma) \vee (\beta \Rightarrow \gamma)) \quad (5.23)$$

holds in Boolean logic. A straightforward calculation is to simplify the Boolean expression using Eq. (5.5), which only holds in Boolean logic (but not in the constructive logic). We find

$$\begin{aligned} \text{left-hand side of Eq. (5.23)} : & (\alpha \wedge \beta) \Rightarrow \gamma \\ \text{use Eq. (5.5)} : & = \neg(\alpha \wedge \beta) \vee \gamma \\ \text{use de Morgan's law} : & = \neg\alpha \vee \neg\beta \vee \gamma . \\ \text{right-hand side of Eq. (5.23)} : & (\alpha \Rightarrow \gamma) \vee (\beta \Rightarrow \gamma) \\ \text{use Eq. (5.5)} : & = \neg\alpha \vee \gamma \vee \neg\beta \vee \gamma \\ \text{use identity } \gamma \vee \gamma = \gamma : & = \neg\alpha \vee \neg\beta \vee \gamma . \end{aligned}$$

Both sides of Eq. (5.23) are equal to the same formula, $\neg\alpha \vee \neg\beta \vee \gamma$, so the identity holds.

This proof does not work in the constructive logic because neither the Boolean formula (5.5) nor the law of de Morgan,

$$\neg(\alpha \wedge \beta) = (\neg\alpha \vee \neg\beta) ,$$

can be derived via the proof rules of the constructive logic.

Another way of proving the Boolean identity (5.23) 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 $\alpha \wedge \beta = \text{True}$ (that is, both α and β are *True*) and $\gamma = \text{False}$; 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 $\alpha = \text{True}$, $\beta = \text{True}$, and $\gamma = \text{False}$. So, the two sides of the identity (5.23) 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 (5.23).

It is important to note that the proof rules of the constructive logic are not equivalent to checking whether some propositions are *True* or *False*. A general form of this statement was proved by K. Gödel in 1932.⁶ 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.

The following example shows how to use the identities from Tables 5.5–5.6 to derive type equivalence for complicated type expressions, without need for proofs.

⁶See plato.stanford.edu/entries/intuitionistic-logic-development/

Example 5.4.1.6 Use known rules to verify the type equivalences:

(a) $A \times (A + 1) \times (A + 1 + 1) \cong A \times (1 + 1 + A \times (1 + 1 + 1 + A))$.

(b) $(1 + A + B) \Rightarrow 1 \times B \cong (B \Rightarrow B) \times (A \Rightarrow B) \times B$.

Solution (a) We can expand brackets in the type expression as in arithmetic,

$$\begin{aligned} A \times (A + 1) &\cong A \times A + A \times 1 \cong A \times A + A , \\ A \times (A + 1) \times (A + 1 + 1) &\cong (A \times A + A) \times (A + 1 + 1) \\ &\cong A \times A \times A + A \times A + A \times A \times (1 + 1) + A \times (1 + 1) \\ &\cong A \times A \times A + A \times A \times (1 + 1 + 1) + A \times (1 + 1) . \end{aligned}$$

The result looks like a polynomial in A , which we can now rearrange into the required form:

$$A \times A \times A + A \times A \times (1 + 1 + 1) + A \times (1 + 1) \cong A \times (1 + 1 + A \times (1 + 1 + 1 + A)) .$$

(b) Keep in mind that the conventions of the type notation make the function arrow (\Rightarrow) group weaker than other type operations. So, the type expression $(1 + A + B) \Rightarrow 1 \times B$ means a function from $1 + A + B$ to $1 \times B$.

Begin by using the rule $1 \times B \cong B$ to obtain $(1 + A + B) \Rightarrow B$. Now we use the rule

$$A + B \Rightarrow C \cong (A \Rightarrow C) \times (B \Rightarrow C)$$

and derive the equivalence

$$(1 + A + B) \Rightarrow B \cong (1 \Rightarrow B) \times (A \Rightarrow B) \times (B \Rightarrow B) .$$

Finally, we note that $1 \Rightarrow B \cong B$ and that the type product is commutative, so we can rearrange the last type expression into the required form:

$$B \times (A \Rightarrow B) \times (B \Rightarrow B) \cong (B \Rightarrow B) \times (A \Rightarrow B) \times B .$$

Example 5.4.1.7 Denote $\text{Read}^{E,T} \triangleq E \Rightarrow T$ and implement fully parametric functions with types $A \Rightarrow \text{Read}^{E,A}$ and $\text{Read}^{E,A} \Rightarrow (A \Rightarrow B) \Rightarrow \text{Read}^{E,B}$.

Solution Begin by defining a type alias for the type $\text{Read}^{E,T}$:

```
type Read[E, T] = E => T
```

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 $E \Rightarrow A$ and $A \Rightarrow B$. The forward composition of these functions is a function of type $E \Rightarrow B$, 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 $r^{E \Rightarrow A}$, $f^{A \Rightarrow B}$, and e^E . Write this requirement as

$$\text{map} \triangleq r^{E \Rightarrow A} \Rightarrow f^{A \Rightarrow B} \Rightarrow e^E \Rightarrow ???^B ,$$

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.⁷ The plugin will print the known information about the typed hole.

⁷<https://github.com/cb372/scala-typed-holes>

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 $f^{A \Rightarrow B}$ to some value of type A . So we write

$$\text{map} \triangleq r^{E \Rightarrow A} \Rightarrow f^{A \Rightarrow B} \Rightarrow e^E \Rightarrow f(\text{???}^A) \quad .$$

The only way of getting an A is to apply r to a value of type E ,

$$\text{map} \triangleq r^{E \Rightarrow A} \Rightarrow f^{A \Rightarrow B} \Rightarrow e^E \Rightarrow f(r(\text{???}^E)) \quad .$$

We have exactly one value of type E , namely e^E . So the code must be

$$\text{map}^{E, A, B} \triangleq r^{E \Rightarrow A} \Rightarrow f^{A \Rightarrow B} \Rightarrow e^E \Rightarrow f(r(e)) \quad .$$

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 $e \Rightarrow f(r(e))$ is a function composition $r \circ f$ applied to e , and simplify the code accordingly.

Example 5.4.1.8 Show that the type signature $\text{Read}[A, T] \Rightarrow (A \Rightarrow B) \Rightarrow \text{Read}[B, T]$ cannot be implemented as a fully parametric function.

Solution Expand the type signature and try to implement this function:

```
def m[A, B, T] : (A => T) => (A => B) => B => T = { r => f => b => ??? }
```

Given values $r^{A \Rightarrow T}$, $f^{A \Rightarrow B}$, and b^B , we need to compute a value of type T :

$$m = r^{A \Rightarrow T} \Rightarrow f^{A \Rightarrow B} \Rightarrow b^B \Rightarrow \text{???}^T \quad .$$

The only way of getting a T is to apply r to some value of type A ,

$$m = r^{A \Rightarrow T} \Rightarrow f^{A \Rightarrow B} \Rightarrow b^B \Rightarrow r(\text{???}^A) \quad .$$

However, we do not have any values of type A . We have a function $f^{A \Rightarrow B}$ that *consumes* values of type A , and we cannot use f to produce any values of type A . So we seem to be 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

$$\forall(\alpha, \beta, \tau). (\alpha \Rightarrow \tau) \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \tau) \tag{5.24}$$

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 5.2.3 preserve Boolean truth values (see Section 5.5.4 for more details). 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. (5.24) with the rules of Boolean logic, we find

$$\begin{aligned} & (\alpha \Rightarrow \tau) \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \tau) \\ \text{use Eq. (5.5)} : & = \neg(\alpha \Rightarrow \tau) \vee \neg(\alpha \Rightarrow \beta) \vee (\beta \Rightarrow \tau) \\ \text{use Eq. (5.5)} : & = \neg(\neg\alpha \vee \tau) \vee \neg(\neg\alpha \vee \beta) \vee (\neg\beta \vee \tau) \\ \text{use de Morgan's law} : & = (\alpha \wedge \neg\tau) \vee (\alpha \wedge \neg\beta) \vee \neg\beta \vee \tau \\ \text{use identity } (p \wedge q) \vee q = q : & = (\alpha \wedge \neg\tau) \vee \neg\beta \vee \tau \\ \text{use identity } (p \wedge \neg q) \vee q = p \vee q : & = \alpha \vee \neg\beta \vee \tau \end{aligned}$$

This formula is not identically *True*: it is *False* when $\alpha = \tau = \text{False}$ and $\beta = \text{True}$. So, Eq. (5.24) is not true in Boolean logic, and thus is 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 $P^A \triangleq \mathbb{1} + A + A$ and implement map for it,

$$\text{map}^{A,B} : P^A \Rightarrow (A \Rightarrow B) \Rightarrow P^B \quad .$$

To check that map preserves information, verify the law $\text{map}(p)(x \Rightarrow x) == p$ for all $p : P[A]$.

Solution It is implied that map should be fully parametric and information-preserving. Begin by defining a Scala type for the notation $\mathbb{1} + A + A$:

```
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 implement the required type signature. Each time we find a choice in the 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 here.
  case P2(x) => ???           // Preserve information.
  case P3(x) => ???           // Preserve information.
}
```

In the case $P2(x)$, we are required to produce a value of type P^B from a value $x:A$ and a function $f:A \Rightarrow B$. Since P^B is a disjunctive type with three parts, we can produce a value of type P^B in three different ways: $P1()$, $P2(\dots)$, and $P3(\dots)$. If we return $P1()$, we will lose the information about the value x . If we return $P3(\dots)$, we will preserve the information about x but lose the information that the input value was a $P2$ rather than a $P3$. So we should return $P2(\dots)$ in that scope; in that way, we preserve the entire input information.

The value under $P2(\dots)$ 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 :

$$\text{map}^{A,B} \triangleq p : \mathbb{1} + A + A \Rightarrow f : A \Rightarrow B \Rightarrow p \triangleright \left| \begin{array}{c|ccc} & \mathbb{1} & B & B \\ \hline \mathbb{1} & \text{id} & 0 & 0 \\ A & 0 & f & 0 \\ A & 0 & 0 & f \end{array} \right| \quad .$$

The required law is written as an equation

$$\text{map}(p)(\text{id}) = p \quad .$$

Substituting the code notation for `map`, we verify the law:

$$\begin{array}{l}
 \text{expect to equal } p : \text{ map}(p)(\text{id}) \\
 \text{apply map}() \text{ to arguments : } = p \triangleright \left| \begin{array}{ccc} \text{id} & 0 & 0 \\ 0 & \text{id} & 0 \\ 0 & 0 & \text{id} \end{array} \right| \\
 \text{identity in matrix notation : } = p \triangleright \text{id} \\
 \text{---notation : } = \text{id}(p) = p .
 \end{array}$$

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

$$\begin{array}{l}
 \text{map} : P^A \Rightarrow (A \Rightarrow B) \Rightarrow P^B , \\
 \text{flatMap} : P^A \Rightarrow (A \Rightarrow P^B) \Rightarrow P^B .
 \end{array}$$

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 parameter. For example, when applied to the type parameter A , the type signatures are

$$\begin{array}{l}
 \text{map} : P^{A,S,T} \Rightarrow (A \Rightarrow B) \Rightarrow P^{B,S,T} , \\
 \text{flatMap} : P^{A,S,T} \Rightarrow (A \Rightarrow P^{B,S,T}) \Rightarrow P^{B,S,T} .
 \end{array}$$

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]` (i.e. $L + R$), we keep the type parameter R fixed while L is replaced by M . So we obtain the type signatures

$$\begin{array}{l}
 \text{map} : L + R \Rightarrow (L \Rightarrow M) \Rightarrow M + R , \\
 \text{flatMap} : L + R \Rightarrow (L \Rightarrow M + R) \Rightarrow M + R .
 \end{array}$$

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

$$\begin{array}{l}
 \text{map} \triangleq e^{L+R} \Rightarrow f^{L \Rightarrow M} \Rightarrow e \triangleright \left| \begin{array}{c|cc} & M & R \\ \hline L & f & 0 \\ R & 0 & \text{id} \end{array} \right| , \\
 \text{flatMap} \triangleq e^{L+R} \Rightarrow f^{L \Rightarrow M+R} \Rightarrow e \triangleright \left| \begin{array}{c|c} & M + R \\ \hline L & f \\ R & y^{:R} \Rightarrow 0^{:M} + y \end{array} \right| .
 \end{array}$$

Note that we cannot split f into the M and R columns since $f(x^{:L})$ could return either part of the disjunction $M + R$.

Example 5.4.1.11* Define a type $\text{State}^{S,A} \equiv S \Rightarrow A \times S$ and implement the functions:

- (a) $\text{pure}^{S,A} : A \Rightarrow \text{State}^{S,A}$.
- (b) $\text{map}^{S,A,B} : \text{State}^{S,A} \Rightarrow (A \Rightarrow B) \Rightarrow \text{State}^{S,B}$.
- (c) $\text{flatMap}^{S,A,B} : \text{State}^{S,A} \Rightarrow (A \Rightarrow \text{State}^{S,B}) \Rightarrow \text{State}^{S,B}$.

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 $A \Rightarrow S \Rightarrow A \times S$, 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

$$\text{pu}^{S,A} \triangleq a:A \Rightarrow s:S \Rightarrow a \times s .$$

- (b) The explicit type signature is

$$\text{map}^{S,A,B} : (S \Rightarrow A \times S) \Rightarrow (A \Rightarrow B) \Rightarrow S \Rightarrow B \times S .$$

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 $t:S \Rightarrow A \times S$, $f:A \Rightarrow B$, and $s:S$. We begin writing the code of `map` as a typed hole,

$$\text{map} \triangleq t:S \Rightarrow A \times S \Rightarrow f:A \Rightarrow B \Rightarrow s:S \Rightarrow ???^B \times ???^S .$$

The only way of getting a value of type B is by applying f to a value of type A :

$$\text{map} \triangleq t:S \Rightarrow A \times S \Rightarrow f:A \Rightarrow B \Rightarrow s:S \Rightarrow f(???^A) \times ???^S .$$

To fill the typed hole, we need a value of type A . The only possibility of obtaining a value of type A is by applying 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:

$$\text{map} \triangleq t:S \Rightarrow A \times S \Rightarrow f:A \Rightarrow B \Rightarrow s:S \Rightarrow (a:A \times s_2:S \Rightarrow f(a) \times s_2)(t(s)) .$$

If we temporarily denote by q the destructuring function

$$q \triangleq (a:A \times s_2:S \Rightarrow f(a) \times s_2) ,$$

we will notice that the expression $s \Rightarrow q(t(s))$ is a function composition applied to s . So, we rewrite $s \Rightarrow q(t(s))$ as the composition $t \circ q$ and obtain shorter code,

$$\text{map} \triangleq t:S \Rightarrow A \times S \Rightarrow f:A \Rightarrow B \Rightarrow t \circ (a:A \times s:S \Rightarrow f(a) \times s) .$$

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

$$\text{flatMap}^{S,A,B} : (S \Rightarrow A \times S) \Rightarrow (A \Rightarrow S \Rightarrow B \times S) \Rightarrow S \Rightarrow B \times S \quad .$$

We perform t reasoning with typed holes:

$$\text{flatMap} \triangleq t^{S \Rightarrow A \times S} \Rightarrow f^{A \Rightarrow S \Rightarrow B \times S} \Rightarrow s^S \Rightarrow ???^B \times ???^S \quad .$$

To fill $???^B$, we need to apply f to some arguments, since f is the only function that returns any values of type B . A saturated application of f will yield a value of type $B \times S$, which we can return without change:

$$\text{flatMap} \triangleq t^{S \Rightarrow A \times S} \Rightarrow f^{A \Rightarrow S \Rightarrow B \times S} \Rightarrow s^S \Rightarrow f(???^A)(???^S) \quad .$$

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:

$$\text{flatMap} \triangleq t^{S \Rightarrow A \times S} \Rightarrow f^{A \Rightarrow S \Rightarrow B \times S} \Rightarrow s^S \Rightarrow (a \times s_2 \Rightarrow f(a)(s_2))(t(s)) \quad .$$

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.
}
```

As before, in order to preserve information, we choose not to discard the computed value s_2 .

The code notation for `flatMap` can be simplified to

$$\text{flatMap} \triangleq t^{S \Rightarrow A \times S} \Rightarrow f^{A \Rightarrow S \Rightarrow B \times S} \Rightarrow t \circ (a \times s \Rightarrow f(a)(s)) \quad .$$

5.4.2 Exercises

Exercise 5.4.2.1 Find the cardinality of the Scala type `Option[Boolean => Option[Boolean]]`. Show that this type is equivalent to `Option[Boolean] => Boolean`, and argue that the equivalence is accidental and not “natural”.

Exercise 5.4.2.2 Verify the type equivalences $A + A \cong 2 \times A$ and $A \times A \cong 2 \Rightarrow A$, where 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 logic.

Exercise 5.4.2.4 Use known rules to verify the type equivalences:

- (a) $(A + B) \times (A \Rightarrow B) \cong A \times (A \Rightarrow B) + (\mathbb{1} + A \Rightarrow B)$.
- (b) $(A \times (\mathbb{1} + A) \Rightarrow B) \cong (A \Rightarrow B) \times (A \Rightarrow A \Rightarrow B)$.
- (c) $A \Rightarrow (\mathbb{1} + B) \Rightarrow C \times D \cong (A \Rightarrow C) \times (A \Rightarrow D) \times (A \Rightarrow B \Rightarrow C) \times (A \Rightarrow B \Rightarrow D)$.

Exercise 5.4.2.5 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 5.4.2.6 Define a type $\text{OptE}^{T,A} \triangleq \mathbb{1} + T + A$ 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

$$\begin{aligned} \text{map}^{A,B,T} : \text{OptE}^{T,A} \Rightarrow (A \Rightarrow B) \Rightarrow \text{OptE}^{T,B} \quad , \\ \text{flatMap}^{A,B,T} : \text{OptE}^{T,A} \Rightarrow (A \Rightarrow \text{OptE}^{T,B}) \Rightarrow \text{OptE}^{T,B} \quad . \end{aligned}$$

Exercise 5.4.2.7 Implement the `map` function for $P[A]$ (see Example 5.4.1.2). The required type signature is $P^A \Rightarrow (A \Rightarrow B) \Rightarrow P^B$.

Exercise 5.4.2.8 For the type constructor $Q^{T,A}$ defined in Exercise 5.1.4.1, define the `map` function with the type signature

$$\text{map}^{T,A,B} : Q^{T,A} \Rightarrow (A \Rightarrow B) \Rightarrow Q^{T,B} .$$

The implementation should preserve information as much as possible.

Exercise 5.4.2.9 Define a recursive type constructor Tr_3 as $\text{Tr}_3^A \triangleq \mathbb{1} + A \times A \times A \times \text{Tr}_3^A$ and implement the `map` function for it, with the standard type signature

$$\text{map}^{A,B} : \text{Tr}_3^A \Rightarrow (A \Rightarrow B) \Rightarrow \text{Tr}_3^B .$$

Exercise 5.4.2.10 Implement fully parametric functions with the following types:

- (a) $A + Z \Rightarrow (A \Rightarrow B) \Rightarrow B + Z$.
- (b) $A + Z \Rightarrow B + Z \Rightarrow (A \Rightarrow B \Rightarrow C) \Rightarrow C + Z$.
- (c) $\text{flatMap}^{E,A,B} : \text{Read}^{E,A} \Rightarrow (A \Rightarrow \text{Read}^{E,B}) \Rightarrow \text{Read}^{E,B}$.
- (d) $\text{State}^{S,A} \Rightarrow (S \times A \Rightarrow S \times B) \Rightarrow \text{State}^{S,B}$.

Exercise 5.4.2.11* Denote $\text{Cont}^{R,T} \triangleq (T \Rightarrow R) \Rightarrow R$ and implement the functions:

- (a) $\text{map}^{R,T,U} : \text{Cont}^{R,T} \Rightarrow (T \Rightarrow U) \Rightarrow \text{Cont}^{R,U}$.
- (b) $\text{flatMap}^{R,T,U} : \text{Cont}^{R,T} \Rightarrow (T \Rightarrow \text{Cont}^{R,U}) \Rightarrow \text{Cont}^{R,U}$.

Exercise 5.4.2.12* Denote $\text{Select}^{Z,T} \triangleq (T \Rightarrow Z) \Rightarrow T$ and implement the functions:

- (a) $\text{map}^{Z,A,B} : \text{Select}^{Z,A} \Rightarrow (A \Rightarrow B) \Rightarrow \text{Select}^{Z,B}$.
- (b) $\text{flatMap}^{Z,A,B} : \text{Select}^{Z,A} \Rightarrow (A \Rightarrow \text{Select}^{Z,B}) \Rightarrow \text{Select}^{Z,B}$.

5.5 Discussion

5.5.1 Using the Curry-Howard correspondence for writing code

This chapter shows how the CH correspondence performs two practically important tasks of type-level reasoning: checking whether a type signature can be implemented as a fully parametric function, and determining whether two types are equivalent. The first task is accomplished by mapping type expressions into formulas in the constructive logic and by applying the proof rules of that logic. The second task is accomplished by mapping type expressions into *arithmetic* formulas and applying the ordinary rules of arithmetic.

Fully parametric functions can be often derived from their type signatures alone. It is useful for a programmer to know that certain type signatures, such as

```
def f[A, B]: A => (A => B) => B
```

have only one possible implementation, while other type signatures, such as

```
def g[A, B]: A => (B => A) => B
def h[A, B]: ((A => B) => A) => A
```

cannot be implemented as fully parametric functions.

Although tools such as the `curryhoward` library could perform the code derivation, in most cases it is more beneficial if a programmer is able to derive an implementation by hand, or to see quickly that an implementation is impossible. Exercises in this chapter build up the required technique. 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.

Throughout this chapter, we required all functions to be fully parametric. The reason is that the CH correspondence becomes informative only with parameterized types and with fully parametric

functions. For concrete types, e.g. `Int`, one can always produce *some* value even with no previous data, so the proposition $\mathcal{CH}(\text{Int})$ is always true within any code.

Consider the function $(x:\text{Int}) \Rightarrow x + 1$. Its type signature, `Int => Int`, is insufficient to specify the code of the function, because there are many different functions with the same type signature, such as $x \Rightarrow x - 1$, $x \Rightarrow x * 2$, etc. So, deriving code from the type signature `Int => Int` is not a meaningful task. Only a fully parametric type signature, such as $A \Rightarrow (A \Rightarrow B) \Rightarrow B$, gives enough information for possibly deriving the function's code. If we permit functions that are not fully parametric, we will not be able to reason about implementability of type signatures or about code derivation.

Information about the implementability of type signatures is given by logical formulas involving \mathcal{CH} -propositions. The validity of \mathcal{CH} -propositions means that we can compute *some* values of given types, but it does not give any information about the properties of those values, such as whether they satisfy any additional laws. This is why type equivalence is not determined by an equivalence of logical formulas.

It is useful for programmers to be able to reason about types and transform type expressions to equivalent simpler types before starting to write code. We have shown that a type equivalence corresponds to *each* standard arithmetic identity such as $(a + b) + c = a + (b + c)$, $(a \times b) \times c = a \times (b \times c)$, $1 \times a = a$, $(a + b) \times c = a \times c + b \times c$, etc. So, we are allowed to transform and simplify types as if they were arithmetic expressions, e.g. to rewrite

$$(A + B) \times C + D \cong D + A \times C + B \times C .$$

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 may be called **polynomial**. Type expressions that also contain function types may be called **exponential-polynomial**.⁸ The set of exponential-polynomial types covers almost all data types and design patterns used in functional programming, and so this book focuses on these types.

There are no type constructions corresponding to subtraction or division, so equations such as

$$(1 - t) \times (1 + t) = 1 - t \times t \quad \text{or} \quad \frac{t + t \times t}{t} = 1 + t$$

do not directly yield any type equivalences. However, consider this well-known formula,

$$\frac{1}{1 - t} = 1 + t + t^2 + t^3 + \dots + t^n + \dots .$$

At first sight, this formula appears to involve subtraction, division, and an infinite series, and thus cannot be directly translated into a type equivalence. However, the formula can be rewritten as

$$\frac{1}{1 - t} \triangleq L(t) = 1 + t + t^2 + t^3 + \dots + t^n \times L(t) , \quad (5.25)$$

which is finite and only contains additions and multiplications. So, Eq. (5.25) can be translated into a type equivalence:

$$L^A \cong 1 + A + A \times A + A \times A \times A + \dots + \underbrace{A \times \dots \times A}_{n} \times L^A . \quad (5.26)$$

This type formula (with $n = 1$) is equivalent to a recursive definition of the `List` type,

$$\text{List}^A \triangleq 1 + A \times \text{List}^A .$$

The type equivalence (5.26) suggests that we may view the recursive type `List` as an “infinite disjunction” describing lists of zero, one, etc. elements.

⁸Polynomial types are often called “algebraic”, but we will use more specific terms “polynomial” and “exponential-polynomial”.

5.5.2 Implications for designing new programming languages

The functional programming paradigm assumes that programmers will use the six standard type constructions (Section 5.1.2) and the eight standard code constructions (Section 5.2.3). 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 consequence of the CH correspondence is that the type system of any 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. Functional programming languages that support all the standard type and code constructions – for instance, OCaml, Haskell, F#, Scala, Swift, etc., – will be mapped into the constructive logic with all standard logical operations available (*True*, *False*, disjunction, conjunction, and implication). Languages such as C, C++, Java, C# 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 theorems will not be provable. (If $\text{CH}(A)$ is true but not provable, a value of type *A* is not directly computable by programs, although it could have been.) Languages such as Python, JavaScript, Ruby, Clojure have no type checking and so are mapped to *inconsistent* logics where any proposition can be derived – even propositions normally considered *False*.

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. Inconsistency of a logic means that the logic may derive *False* from *True*. The CH correspondence will map such derivations to code that appears to compute a certain value although that value is not actually available. In practice, such code *crashes* (although all types match!) because the computed value has a wrong type, is “null”, or is a pointer to an invalid memory location.

None of these errors will happen in a programming language whose logic of types is complete and consistent, provided that types are checked at compile time.

So, the CH correspondence gives a mathematically justified procedure for designing type systems in new programming languages. The procedure has the following steps:

- Choose a complete formal logic that is free of inconsistencies.
- For each logical operation, provide a type construction in the language.
- For each proof rule, provide a code construction in the language.

Mathematicians have studied different logics (e.g. modal logic, temporal logic, or linear logic). Compared with the constructive logic, these other logics have some additional 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 choose a logic and translate it into a minimal programming language where the code is guaranteed *not to crash* as long as types match. This mathematical guarantee (known as **type safety**) is a powerful help for programmers since it automatically prevents a large set of programming errors. So, programmers will benefit if their programming language is designed using the CH correspondence.

Practically useful programming languages will, of course, introduce many more features than the minimal, mathematically necessary constructions derived from the chosen logic. Programmers will still benefit from type safety as long as the program stays within the mathematically consistent subset of the language. For Scala, a “safe” subset is identified by the scalazzi project.⁹

At the present time, it is not fully understood whether it would be better for a programming language to be based not on constructive logic but on, say, linear logic, temporal logic, or some other logic that adds more operations to the constructive logic. Practical experience suggests that at least

⁹<https://github.com/scalaz/scalazzi>

the operations of the 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.

5.5.3 Uses of the void type

Scala's void type (`Nothing`) corresponds to the logical constant *False*. The practical uses of *False* are quite limited. One use case is for a branch of a `match` / `case` expression that does not return a value because it throws an exception. Such branches are considered formally to return a value of type `Nothing`, which can then be mapped to a value of any other type (through the function `absurd[A]: Nothing => A`, see Example 5.3.4.2).

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 be of type `Double`. For this code to type-check, 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 implicitly uses the function `absurd: Nothing => Double` to convert that type to `Double`. In this way, types will match in the definition of the value `x`.

We will not use exceptions in this book. The functional programming paradigm avoids exceptions because their presence significantly complicates reasoning about code.

So far, none of our examples involved the logical **negation** operation. It is defined as

$$\neg A \triangleq A \Rightarrow False ,$$

and its practical use is as limited as that of *False* and the void type. However, logical negation plays an important role in Boolean logic, which we will discuss next.

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, any theorem of constructive logic is also a theorem of Boolean logic. The reason is that all eight rules of constructive logic (Section 5.2.3) are also true in Boolean logic.

To verify that a formula is true in Boolean logic, we only need 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 $\gamma = \text{True}$ under the assumption that $\alpha = \beta = \text{True}$. So, the sequent $\alpha, \beta \vdash \gamma$ is translated into the Boolean formula

$$\alpha, \beta \vdash \gamma = ((\alpha \wedge \beta) \Rightarrow \gamma) = (\neg\alpha \vee \neg\beta \vee \gamma) .$$

Table 5.7 translates all proof rules of Section 5.2.3 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 $(\neg\Gamma \vee \dots)$ 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 $\Gamma = \text{True}$, 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.

$\text{formula "use function": } (\alpha \wedge (\alpha \Rightarrow \beta)) \Rightarrow \beta$ $\text{use Eq. (5.5): } = \neg(\alpha \wedge (\neg\alpha \vee \beta)) \vee \beta$ $\text{de Morgan's laws: } = \neg\alpha \vee (\alpha \wedge \neg\beta) \vee \beta$ $\text{identity } p \vee (\neg p \wedge q) = p \vee q \text{ with } p = \neg\alpha \text{ and } q = \beta: = \neg\alpha \vee \underline{\neg\beta} \vee \beta$ $\text{axiom "use arg": } = \text{True} .$

Constructive logic	Boolean logic
$\overline{\Gamma \vdash C\mathcal{H}(\mathbb{1})}$ (create unit)	$\neg\Gamma \vee \text{True} = \text{True}$
$\overline{\Gamma, \alpha \vdash \alpha}$ (use arg)	$\neg\Gamma \vee \neg\alpha \vee \alpha = \text{True}$
$\overline{\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \Rightarrow \beta}}$ (create function)	$(\neg\Gamma \vee \neg\alpha \vee \beta) = (\neg\Gamma \vee (\alpha \Rightarrow \beta))$
$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \alpha \Rightarrow \beta}{\Gamma \vdash \beta}$ (use function)	$((\neg\Gamma \vee \alpha) \wedge (\neg\Gamma \vee (\alpha \Rightarrow \beta))) \Rightarrow (\neg\Gamma \vee \beta)$
$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta}{\Gamma \vdash \alpha \wedge \beta}$ (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)	$(\neg\Gamma \vee \alpha) \Rightarrow (\neg\Gamma \vee (\alpha \vee \beta))$
$\frac{\Gamma \vdash \alpha \vee \beta \quad \Gamma, \alpha \vdash \gamma \quad \Gamma, \beta \vdash \gamma}{\Gamma \vdash \gamma}$ (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.

formula “use Either” : $((\alpha \vee \beta) \wedge (\alpha \Rightarrow \gamma) \wedge (\beta \Rightarrow \gamma)) \Rightarrow \gamma$

use Eq. (5.5) : $= \neg((\alpha \vee \beta) \wedge (\neg\alpha \vee \gamma) \wedge (\neg\beta \vee \gamma)) \vee \gamma$

de Morgan’s laws : $= (\neg\alpha \wedge \neg\beta) \vee (\underline{\alpha \wedge \neg\gamma}) \vee (\underline{\beta \wedge \neg\gamma}) \vee \gamma$

identity $p \vee (\neg p \wedge q) = p \vee q$: $= (\neg\alpha \wedge \neg\beta) \vee \alpha \vee \beta \vee \gamma$

identity $p \vee (\neg p \wedge q) = p \vee q$: $= \underline{\neg\alpha \vee \alpha} \vee \beta \vee \gamma$

axiom “use arg” : $= \text{True}$.

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,

$$\text{True} = (\dots) \Rightarrow (\dots) \Rightarrow \dots \Rightarrow f(\alpha, \beta, \gamma) .$$

Since $(\text{True} \Rightarrow \alpha) = \alpha$, this chain proves the Boolean formula $f(\alpha, \beta, \gamma)$.

For example, the proof tree shown in Figure 5.1 is translated into

axiom “use arg” : $\text{True} = (\neg((\alpha \Rightarrow \alpha) \Rightarrow \beta) \vee \neg\alpha \vee \alpha)$

rule “create function” : $\Rightarrow \neg((\alpha \Rightarrow \alpha) \Rightarrow \beta) \vee (\alpha \Rightarrow \alpha)$.

axiom “use arg” : $\text{True} = \neg((\alpha \Rightarrow \alpha) \Rightarrow \beta) \vee ((\alpha \Rightarrow \alpha) \Rightarrow \beta)$.

rule “use function” : $\text{True} \Rightarrow (\neg((\alpha \Rightarrow \alpha) \Rightarrow \beta) \vee \beta)$

rule “create function” : $\Rightarrow (((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta)$.

It is easier to check Boolean truth than to find a proof tree in constructive logic (or to establish that no proof tree exists). So, 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. In addition to formulas shown in Table 5.3 (Section 5.2.1), further examples of formulas that are not true in Boolean logic are

$$\begin{aligned} & \forall A. A , \\ & \forall(A, B). A \Rightarrow B , \\ & \forall(A, B). (A \Rightarrow B) \Rightarrow B . \end{aligned}$$

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 “use arg” into the Boolean axiom $\neg\alpha \vee \alpha = \text{True}$. The

formula $\neg\alpha \vee \alpha = \text{True}$ is known as the **law of excluded middle**,¹⁰ and is equivalent to saying that any proposition α is either true or false. It is remarkable that the constructive logic *does not have* the law of excluded middle; it is neither an axiom nor a derived theorem of constructive logic.

To see why, consider what it would mean for $\neg\alpha \vee \alpha = \text{True}$ to hold in the constructive logic. The negation operation, $\neg\alpha$, is defined as the implication $\alpha \Rightarrow \text{False}$. So, the logical formula $\forall\alpha. \neg\alpha \vee \alpha$ corresponds to the type $\forall A. (A \Rightarrow \emptyset) + A$. Can we compute a value of this type in a fully parametric function? We would need to compute either a value of type $A \Rightarrow \emptyset$ 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. Should we decide to return A or $A \Rightarrow \emptyset$? We certainly cannot compute a value of type A from scratch, since A is a type parameter. As we have seen in Example 5.3.4.3, a value of type $A \Rightarrow \emptyset$ exists if the type A is itself \emptyset ; but we do not know that, and a fully parametric function needs to have the same code for all types A . Since there are no values of type \emptyset , and the type parameter A could be, say, `Int`, we cannot compute a value of type $A \Rightarrow \emptyset$.

Example 5.3.4.3 showed that the type $A \Rightarrow \emptyset$ is equivalent to \emptyset if A is not itself void ($A \not\cong \emptyset$), and to \top otherwise. Surely, any type A is either void or not void. So, why exactly is it impossible to implement a value of the type $(A \Rightarrow \emptyset) + A$? We could say that if A is void then $(A \Rightarrow \emptyset) \cong \top$ is not void, and so one of the types in the disjunction $(A \Rightarrow \emptyset) \vee A$ should be non-void (i.e. have values).

However, this reasoning is incorrect. A fully parametric functions' code must work in the same general way for all types; the code cannot decide what to do depending on a specific type. So, it is insufficient to show that a value "should exist"; the real requirement is to compute a value of type $(A \Rightarrow \emptyset) + A$ in fully parametric code that works the same way for all types A . But, as we have seen, this is impossible.

In Boolean logic, it is sufficient to prove that a value "should exist" (or that the non-existence of a value is contradictory in some way). However, any practically useful program needs to "construct" (i.e. compute) actual values and return them. 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 functional programs.

¹⁰https://en.wikipedia.org/wiki/Law_of_excluded_middle

6 Functors, contrafunctors, and profunctors

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 fully realize 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 return a value of type `A` (or a failure) when finished at a later time.

Since the meaning of the “wrappers” `Seq`, `Try`, and `Future` is quite 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 $f:A \rightarrow B$ to the data of type `A` held inside the wrapper, and the 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` function as the requirement for the “wrapping” functionality: A type constructor `Wrap[A]` is a “wrapper” if there exists a function `map` with the type signature

```
def map[A, B]: Wrap[A] => (A => B) => Wrap[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., – implemented by different methods specific to each wrapper type.

6.1.2 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

$$\text{Opt}^A \triangleq \mathbb{1} + A \quad .$$

The type signature of its `map` function is

$$\text{map}^{A,B} : \mathbb{1} + A \Rightarrow (A \Rightarrow B) \Rightarrow \mathbb{1} + B \quad .$$

This function produces a new `Option[B]` value, possibly holding 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 implementation `mapX` loses information since it ignores all input and always returns a constant value `None`. The implementation `mapY` preserves information and is clearly a more useful version of `map`.

How can we formulate this property of `map` in a rigorous way? The trick is to write the expression `map(oa)(f)` and to choose the argument $f: A \Rightarrow B$ to be the identity function $\text{id}: A \Rightarrow A$ (setting `map`'s type parameters to be equal, $A = B$, so that the types match). Using an identity function to transform the data wrapped in a given value of type `Option[A]` should not change that value. To check if this is so, substitute the identity function instead of `f` into `mapY` and get

```
mapY[A, A](oa: Option[A])(identity[A]: A => A): Option[A]
  == oa match {
    case None      => None          // No change.
    case Some(x)   => Some(x)       // No change.
  } == oa
```

The result is always equal to `oa`. We can write that fact as an equation,

$$\forall x:A. \text{map}(x)(\text{id}) = x \quad .$$

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

$$\text{map}^{A,B} \triangleq p:\mathbb{1}+A \Rightarrow f:A \Rightarrow B \Rightarrow p \triangleright \begin{array}{|c||c|c|} \hline & 1 & B \\ \hline 1 & id & 0 \\ \hline A & 0 & f \\ \hline \end{array} \quad .$$

When writing code, it is convenient to use is the `.map` method defined by 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

$$\text{fmap}^{A,B} : (A \Rightarrow B) \Rightarrow \mathbb{1} + A \Rightarrow \mathbb{1} + B \quad .$$

The Scala implementation of `fmap` is shorter than that of `map`:

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  case None      => None
  case Some(x)   => Some(f(x))
}
```

The code notation for this function is

$$\text{fmap}(f:A \Rightarrow B) \triangleq \begin{array}{c|c} & \begin{matrix} 1 & B \end{matrix} \\ \begin{matrix} 1 \\ A \end{matrix} & \begin{array}{c|c} \text{id} & \mathbb{0} \\ \mathbb{0} & f \end{array} \end{array} . \quad (6.1)$$

The identity law also looks simpler if expressed in terms of `fmap`, namely $\text{fmap}(\text{id}) = \text{id}$. Here we omit the type parameters A and B , which must be both equal.

Note that the type signature of `fmap` looks like a transformation from functions of type $A \Rightarrow B$ to functions of type `Option[A] => Option[B]`. This transformation is called **lifting** because it “lifts” a function $f:A \Rightarrow B$ operating on simple values into a function operating on `Option`-wrapped values.

So, the identity law can be formulated as “a lifted identity function is also an identity function”. If we lift an identity function (or a composition of functions that equals an identity), we expect the wrapped data not to change. The identity law expresses this expectation in a mathematical equation.

6.1.3 Motivation for the composition law

The main feature of a “data wrapper” or “data container” 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

$$\text{incr} \triangleq x:\text{Int} \Rightarrow x + 1 .$$

In order to apply this function to the counter `c`, we need to lift it. 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 $x \Rightarrow x + 2$:

```
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 function composition for arbitrary functions $f:A \Rightarrow B$ and $g:B \Rightarrow C$. This is written as

$$\text{fmap}(f:A \Rightarrow B ; g:B \Rightarrow C) = \text{fmap}(f:A \Rightarrow B) ; \text{fmap}(g:B \Rightarrow C) .$$

This equation is called the **composition law**.

Let us formally verify the composition law for the `Option` type. To practice the code derivations, we will perform the calculations both by writing Scala code fragments and by writing expressions in the code notation.

The Scala code for the function `fmap` was given in Section 6.1.2. To evaluate $\text{fmap}(f ; g)$, we apply `fmap(f andThen g)`, where $f: A \Rightarrow B$ and $g: B \Rightarrow C$ are arbitrary functions, to an arbitrary value `oa: Option[A]`. In Scala code, it is more convenient to use the infix 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 $(f \text{ andThen } g)(x) == g(f(x))$, 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 right-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))
}
```

So 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:

$$\begin{aligned} \text{fmap}(f) ; \text{fmap}(g) &= \begin{vmatrix} \text{id} & 0 \\ 0 & f \end{vmatrix} ; \begin{vmatrix} \text{id} & 0 \\ 0 & g \end{vmatrix} \\ \text{matrix composition : } &= \begin{vmatrix} \text{id} ; \text{id} & 0 \\ 0 & f ; g \end{vmatrix} = \begin{vmatrix} \text{id} & 0 \\ 0 & f ; g \end{vmatrix} \\ \text{definition of fmap : } &= \text{fmap}(f ; g) . \end{aligned}$$

These calculations prove that the `map` method of the `Option` type satisfies the composition law. If the composition law did not hold, we would not be able to understand how `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 be incorrect. 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 $f:A\Rightarrow B$, $g:B\Rightarrow C$, and $h:C\Rightarrow D$, we have

$$\text{fmap}(f) ; \text{fmap}(g ; h) = \text{fmap}(f ; g) ; \text{fmap}(h)$$

Proof The left-hand side is rewritten as

$$\begin{aligned} &\text{fmap}(f) ; \underline{\text{fmap}(g ; h)} \\ \text{composition law for } (g ; h) : &= \text{fmap}(f) ; (\text{fmap}(g) ; \text{fmap}(h)) \\ \text{associativity law (4.3)} : &= \underline{(\text{fmap}(f) ; \text{fmap}(g)) ; \text{fmap}(h)} \\ \text{composition law for } (f ; g) : &= \text{fmap}(f ; g) ; \text{fmap}(h) , \end{aligned}$$

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, and also when the name of the type parameter is not needed.

- A fully parametric function `fmap` with type signature

$$\text{fmap}_L : (A \Rightarrow B) \Rightarrow L^A \Rightarrow L^B ,$$

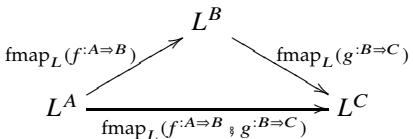
- satisfying the laws

identity law for L : $\text{fmap}_L(\text{id}^{A \Rightarrow A}) = \text{id}^{L^A \Rightarrow L^A} ,$ (6.2)

composition law for L : $\text{fmap}_L(f^{A \Rightarrow B} ; g^{B \Rightarrow C}) = \text{fmap}_L(f^{A \Rightarrow B}) ; \text{fmap}_L(g^{B \Rightarrow C}) .$ (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. The diagram contains two paths from L^A to L^C ; by Eq. (6.3), both paths must give the same result. Mathematicians call such diagrams **commutative**.



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,

$$\text{fmap}_L(g^{B \Rightarrow C} \circ f^{A \Rightarrow B}) = \text{fmap}_L(g^{B \Rightarrow C}) \circ \text{fmap}_L(f^{A \Rightarrow B}) .$$

The function `map` is computationally equivalent to `fmap` and can be defined through `fmap` by

$$\begin{aligned} \text{map}_L : L^A \Rightarrow (A \Rightarrow B) \Rightarrow L^B , \\ \text{map}_L(x^{L^A})(f^{A \Rightarrow B}) = \text{fmap}_L(f^{A \Rightarrow B})(x^{L^A}) . \end{aligned}$$

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 `mapL` and `fmapL`, 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. The most 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 derived classes such as `List`, `Range`, `Vector`, `IndexedSeq`, and `Stream`.
- The “task-like” constructors: `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

```
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 infix 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 an infix 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

$$\text{Counted}^A \triangleq \text{Int} \times A$$

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:

$$\text{fmap}_{\text{Counted}}(f: A \Rightarrow B) \triangleq (n: \text{Int} \times a: A \Rightarrow n \times f(a)) .$$

To verify the identity law, we write

```
expect to equal id : fmap_{Counted}(id)
definition of fmap_{Counted} : = (n \times a \Rightarrow n \times \underline{id(a)})
definition of id : = (n \times a \Rightarrow n \times a) = id .
```

To verify the composition law,

```
expect to equal fmap_{Counted}(f ; g) : fmap_{Counted}(f) ; fmap_{Counted}(g)
definition of fmap_{Counted} : = (n \times a \Rightarrow n \times f(a)) ; (n \times b \Rightarrow n \times g(b))
compute composition : = n \times a \Rightarrow n \times \underline{g(f(a))}
definition of (f ; g) : = (n \times a \Rightarrow n \times (f ; g)(a)) = fmap_{Counted}(f ; g) .
```

6 Functors, contrafunctors, and profunctors

We will prove later that all polynomial type constructors have a definition of `map` that satisfies the functor laws. It will be clear that we defined `map` for `Counted` correctly here.

What could be an *incorrect* implementation of `map`? Suppose that `n` counts 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)
```

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 simple examples of polynomial type constructors.

Example 6.1.4.2 Implement the `fmap` function for the type constructor

```
case class Vec3[A](x: A, y: A, z: A)
```

Solution Begin by implementing a fully parametric function:

```
def fmap[A, B](f: A => B): Vec3[A] => Vec3[B] = {
  case Vec3(x, y, z) => Vec3(f(x), f(y), f(z))
}
```

Since all three values $f(x)$, $f(y)$, $f(z)$ have type B , the code of `fmap` would still satisfy the required type signature by returning, say, `Vec3(f(z), f(x), f(x))` or some other combination of these values. However, that implementation would not preserve information about the values x, y, z and about the ordering of these values in the original data `Vec(x, y, z)`. For this reason, we choose the implementation of `fmap` shown above.

The type notation for the type constructor `Vec3[_]` is

$$\text{Vec}_3^A \triangleq A \times A \times A ,$$

and the code notation for `fmap` is

$$\text{fmap}_{\text{Vec}_3}(f:A \Rightarrow B) \triangleq x:A \times y:A \times z:A \Rightarrow f(x) \times f(y) \times f(z) .$$

Example 6.1.4.3 Implement the `fmap` function for the type constructor

$$\text{QueryResult}^A \triangleq \text{String} + \text{String} \times \text{Long} \times A .$$

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 are recursive functions; they usually *cannot* be directly implemented with tail recursion.

Example Define a list of *odd* length as a recursive type LO^* ,

$$\begin{aligned}\text{LO}^A &\triangleq A + A \times A \times \text{LO}^A \\ &\cong A + A \times A \times A + A \times A \times A \times A + \dots\end{aligned}\tag{6.4}$$

and implement `fmap` for it.

Solution The Scala definition of the type constructor `LO[_]` is

```
sealed trait LO[A]
final case class L01[A](x: A) extends LO[A]
final case class L02[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 L01(x)          => L01[B](f(x))
  case L02(x, y, tail)  => L02[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 `L02`.

The type constructor LO^* is a **recursive polynomial functor** because it is defined by a recursive type equation (6.4) that uses only polynomial type operations (sums and products) in its right-hand side. For the same reason, lists and trees are recursive polynomial 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:

```
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:

- There is less code to read and to write; no `.map` or `case` and fewer curly braces.
- Values such as `name` and `x` need to be kept in tuples and passed from one `.map` function to another, but any line in a functor block can directly reuse all values defined in previous lines.

The functor block is an important idiom 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:LO[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.
  y = f(x)         // Some computation involving 'x'.
  z = g(y)         // Another computation, uses 'y'.
} yield h(z)       // The 'yield' may use 'x', 'y', 'z', and any other defined variables.
```

Translating functor blocks back into a chain of `.map` operations is straightforward except for one complication: if later lines in the functor block make use of variables defined in previous lines, the `.map` operations may need to create some intermediate tuples that are not present in the functor block syntax. Consider the code

```
val result: L[B] = for {
  x <- p           // The first line must contain a left arrow.
  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'.
  .map { case (x, y) => (x, y, g(x, y)) } // Need to keep 'x' and 'y'.
  ...
  .map { case (x, y, z) => q(x, y, z) } // Need to keep 'x', 'y', '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.

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 value 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 an expression of a functor type, i.e. of type `L[A]` for some functor `L[_]`. The result type will be `L[B]` where `B` is the type of the expression after the `yield` keyword.

For instance, the first line of the following functor block contains an `Option` type constructor, so the value of the entire expression will also be wrapped in an `Option` type constructor:

```
scala> for {
  x <- Some(123)
  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]`. (The expression following `yield` can be a large block containing new `vals` and/or other `for` / `yield` functor blocks if needed.)

Functor blocks can be used with any functor that has an infix `.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 `LO` 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(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(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 {
  y <- List(10, 20, 30)
  x = y
  z = f(x)    // 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,

Functor block syntax	Chains of .map methods
<pre>for f // Code fragment 1a. y <- List(10, 20, 30) x = y z = f(x) } yield g(z)</pre>	<pre>List(10, 20, 30) // Code fragment 1b. .map(y => y) .map(x => f(x)) .map(z => g(z))</pre>
<pre>for f // Code fragment 2a. x <- List(10, 20, 30) z = f(x) } yield g(z)</pre>	<pre>List(10, 20, 30) // Code fragment 2b. .map(x => f(x)) .map(z => g(z))</pre>
<pre>for f // Code fragment 3a. x <- List(10, 20, 30) y = x + 1 z = f(y) } yield g(z)</pre>	<pre>List(10, 20, 30) // Code fragment 3b. .map(x => x + 1) .map(y => f(y)) .map(z => g(z))</pre>
<pre>for f // Code fragment 4a. x <- List(10, 20, 30) z = f(x + 1) } yield g(z)</pre>	<pre>List(10, 20, 30) // Code fragment 4b. .map(x => f(x + 1)) .map(z => g(z))</pre>

Table 6.1: Example translations of functor blocks into .map methods.

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 into .map expressions; the resulting code fragments are summarized in Table 6.1.

We find that code fragments 1b and 2b are equal if $\text{.map}(y \Rightarrow y)$ does not modify the list to which it applies. This holds if the .map method obeys the functor identity law, $p.\text{map}(\text{identity}) == p$, for all p of the appropriate type. We also find that code fragments 3b and 4b are equal if we can replace $\text{.map}(x \Rightarrow x + 1).\text{map}(f)$ by $\text{.map}(x \Rightarrow f(x + 1))$. This replacement is justified as long as the .map method obeys the functor composition law,

$$p.\text{map}(h).\text{map}(f) == p.\text{map}(x \Rightarrow f(h(x)))$$

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 C^\bullet defined by

$$C^A \triangleq A \Rightarrow \text{Int} \quad .$$

Scala code for this type notation can be

```
final case class C[A](r: A => Int)
```

The data type `C[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

$$\text{map}^{A,B} : (A \Rightarrow \text{Int}) \Rightarrow (A \Rightarrow B) \Rightarrow (B \Rightarrow \text{Int}) \quad .$$

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]: C[A] => (A => B) => C[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

$$\text{map}^{A,B,N} : (A \Rightarrow N) \Rightarrow (A \Rightarrow B) \Rightarrow B \Rightarrow N \quad .$$

We have seen in Example 5.4.1.8 that this type signature is not implementable. So, the type constructor C is not a functor.

Another important example of type constructors where the `map`'s type signature cannot be implemented are certain kinds of **generalized algebraic data types** (GADTs). In this book, they are called **unfunctors**. An unfunctor is a type constructor that has specially constructed values when its type parameter is set only to certain specific types. An example of an unfunctor defined in Scala is

```
sealed trait ServerAction[R]
final case class GetResult[R](r: String => R) extends ServerAction[R]
final case class StoreId(x: Long, y: String) extends ServerAction[Boolean]
final case class StoreName(name: String) extends ServerAction[Unit]
```

We see that some parts of the disjunctive type `ServerAction[R]` do not carry the type parameter R but instead set R to a specific type, such as $R=\text{Boolean}$ or $R=\text{Unit}$. As a consequence, the case class `StoreName` has no type parameters and can only represent values of type `ServerAction[Unit]` but not, say, `ServerAction[Int]`. To implement a fully parametric function with type signature

```
def map[A, B]: ServerAction[A] => (A => B) => ServerAction[B]
```

we are required to support any choice of the type parameters A and B . For example, with $A=\text{Unit}$, we must be able to transform a `StoreName("abc")` from its fixed type `ServerAction[Unit]` to a value of type `ServerAction[B]` with a given type parameter B . However, the only way of creating a value of type `ServerAction[B]` with an arbitrary type B is to use the `GetResult[B]` case class, which requires us to have a function of type `String => B`. It is impossible for us to produce such a function because the type B is unknown and no values of type B are given.

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 typical, and unfunctors are only used in situations where the lack of type parametricity does not lead to problems (see Chapter 12).

Cannot implement a lawful `map` An example of a non-functor of the second kind is

$$Q^A \triangleq (A \Rightarrow \text{Int}) \times A \quad .$$

Scala code for this type constructor is

```
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):

$$\text{map}^{A,B} \triangleq q^{A \Rightarrow \text{Int}} \times a^A \Rightarrow f^{A \Rightarrow B} \Rightarrow (_ \Rightarrow q(a))^{B \Rightarrow \text{Int}} \times f(a) .$$

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 $q^{A \Rightarrow \text{Int}} \times a^A$ and compute:

$$\begin{aligned} \text{expect to equal } q \times a : & \quad \text{map}(q \times a)(\text{id}) \\ \text{definition of map} : & = (_ \Rightarrow q(a)) \times \underline{\text{id}(a)} \\ \text{definition of id} : & = (_ \Rightarrow q(a)) \times a \\ \text{expanded function, } q = (x \Rightarrow q(x)) : & \quad \neq q \times a = (x \Rightarrow q(x)) \times a . \end{aligned}$$

The law must hold for arbitrary functions $q^{A \Rightarrow \text{Int}}$, but the function $(_ \Rightarrow q(a))$ 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 $P^A \triangleq A \times A$ with the `map` function

$$\text{map} \triangleq x^A \times y^A \Rightarrow f^{A \Rightarrow B} \Rightarrow f(y) \times f(x) .$$

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:

$$\begin{aligned} \text{expect to equal } x \times y : & \quad \text{map}(x^A \times y^A)(\text{id}^A) \\ \text{definition of map} : & = \underline{\text{id}(y)} \times \underline{\text{id}(x)} \\ \text{definition of id} : & = y \times x \neq x \times y . \end{aligned}$$

We should not have swapped the values in the pair. The correct implementation of `map`,

$$\text{map} \triangleq x^A \times y^A \Rightarrow f^{A \Rightarrow B} \Rightarrow f(x) \times f(y) ,$$

preserves information and satisfies the functor laws.

Example 6.1.4.2 shows the type constructor 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 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 (run-time type reflection and `TypeTag`) for comparing two type parameters as types:

```
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 different `map` function that satisfies the functor 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 e.g. the type constructors `Option[_]` and `List[_]` “are” functors, meaning that a suitable lawful implementation of `map` is known.

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 κ . The reason for this failure is complicated. A value of type `Set[K]` represents a set of zero or more values of type κ , and it is enforced that all values in the set are distinct. So, the correct functionality of `Set` implies that we are able to check whether two values of type κ 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, code using the `equals` operation will work as expected only if that operation obeys the laws of **identity** (if $x = y$ then $f(x) = f(y)$ for any f), **reflexivity** ($x = x$ for any x), and **transitivity** (if $x = y$ and $y = z$ then $x = z$). In most practical applications, the required type κ (such as `String` or `Int`) has a mathematically correct `equals` method. In some cases, however, data types will redefine their `equals` method for application-specific purposes and violate some of the required laws.

We will give two non-trivial examples of types that violate the laws of equality. The first example¹ is the type $A + B$ whose `equals` method always returns `false` for any values of type $0 + B$. The code is

¹This example is based on a comment by Paweł Szulc at <https://gist.github.com/tpolecat/7401433>

```
final case class Weird[A, B](x: Either[A, B]) {
  override def equals(y: Any): Boolean = (x, y) match {
    case (Left(a1), Weird(Left(a2))) => a1 == a2
    case _                           => false // Only equal if both 'Left'.
  }
}
```

This implementation of `.equals` is mathematically invalid: it violates the reflexivity law ($\forall x. x = x$) because values of the form `Weird(Right(...))` are never equal to each other:

```
scala> Weird(Right(0)) equals Weird(Right(0))
res2: Boolean = false
```

As a result, the standard code of `Set[Weird]` will fail to detect that e.g. several values `Weird(Right(0))` are equal. The composition law of functors will fail when intermediate values of that type are used:

```
val f: Weird[Int, Int] => Int = { case Weird(Left(a)) => a; case Weird(Right(a)) => a }
val g: Int => Weird[Int, Int] = { a => Weird(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[Weird[Int, Int]] = Set(Weird(Right(0)), Weird(Right(0)), Weird(Right(0)))

scala> xs.map(f).map(g) // 'Set' detects identical values.
res4: Set[Weird[Int, Int]] = Set(Weird(Right(0)))
```

The second example is the pair type $A \times B$ whose `.equals` method ignores the part of type B :

```
final case class Ignore2[A, B](a: A, b: B) {
  override def equals(y: Any): Boolean = y match {
    case Ignore2(p, q) => a == p
    case _              => false
  }
}

scala> Ignore2(123, "abc") == Ignore2(123, "def")
res5: Boolean = true
```

As a result, the code of `Set[Ignore2]` will fail to detect that some values are different. This also violates the functor composition law:

```
val f: Ignore2[Int, Int] => Ignore2[Int, Int] = { case Ignore2(x, y) => Ignore2(y, x) } // f ∘ f = id
val xs = Set(Ignore2(0, 0), Ignore2(1, 0))

scala> xs.map(f andThen f) // This is equal to 'xs'.
res6: Set[Ignore2[Int, Int]] = Set(Ignore2(0,0), Ignore2(1,0))

scala> xs.map(f).map(f) // This is not equal to 'xs'.
res7: Set[Ignore2[Int, Int]] = Set(Ignore2(0,0))
```

The functor laws for a type constructor L^\bullet do not assume that the types A, B used in the function

$$\text{fmap}_L : (A \Rightarrow B) \Rightarrow L^A \Rightarrow L^B$$

should have a mathematically valid definition of the `.equals` method (or of any other operation). The `map` operation 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 implementations of certain operations violate some other laws. For this reason, `Set[_]` cannot be considered a functor in a full sense.

The `map` method for dictionaries is similar: the “keys” of a dictionary must be distinct and will be compared using the `.equals` method. In other words, a dictionary `Map[K, V]` behaves as a set with respect to the type K of the dictionary’s “keys” and thus is not a functor with respect to K .

The Scala library still provides the `.map` and `.flatMap` methods for sets `Set[K]` and dictionaries `Map[K, V]` because in most applications the type K will have a correctly defined `.equals` operation, and so it is likely that we will not see any violations of the functor laws.

6.1.7 Contrafunctors

As we have seen in Section 6.1.6, the type constructor C^\bullet defined by $C^A \triangleq A \Rightarrow \text{Int}$ is not a functor because it is impossible to implement the type signature of `map` for it,

$$\text{map}^{A,B} : (A \Rightarrow \text{Int}) \Rightarrow (A \Rightarrow B) \Rightarrow B \Rightarrow \text{Int} .$$

To see why, begin writing the code with a typed hole,

$$\text{map}(c^{A \Rightarrow \text{Int}})(f^{A \Rightarrow B})(b^B) = ???^{\text{Int}} .$$

Since $c^{A \Rightarrow \text{Int}}$ consumes (rather than wraps) values of type A , we have no values of type A and cannot apply the function $c^{A \Rightarrow \text{Int}}$. However, it would be possible to apply a function of type $B \Rightarrow A$ 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 $B \Rightarrow A$ instead of $A \Rightarrow B$:

$$\text{contramap}^{A,B} : (A \Rightarrow \text{Int}) \Rightarrow (B \Rightarrow A) \Rightarrow B \Rightarrow \text{Int} .$$

The implementation of this function is written in the code notation as

$$\text{contramap} \triangleq c^{A \Rightarrow \text{Int}} \Rightarrow f^{B \Rightarrow A} \Rightarrow (f ; c)^{B \Rightarrow \text{Int}} ,$$

and the corresponding Scala code is

```
def contramap[A, B](ca: C[A])(f: B => A): C[B] = { f andThen ca }
```

Flipping the order of the curried arguments in `contramap`, we define `contrafmap` as

$$\begin{aligned} \text{contrafmap}^{A,B} &: (B \Rightarrow A) \Rightarrow C^A \Rightarrow C^B , \\ \text{contrafmap} &\triangleq f^{B \Rightarrow A} \Rightarrow c^{A \Rightarrow \text{Int}} \Rightarrow (f ; c)^{B \Rightarrow \text{Int}} . \end{aligned} \quad (6.5)$$

The type signature of `contrafmap` has the form of a “reverse lifting”: functions of type $B \Rightarrow A$ are lifted into the type $c[A] \Rightarrow C[B]$. The Scala code for `contrafmap` is

```
def contrafmap[A, B](f: B => A): C[A] => C[B] = { ca => f andThen ca }
```

We can check that this `contrafmap` satisfies two laws analogous to the functor laws:

$$\begin{aligned} \text{identity law : } \text{contrafmap}^{A,A}(\text{id}^{A \Rightarrow A}) &= \text{id}^{C^A \Rightarrow C^A} , \\ \text{composition law : } \text{contrafmap}^{A,B}(f^{B \Rightarrow A}) ; \text{contrafmap}^{B,C}(g^{C \Rightarrow B}) &= \text{contrafmap}(g ; f) . \end{aligned}$$

Since the function argument $f^{B \Rightarrow A}$ has the reverse order of types, the composition law reverses the order of composition ($g ; f$) on one side; in this way, all types match. To verify the identity law:

$$\begin{aligned} \text{expect to equal id : } \text{contrafmap}(\text{id}) & \\ \text{use Eq. (6.5) : } &= c \Rightarrow (\text{id} ; c) \\ \text{definition of id : } &= (c \Rightarrow c) = \text{id} . \end{aligned}$$

To verify the composition law:

$$\begin{aligned} \text{expect to equal contrafmap}(g ; f) : \text{contrafmap}(f) ; \text{contrafmap}(g) & \\ \text{use Eq. (6.5) : } &= (c \Rightarrow (f ; c)) ; (\underline{c} \Rightarrow (g ; \underline{c})) \\ \text{rename } c \text{ to } d \text{ for clarity : } &= (c \Rightarrow (f ; c)) ; (d \Rightarrow (g ; d)) \\ \text{compute composition : } &= (c \Rightarrow g ; f ; c) \\ \text{use Eq. (6.5) : } &= \text{contrafmap}(g ; f) . \end{aligned}$$

A type constructor with a fully parametric `contramap` is called a **contrafunctor** if the identity and the composition laws are satisfied.

Example 6.1.7.1 Show that the type constructor $D^A \triangleq A \Rightarrow A \Rightarrow \text{Int}$ 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:

$$\text{contramap}^{A,B} \triangleq d: A \Rightarrow A \Rightarrow \text{Int} \Rightarrow f: B \Rightarrow A \Rightarrow b_1: B \Rightarrow b_2: B \Rightarrow ???: \text{Int} \quad .$$

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 $f: B \Rightarrow A$ 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

$$\text{contramap}^{A,B} \triangleq d: A \Rightarrow A \Rightarrow \text{Int} \Rightarrow f: B \Rightarrow A \Rightarrow b_1: B \Rightarrow b_2: B \Rightarrow d(f(b_1))(f(b_2)) \quad .$$

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 `contrafmap` defined by

$$\text{contrafmap}^{A,B}(f: B \Rightarrow A) \triangleq d: A \Rightarrow A \Rightarrow \text{Int} \Rightarrow b_1: B \Rightarrow b_2: B \Rightarrow d(f(b_1))(f(b_2)) \quad . \quad (6.6)$$

To verify the identity law:

$$\begin{aligned} &\text{expect to equal id : } \text{contrafmap}(\text{id}) \\ &\text{use Eq. (6.6) : } = d \Rightarrow b_1 \Rightarrow b_2 \Rightarrow d(\underline{\text{id}(b_1)})(\underline{\text{id}(b_2)}) \\ &\text{definition of id : } = d \Rightarrow b_1 \Rightarrow b_2 \Rightarrow d(b_1)(b_2) \\ &\text{simplify curried function : } = (d \Rightarrow d) = \text{id} \quad . \end{aligned}$$

To verify the composition law, we rewrite its left-hand side into the right-hand side:

$$\begin{aligned} &\text{contrafmap}(f) ; \text{contrafmap}(g) \\ &\text{use Eq. (6.6) : } = (d \Rightarrow b_1 \Rightarrow b_2 \Rightarrow d(f(b_1))(f(b_2))) ; (d \Rightarrow b_1 \Rightarrow b_2 \Rightarrow d(g(b_1))(g(b_2))) \\ &\text{rename } d \text{ to } e : = (d \Rightarrow b_1 \Rightarrow b_2 \Rightarrow d(f(b_1))(f(b_2))) ; (e \Rightarrow b_1 \Rightarrow b_2 \Rightarrow e(g(b_1))(g(b_2))) \\ &\text{compute composition : } = d \Rightarrow b_1 \Rightarrow b_2 \Rightarrow d(f(g(b_1)))(f(g(b_2))) \\ &\text{use Eq. (6.6) : } = \text{contrafmap}(b \Rightarrow f(g(b))) \\ &\text{definition of } (g ; f) : = \text{contrafmap}(g ; f) \quad . \end{aligned}$$

The type C^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 $A \Rightarrow A \Rightarrow \text{Int}$, 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

$$N^A \triangleq (A \Rightarrow \text{Int}) \times (\mathbb{1} + A) \quad .$$

We can implement neither `map` nor `contramap` for N^* . 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 violate 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 for showing that `serverAction[_]` cannot be a functor.

6.1.8 Subtyping, covariance, and contravariance

A type P is called a **subtype** of a type Q if there exists a designated **type conversion** function of type $P \Rightarrow Q$ that the compiler will automatically use whenever necessary to match types. For instance, applying a function of type $Q \Rightarrow Z$ to a value of type P is ordinarily a type error,

```
val h: Q => Z = ???
val p: P = ???
h(p) // Type error: the argument of h must be of type Q, not P.
```

However, this code will work when P is a subtype of Q because the compiler will automatically use the type conversion $P \Rightarrow Q$ before applying the function h .

Different programming languages define subtyping differently because they make different choices of the type conversion functions and of types P, Q to which type conversions apply. Most frequently, the language designers choose the type conversion functions to be *identity* functions that merely reassign the types. Let us look at some examples of such type conversion functions.

Within the focus of this book, the main example of subtyping is with disjunctive types. Consider this 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
```

The corresponding type notation can be written as

$$\text{AtMostTwo} \triangleq \mathbb{1} + \text{Int} + \text{Int} \times \text{Int} \quad .$$

Each of the case classes (`Zero`, `One`, and `Two`) defines a type that is a subtype of `AtMostTwo`. To see that, we need to implement type 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 implementation of these type conversion functions looks like the code of *identity* functions. In the matrix notation, we can write

$$\begin{array}{c} f_0 \triangleq \left| \begin{array}{c|ccc} & \text{Zero} & \text{One} & \text{Two} \\ \text{Zero} & \text{id} & 0 & 0 \end{array} \right|, \quad f_0(1^{\text{Zero}}) \triangleq 1 + 0^{\text{One}} + 0^{\text{Two}} , \\ f_1 \triangleq \left| \begin{array}{c|ccc} & \text{Zero} & \text{One} & \text{Two} \\ \text{One} & 0 & \text{id} & 0 \end{array} \right|, \quad f_1(x^{\text{Int}}) \triangleq 0^{\text{Zero}} + x^{\text{One}} + 0^{\text{Two}} , \\ f_2 \triangleq \left| \begin{array}{c|ccc} & \text{Zero} & \text{One} & \text{Two} \\ \text{Two} & 0 & 0 & \text{id} \end{array} \right|, \quad f_2(x^{\text{Int}} \times y^{\text{Int}}) \triangleq 0^{\text{Zero}} + 0^{\text{One}} + (x \times y)^{\text{Two}} . \end{array}$$

This notation emphasizes that the code consists of identity functions with reassigned types.

Another example is a subtyping relation between function types. Consider the types

```
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 the type conversion is

```
def p2q(f: P): Q = { t: Two => f(t) }
```

This is written in the code notation as

$$\text{p2q}(f^{\text{:AtMostTwo} \Rightarrow \text{Int}}) \triangleq t^{\text{:Two}} \Rightarrow f(t) .$$

Note that $t^{\text{:Two}} \Rightarrow f(t)$ is the same function as f , except applied to a subtype `Two` of `AtMostTwo`. So, the implementation of `p2q` is an identity function with reassigned types.

In these cases, it is useful if the compiler could insert the appropriate conversion functions automatically whenever necessary. Any function that consumes an argument of type Q could be then automatically applicable to an argument of type P . The compiler could also remove the identity functions from the code, since they do not perform any data transformations. In this way, code involving subtypes becomes more concise with no decrease in performance.

To achieve this, we need to declare to the Scala compiler that certain types have a subtyping relation involving an identity function as the type conversion. This can be done in one of three ways depending on the situation at hand:

1. Declaring a class that `extends` another class (as we have just seen).
2. Declaring type parameters with a “variance annotation” such as `L[+A]` or `L[-B]`.
3. Declaring type parameters with a “subtyping annotation” (`A <: B`).

Subtyping for disjunctive types 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 used the type 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 type conversion does not change any data and only reassigns the types of the given values. So the compiler does not need to insert any additional code. Type conversion does not lead to any performance cost here.

Subtyping for type constructors If a type constructor L^A is a functor, we can use its fmap_L method to lift a type conversion function $f : P \Rightarrow Q$ into

$$\text{fmap}_L(f) : L^P \Rightarrow L^Q ,$$

which gives a type conversion function from L^P to L^Q . This gives a subtyping relation between the types L^P and L^Q because the code of the lifted function $\text{fmap}_L(f)$ is an identity function, due to functor L 's identity law, $\text{fmap}_L(\text{id}) = \text{id}$.

If a type constructor C^A is a contrafunctor, a type conversion function $f^{\text{:}P \Rightarrow Q}$ is lifted to

$$\text{contrafmap}_C(f) : C^Q \Rightarrow C^P ,$$

showing that C^Q is a subtype of C^P . The identity law of the contrafunctor C ,

$$\text{contrafmap}_C(\text{id}) = \text{id} ,$$

shows that the lifted conversion function is 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 C has the subtype relation in the opposite direction: C^B is a subtype of C^A . So, we have established that all functors are covariant type constructors, and all contrafunctors are contravariant type constructors.²

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 like `F[+A]`, on the relevant type parameters. For

²The name “contrafunctor” was chosen in this book as a shortened form of “contravariant functor”.

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]`. So we may now apply a 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 given type constructor may have several type parameters and may be covariant with respect to some of them and contravariant with respect to others. As we have seen, the position of a type parameter in a type expression indicates whether the value is “wrapped” (used in a **covariant position**) or “consumed” (used in a **contravariant position**). Covariant positions are to the right of function arrows, or outside function arrows; contravariant positions are to the left of a function arrow. The next examples confirm this intuition, which will be made rigorous in Section 6.2.

6.1.9 Solved examples: 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) $\text{Data}^A \triangleq \text{String} + A \times \text{Int} + A \times A \times A$.
- (b) $\text{Data}^A \triangleq \mathbb{1} + A \times (\text{Int} \times \text{String} + A)$.
- (c) $\text{Data}^A \triangleq (\text{String} \Rightarrow \text{Int} \Rightarrow A) \times A + (\text{Bool} \Rightarrow \text{Double} \Rightarrow A) \times A$.

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 the type signature

$$\text{fmap}^{A,B} : f:A \Rightarrow B \Rightarrow \text{Data}^A \Rightarrow \text{Data}^B .$$

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 $f:A \Rightarrow B$

need to be applied, let us write the type notation for Data^A and Data^B side by side:

$$\begin{aligned}\text{Data}^A &\triangleq \text{String} + A \times \text{Int} + A \times A \times A , \\ \text{Data}^B &\triangleq \text{String} + B \times \text{Int} + B \times B \times B .\end{aligned}$$

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 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 $(1 + \dots)$ 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 help us implement `fmap` correctly, we write the type expressions

$$\begin{aligned}\text{Data}^A &\triangleq 1 + A \times (\text{Int} \times \text{String} + A) , \\ \text{Data}^B &\triangleq 1 + B \times (\text{Int} \times \text{String} + B) ,\end{aligned}$$

and transform Data^A into Data^B by applying $f: A \Rightarrow B$ 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 $(\text{String} \Rightarrow \text{Int} \Rightarrow A) \times A$ and $(\text{Bool} \Rightarrow \text{Double} \Rightarrow A) \times A$ have a similar pattern, let us define a parameterized type

$$Q^{X,Y,A} \triangleq (X \Rightarrow Y \Rightarrow A) \times A ,$$

and express the given type expression as

$$\text{Data}^A \triangleq Q^{\text{String}, \text{Int}, A} + Q^{\text{Bool}, \text{Double}, A} .$$

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]]
```

To make the code clearer, we will implement `fmap` separately for Q^\bullet and Data^\bullet .

To derive the code of `fmap` for Q^\bullet , we begin with the type signature

$$\text{fmap}_{Q^\bullet}^{A,B} : (A \Rightarrow B) \Rightarrow (X \Rightarrow Y \Rightarrow A) \times A \Rightarrow (X \Rightarrow Y \Rightarrow B) \times B$$

and start writing the code using typed holes,

$$\text{fmap}_{Q^\bullet}(f^{A \Rightarrow B}) \triangleq g^{X \Rightarrow Y \Rightarrow A} \times a^{A \Rightarrow \dots} \Rightarrow \dots^{X \Rightarrow Y \Rightarrow B} \times \dots^{B \Rightarrow \dots} .$$

The typed hole $\dots^{B \Rightarrow \dots}$ is filled by $f(a)$. To fill the remaining type hole, we write

$$\begin{aligned} & \dots^{X \Rightarrow Y \Rightarrow B} \\ &= x^{X \Rightarrow} \Rightarrow y^{Y \Rightarrow} \Rightarrow \dots^{B \Rightarrow} \\ &= x^{X \Rightarrow} \Rightarrow y^{Y \Rightarrow} \Rightarrow f(\dots^{A \Rightarrow}) . \end{aligned}$$

It would be wrong to fill the typed hole $\dots^{A \Rightarrow}$ by $a^{A \Rightarrow}$ because a value of type $X \Rightarrow Y \Rightarrow B$ should be computed using the given data $g^{X \Rightarrow Y \Rightarrow A}$ of type $X \Rightarrow Y \Rightarrow A$. So we write

$$\dots^{X \Rightarrow Y \Rightarrow B} = x^{X \Rightarrow} \Rightarrow y^{Y \Rightarrow} \Rightarrow f(g(x)(y)) .$$

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 `fmap` 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 would be

```
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 which of these types are functors or contrafunctors, and implement `fmap` or `contrafmap` as appropriate:

$$(a) \text{Data}^A \triangleq (A \Rightarrow \text{Int}) + (A \Rightarrow A \Rightarrow \text{String}) .$$

$$(b) \text{Data}^{A,B} \triangleq (A + B) \times ((A \Rightarrow \text{Int}) \Rightarrow B) .$$

Solution (a) The type constructor Data^A uses its type parameter A always as an argument of some functions, i.e. to the left of function arrows:

```
type Data[A] = Either[A => Int, A => A => String]
```

So, Data^A consumes values of type A , and we expect that Data^A is a contrafunctor. Indeed, we can implement `contrafmap`:

```
def contrafmap[A, B](f: B => A): Data[A] => Data[B] = {
  case Left(a2Int) => Left(b => a2Int(f(b)))
  case Right(a2a2String) => Right(b1 => b2 => a2a2String(f(b1))(f(b2)))
}
```

(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 $(A \Rightarrow \text{Int}) \Rightarrow B$. This function consumes a function of type $A \Rightarrow \text{Int}$, 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 will 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,

$$\text{fmap}^{A,C,Z} : (A \Rightarrow C) \Rightarrow (A + Z) \times ((A \Rightarrow \text{Int}) \Rightarrow Z) \Rightarrow (C + Z) \times ((C \Rightarrow \text{Int}) \Rightarrow Z) .$$

We need to transform each part of the tuple separately. Transforming $A + Z$ into $C + Z$ is straightforward via the function

	C	Z
A	f	id
Z	id	id

The corresponding Scala code is

```
{
  case Left(x)    => Left(f(x))
  case Right(z)   => Right(z)
}
```

To derive code transforming $(A \Rightarrow \text{Int}) \Rightarrow Z$ into $(C \Rightarrow \text{Int}) \Rightarrow Z$, we use typed holes:

$$\begin{aligned}
 f : A \Rightarrow C &\Rightarrow g : (A \Rightarrow \text{Int}) \Rightarrow Z \Rightarrow \underline{\text{???}} : (C \Rightarrow \text{Int}) \Rightarrow Z \\
 \text{nameless function : } &= f : A \Rightarrow C \Rightarrow g : (A \Rightarrow \text{Int}) \Rightarrow Z \Rightarrow p : C \Rightarrow \text{Int} \Rightarrow \underline{\text{???}} : Z \\
 \text{get a } Z \text{ by applying } g : &= f \Rightarrow g \Rightarrow p \Rightarrow g(\underline{\text{???}} : A \Rightarrow \text{Int}) \\
 \text{nameless function : } &= f \Rightarrow g \Rightarrow p \Rightarrow g(a : A \Rightarrow \underline{\text{???}} : \text{Int}) \\
 \text{get an Int by applying } p : &= f \Rightarrow g \Rightarrow p \Rightarrow g(a \Rightarrow p(\underline{\text{???}} : C)) \\
 \text{get a } C \text{ by applying } f : &= f \Rightarrow g \Rightarrow p \Rightarrow g(a \Rightarrow p(f(\underline{\text{???}} : A))) \\
 \text{use argument } a : A : &= f \Rightarrow g \Rightarrow p \Rightarrow g(a \Rightarrow p(f(a))) .
 \end{aligned}$$

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 always used in covariant positions, i.e. never to the left of a function arrow. 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 together all parts of the disjunctive type:

$$\text{Coi}^{A,B} \triangleq A \times B \times (B \Rightarrow \text{Int}) + A \times B \times \text{Int} + (\text{String} \Rightarrow A) \times (B \Rightarrow A)$$

Now find which types are wrapped and which are consumed in this type expression. We find that the type parameter A is wrapped and never consumed, but B is both wrapped and consumed (in $B \Rightarrow A$). 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 Scala 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: 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: Int) => Right(f(a))
  case Left(a)       => Left(f(a))
  case Right(a)      => Right(f(a))
}
```

Show that this implementation of `fmap` violates the functor laws. Implement `fmap` correctly for this type constructor.

Exercise 6.1.10.2 Define these type constructors in Scala, decide whether they are covariant or contravariant, and implement `fmap` or `contrafmap` as appropriate:

- (a) $\text{Data}^A \triangleq (\mathbb{1} + A) \times (\mathbb{1} + A) \times \text{String}$
- (b) $\text{Data}^A \triangleq (A \Rightarrow \text{Bool}) \Rightarrow (A \times (\text{Int} + A))$
- (c) $\text{Data}^{A,B} \triangleq (A \Rightarrow \text{Bool}) \times ((A + B) \Rightarrow \text{Int})$
- (d) $\text{Data}^A \triangleq (\mathbb{1} + (A \Rightarrow \text{Bool})) \Rightarrow (\mathbb{1} + (A \Rightarrow \text{Int})) \Rightarrow \text{Int}$

(e) Data^B $\triangleq (B + (\text{Int} \Rightarrow B)) \times (B + (\text{String} \Rightarrow B))$.

Exercise 6.1.10.3 Rewrite the following code in the type notation; identify covariant and contravariant type usages; add variance annotations and verify that the resulting code compiles:

```
sealed trait Result[A,B]
final case class P[A,B](a: A, b: B, c: Int) extends Result[A,B]
final case class Q[A,B](d: Int  $\Rightarrow$  A, e: Int  $\Rightarrow$  B) extends Result[A,B]
final case class R[A,B](f: A  $\Rightarrow$  A, g: A  $\Rightarrow$  B) extends Result[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 that a given type constructor is a functor or perhaps a contrafunctor? For example, consider the type constructor $Z^{A,R}$ defined by

$$Z^{A,R} \triangleq ((A \Rightarrow A \Rightarrow R) \Rightarrow R) \times A + (\mathbb{1} + R \Rightarrow A + \text{Int}) + A \times A \times \text{Int} \times \text{Int} . \quad (6.7)$$

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 Reformulations of laws

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:

$$\begin{aligned} \text{map}_L(x^{:L^A})(\text{id}^{:A \Rightarrow A}) &= x , \\ \text{map}_L(x^{:L^A})(f^{:A \Rightarrow B} ; g^{:B \Rightarrow C}) &= \text{map}_L(\text{map}_L(x)(f))(g) . \end{aligned}$$

However, the laws again look clearer when using the infix method `.map`:

```
x.map(identity) == x
x.map(f).map(g) == x.map(f andThen g)
```

To take advantage of this syntax, we can use the pipe notation where $x \triangleright \text{fmap}(f)$ means `x.map(f)`, and write the functor laws as

$$\begin{aligned} x \triangleright \text{fmap}_L(\text{id}) &= x , \\ x \triangleright \text{fmap}_L(f) \triangleright \text{fmap}_L(g) &= x \triangleright \text{fmap}_L(f ; g) . \end{aligned}$$

In later chapters of this book, we will find that the `.map` methods (equivalently, the `fmap` function) are 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 mark 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 ”) defined, for any function f , by

$$(f^{:A \Rightarrow B})^{\uparrow L} : L^A \Rightarrow L^B , \quad f^{\uparrow L} \triangleq \text{fmap}_L(f) .$$

Now we can write

$$\begin{aligned} x \triangleright \text{fmap}_L(f) &= x \triangleright f^{\uparrow L} = f^{\uparrow L}(x) , \\ \text{map}_L(x)(f) &= f^{\uparrow L}(x) . \end{aligned}$$

In this notation, the identity and composition laws for a functor L are especially easy to use:

$$\text{id}^{\uparrow L} = \text{id} , \quad (f ; g)^{\uparrow L} = f^{\uparrow L} ; g^{\uparrow L} .$$

Applying a composition of lifted functions to a value looks like this,

$$x \triangleright (f \circ g)^{\uparrow L} = x \triangleright f^{\uparrow L} ; g^{\uparrow L} = x \triangleright f^{\uparrow L} \triangleright g^{\uparrow L} .$$

This equation directly represents the Scala code syntax

```
x.map(f andThen g) == x.map(f).map(g)
```

since the piping symbol (\triangleright) groups weaker than the composition symbol (\circ).

Written in the *backward* notation ($f \circ g$), the functor composition law is

$$(g \circ f)^{\uparrow L} = g^{\uparrow L} \circ f^{\uparrow L} .$$

The analogous notation for a contrafunctor C^\bullet is

$$f^{\downarrow C} \triangleq \text{contrafmap}_C(f) .$$

The contrafunctor laws are then written as

$$\text{id}^{\downarrow C} = \text{id} , \quad (f \circ g)^{\downarrow C} = g^{\downarrow C} ; f^{\downarrow C} , \quad (g \circ f)^{\downarrow C} = f^{\downarrow C} \circ g^{\downarrow C} .$$

We will mostly use the forward composition $f \circ g$ in this book, keeping in mind that one can straightforwardly and mechanically translate between forward and backward notations via

$$f \circ g = g \circ f , \quad x \triangleright f = f(x) .$$

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 type parameters at once.

As an example, consider the type constructor F defined by

$$F^{A,B} \triangleq A \times B \times B .$$

If we fix the type parameter B but let the parameter A vary, we get a type constructor that we can denote as $F^{\bullet,B}$. We see that the type constructor $F^{\bullet,B}$ is a functor, with the corresponding `fmap` function

$$\text{fmap}_{F^{\bullet,B}}(f:A \Rightarrow C) \triangleq a:A \times b_1^B \times b_2^B \Rightarrow f(a) \times b_1 \times b_2 .$$

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 .

If we now fix the type parameter A , we find that the type constructor $F^{A,\bullet}$ is a functor, with the `fmap` function

$$\text{fmap}_{F^{A,\bullet}}(g:B \Rightarrow D) \triangleq a:A \times b_1^B \times b_2^B \Rightarrow a \times g(b_1) \times g(b_2) .$$

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 $f:A \Rightarrow C$ and $g:B \Rightarrow D$ as arguments:

$$\begin{aligned} \text{bimap}_F(f:A \Rightarrow C)(g:B \Rightarrow D) &: F^{A,B} \Rightarrow F^{C,D} , \\ \text{bimap}_F(f:A \Rightarrow C)(g:B \Rightarrow D) &\triangleq \text{fmap}_{F^{\bullet,B}}(f:A \Rightarrow C) ; \text{fmap}_{F^{A,\bullet}}(g:B \Rightarrow D) . \end{aligned} \tag{6.8}$$

In the condensed notation, this is written as

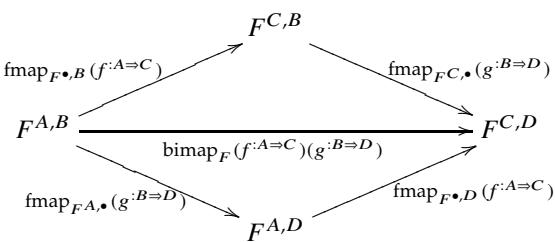
$$\text{bimap}_F(f:A \Rightarrow C)(g:B \Rightarrow D) \triangleq f^{\uparrow F^{\bullet,B}} ; g^{\uparrow F^{A,\bullet}} ,$$

but in this case the longer notation in Eq. (6.8) is 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 $F^{A,B} \Rightarrow F^{C,D}$ should be independent of the order of application:

$$\text{fmap}_{F^{\bullet},B}(f:A \Rightarrow C) ; \text{fmap}_{F^{\bullet},C}(g:B \Rightarrow D) = \text{fmap}_{F^{\bullet},A}(g:B \Rightarrow D) ; \text{fmap}_{F^{\bullet},D}(f:A \Rightarrow C) . \quad (6.9)$$

This equation is illustrated by the type diagram below.



Different paths in this diagram give the same results if they arrive at the same vertex (as mathematicians say, “the diagram commutes”). In this way, the diagram illustrates at once the commutativity law (6.9) and the definition (6.8) of `bimap`.

Let us verify the commutativity law for the bifunctor $F^{A,B} \triangleq A \times A \times B$:

left-hand side : $\text{fmap}_{F^{\bullet},B}(f:A \Rightarrow C) ; \text{fmap}_{F^{\bullet},C}(g:B \Rightarrow D)$

definitions of fmap : $= (a:A \times b_1^B \times b_2^B \Rightarrow f(a) \times b_1 \times b_2) ; (c^C \times b_1^B \times b_2^B \Rightarrow c \times g(b_1) \times g(b_2))$

compute composition : $= a:A \times b_1^B \times b_2^B \Rightarrow f(a) \times g(b_1) \times g(b_2) ,$

right-hand side : $\text{fmap}_{F^{\bullet},A}(g:B \Rightarrow D) ; \text{fmap}_{F^{\bullet},D}(f:A \Rightarrow C)$

definitions of fmap : $= (a:A \times b_1^B \times b_2^B \Rightarrow a \times g(b_1) \times g(b_2)) ; (a:A \times d_1^D \times d_2^D \Rightarrow f(a) \times d_1 \times d_2)$

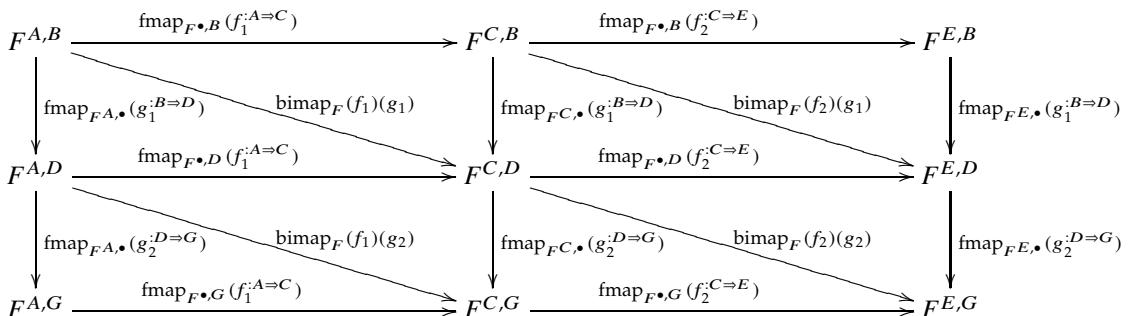
compute composition : $= a:A \times b_1^B \times b_2^B \Rightarrow f(a) \times g(b_1) \times g(b_2) .$

Both sides of the law are equal.

The commutativity law (6.9) leads to the composition law for `bimap`,

$$\text{bimap}_F(f_1:A \Rightarrow C)(g_1:B \Rightarrow D) ; \text{bimap}_F(f_2:C \Rightarrow E)(g_2:D \Rightarrow G) = \text{bimap}_F(f_1 ; f_2)(g_1 ; g_2) . \quad (6.10)$$

The following type diagram shows the relationships between various `bimap` and `fmap` functions:



To derive the composition law from Eq. (6.9), write

$$\text{bimap}_F(f_1)(g_1) ; \text{bimap}_F(f_2)(g_2)$$

use Eq. (6.8) : $= \text{fmap}_{F^{\bullet},B}(f_1) ; \underline{\text{fmap}_{F^{\bullet},C}(g_1)} ; \underline{\text{fmap}_{F^{\bullet},D}(f_2)} ; \text{fmap}_{F^{\bullet},E}(g_2)$

commutativity law (6.9) : $= \text{fmap}_{F^{\bullet},B}(f_1) ; \underline{\text{fmap}_{F^{\bullet},B}(f_2)} ; \text{fmap}_{F^{\bullet},E}(g_1) ; \underline{\text{fmap}_{F^{\bullet},E}(g_2)}$

composition laws : $= \text{fmap}_{F^{\bullet},B}(f_1 ; f_2) ; \text{fmap}_{F^{\bullet},E}(g_1 ; g_2)$

use Eq. (6.8) : $= \text{bimap}_F(f_1 ; f_2)(g_1 ; g_2) .$

Conversely, we can derive Eq. (6.9) from the composition law (6.10). We write the composition law with specially chosen functions:

$$\text{bimap}_F(f:A \Rightarrow C)(g:B \Rightarrow D) = \text{bimap}_F(\text{id}:A \Rightarrow A)(g:B \Rightarrow D) ; \text{bimap}_F(f:A \Rightarrow C)(\text{id}:D \Rightarrow D) . \quad (6.11)$$

Using Eq. (6.8), we find

$$\begin{aligned} \text{expect fmap}_{F^{\bullet,\bullet}}(g) ; \text{fmap}_{F^{\bullet,D}}(f) : & \text{ fmap}_{F^{\bullet,B}}(f:A \Rightarrow C) ; \text{fmap}_{F^{\bullet,C}}(g:B \Rightarrow D) \\ \text{use Eq. (6.8)} : & = \text{bimap}_F(f:A \Rightarrow C)(g:B \Rightarrow D) \\ \text{use Eq. (6.11)} : & = \text{bimap}_F(\text{id}:A \Rightarrow A)(g:B \Rightarrow D) ; \text{bimap}_F(f:A \Rightarrow C)(\text{id}:D \Rightarrow D) \\ \text{use Eq. (6.8)} : & = \underline{\text{fmap}_{F^{\bullet,B}}(\text{id})} ; \text{fmap}_{F^{\bullet,A}}(g) ; \text{fmap}_{F^{\bullet,D}}(f) ; \underline{\text{fmap}_{F^{\bullet,C}}(\text{id})} \\ \text{identity laws for } F : & = \text{fmap}_{F^{\bullet,A}}(g) ; \text{fmap}_{F^{\bullet,D}}(f) . \end{aligned}$$

The identity law for `bimap` holds as well,

$$\begin{aligned} \text{expect to equal id} : & \text{ bimap}_F(\text{id}:A \Rightarrow A)(\text{id}:B \Rightarrow B) \\ \text{use Eq. (6.8)} : & = \underline{\text{fmap}_{F^{\bullet,B}}(\text{id})} ; \underline{\text{fmap}_{F^{\bullet,C}}(\text{id})} \\ \text{identity laws for } F : & = \text{id} ; \text{id} = \text{id} . \end{aligned}$$

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 $F^{A,B} \triangleq A \times B \times B$ shows that the two `fmap` functions commute because they work on different parts of the data structure $F^{A,B}$. This turns out³ to be true in general: the commutativity law follows from the parametricity of the `fmap` functions. Because of this, 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. For brevity, we will talk about “covariant” and “contravariant” type parameters in those cases.

6.2.3 Type constructions for functors

What type expressions will produce a functor? Functional programming languages support the six standard type constructions (see Section 5.1.2). We 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. We will now derive the code for these constructions and prove their validity. We will use the code notation for brevity, occasionally showing the translation into the Scala syntax.

Statement 6.2.3.1 The type constructor $\text{Id}^A \triangleq A$ is a lawful functor (the **identity functor**).

Proof The `fmap` function is defined by

$$\begin{aligned} \text{fmap}_{\text{Id}} : (A \Rightarrow B) & \Rightarrow \text{Id}^A \Rightarrow \text{Id}^B \cong (A \Rightarrow B) \Rightarrow A \Rightarrow B , \\ \text{fmap}_{\text{Id}} \triangleq (f:A \Rightarrow B \Rightarrow f) & = \text{id}^{(A \Rightarrow B) \Rightarrow A \Rightarrow B} . \end{aligned}$$

The identity function is the only fully parametric implementation of the type signature $(A \Rightarrow B) \Rightarrow A \Rightarrow B$. Since the code of `fmap` is the identity function, the laws are satisfied automatically:

$$\begin{aligned} \text{identity law} : \text{fmap}_{\text{Id}}(\text{id}) &= \text{id}(\text{id}) = \text{id} , \\ \text{composition law} : \text{fmap}_{\text{Id}}(f ; g) &= f ; g = \text{fmap}_{\text{Id}}(f) ; \text{fmap}_{\text{Id}}(g) . \end{aligned}$$

³The proof is beyond the scope of this chapter; see <https://byorgey.wordpress.com/2018/03/30/>

Construction	Type notation	Comment
type parameter	$L^A \triangleq A$	the identity functor
product type	$L^A \triangleq P^A \times Q^A$	the functor product; P and Q must be functors
disjunctive type	$L^A \triangleq P^A + Q^A$	the functor co-product; P and Q must be functors
function type	$L^A \triangleq C^A \Rightarrow P^A$	the exponential functor; P is a functor and C a contrafunctor
primitive type	$L^A \triangleq Z$	the constant functor; Z is a fixed type
type constructor	$L^A \triangleq P^Q^A$	functor composition; P and Q are both functors or both contrafunctors
recursive type	$L^A \triangleq S^{A,L^A}$	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 $\text{Const}^{Z,A} \triangleq Z$ is a lawful functor (a **constant functor**) with respect to the type parameter A .

Proof The `fmap` function is defined by

$$\begin{aligned} \text{fmap}_{\text{Const}} : (A \Rightarrow B) &\Rightarrow \text{Const}^{Z,A} \Rightarrow \text{Const}^{Z,B} \cong (A \Rightarrow B) \Rightarrow Z \Rightarrow Z , \\ \text{fmap}_{\text{Const}}(f:A \Rightarrow B) &\triangleq (z:Z \Rightarrow z) = \text{id}^{Z \Rightarrow Z} . \end{aligned}$$

It is a constant function that ignores f and returns the identity $\text{id}^{Z \Rightarrow Z}$. The laws are satisfied:

$$\begin{aligned} \text{identity law} : \text{fmap}_{\text{Const}}(\text{id}) &= \text{id} , \\ \text{composition law} : \text{fmap}_{\text{Const}}(f \circ g) &= \text{id} = \text{fmap}_{\text{Const}}(f) \circ \text{fmap}_{\text{Const}}(g) = \text{id} \circ \text{id} . \end{aligned}$$

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 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.

Functor product We have seen that type constructors with product types, such as $L^A \triangleq A \times A \times A$, 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 $P^A \triangleq L^A \times M^A$ 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

$$f^{\uparrow P} \triangleq l^{\uparrow A} \times m^{\uparrow M^A} \Rightarrow f^{\uparrow L}(l) \times f^{\uparrow M}(m) .$$

Writing this code using the pipe (`▷`) operation makes it somewhat closer to the Scala syntax:

$$(l^{\uparrow A} \times m^{\uparrow M^A}) \triangleright f^{\uparrow P} \triangleq (l \triangleright f^{\uparrow L}) \times (m \triangleright f^{\uparrow M}) . \quad (6.12)$$

An alternative notation uses the **function product** symbol \boxtimes defined by

$$\begin{aligned} p^{A \Rightarrow B} \boxtimes q^{C \Rightarrow D} &: A \times C \Rightarrow B \times D , \\ p \boxtimes q &\triangleq a \times c \Rightarrow p(a) \times q(c) , \\ (a \times c) \triangleright (p \boxtimes q) &= (a \triangleright p) \times (b \triangleright q) . \end{aligned}$$

In this notation, the lifting for P is defined more concisely:

$$f^{\uparrow P} = f^{\uparrow L \times M} \triangleq f^{\uparrow L} \boxtimes f^{\uparrow M} . \quad (6.13)$$

We need to verify the identity law and the composition law.

To verify the identity law for P , pipe an arbitrary value of type $L^A \times M^A$ into both sides of the law:

$$\begin{aligned} \text{expect to equal } l \times m : & (l^{L^A} \times m^{M^A}) \triangleright \text{id}^{\uparrow P} \\ \text{definition of } f^{\uparrow P} : & = (l \triangleright \underline{\text{id}}^{\uparrow L}) \times (m \triangleright \underline{\text{id}}^{\uparrow M}) \\ \text{identity laws for } L, M : & = (\underline{l \triangleright \text{id}}) \times (\underline{m \triangleright \text{id}}) \\ \text{definition of id} : & = l \times m . \end{aligned}$$

To verify the composition law for P , we need to show that

$$f^{\uparrow P} ; g^{\uparrow P} = (f ; g)^{\uparrow P} .$$

Apply both sides of this equation to an arbitrary value of type $L^A \times M^A$:

$$\begin{aligned} \text{expect to equal } (l \times m) \triangleright (f ; g)^{\uparrow P} : & (l^{L^A} \times m^{M^A}) \triangleright f^{\uparrow P} ; g^{\uparrow P} \\ \triangleright \text{ notation} : & = (\underline{l^{L^A} \times m^{M^A}}) \triangleright \underline{f^{\uparrow P}} \triangleright g^{\uparrow P} \\ \text{use Eq. (6.12)} : & = ((l \triangleright f^{\uparrow L}) \times (m \triangleright f^{\uparrow M})) \triangleright g^{\uparrow P} \\ \text{use Eq. (6.12)} : & = (l \triangleright f^{\uparrow L} \triangleright g^{\uparrow L}) \times (m \triangleright f^{\uparrow M} \triangleright g^{\uparrow M}) \\ \triangleright \text{ notation} : & = (l \triangleright \underline{f^{\uparrow L} ; g^{\uparrow L}}) \times (m \triangleright \underline{f^{\uparrow M} ; g^{\uparrow M}}) \\ \text{composition laws for } L, M : & = (l \triangleright (f ; g)^{\uparrow L}) \times (m \triangleright (f ; g)^{\uparrow M}) \\ \text{use Eq. (6.12)} : & = (l \times m) \triangleright (f ; g)^{\uparrow P} . \end{aligned}$$

The calculations are shorter if we use the function product notation:

$$\begin{aligned} \text{expect to equal } (f ; g)^{\uparrow P} : & f^{\uparrow P} ; g^{\uparrow P} = (f^{\uparrow L} \boxtimes f^{\uparrow M}) ; (g^{\uparrow L} \boxtimes g^{\uparrow M}) \\ \text{composition of functions under } \boxtimes : & = (\underline{f^{\uparrow L} ; g^{\uparrow L}}) \boxtimes (\underline{f^{\uparrow M} ; g^{\uparrow M}}) \\ \text{composition laws for } L, M : & = (f ; g)^{\uparrow L} \boxtimes (f ; g)^{\uparrow M} = (f ; g)^{\uparrow P} . \end{aligned}$$

For comparison, the same derivation using the Scala code syntax looks like this,

```
(( 1, m )).map(f).map(g) == (( 1.map(f), m.map(f) )).map(g)
== (( 1.map(f).map(g), m.map(f).map(g) ))
== (( 1.map(f andThen g), m.map(f andThen g) ))
```

assuming that the `.map` method is defined on pairs by Eq. (6.12),

```
(( 1, m )).map(f) == (( 1.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:

$$m \triangleright f^{\uparrow M} \triangleright g^{\uparrow M} = m \triangleright f^{\uparrow M} ; g^{\uparrow M} = m \triangleright (f ; g)^{\uparrow M} .$$

By the conventions of the pipe notation, we have

$$(x \triangleright f) \triangleright g = x \triangleright f \triangleright g = x \triangleright f ; g = x \triangleright (f ; g) = (f ; g)(x) = g(f(x)) .$$

We will often use this notation in code derivations.

Statement 6.2.3.4 If P^A and Q^A are functors then $L^A \triangleq P^A + Q^A$ 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

$$\text{fmap}_L(f:A \Rightarrow B) = f^{\uparrow L} \triangleq \begin{vmatrix} & P^B & Q^B \\ \hline P^A & f^{\uparrow P} & \mathbf{0} \\ Q^A & \mathbf{0} & f^{\uparrow Q} \end{vmatrix}.$$

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

$$\text{fmap}_L(f) = f^{\uparrow L} = \begin{vmatrix} f^{\uparrow P} & \mathbf{0} \\ \mathbf{0} & f^{\uparrow Q} \end{vmatrix}. \quad (6.14)$$

To verify the identity law, use Eq. (6.14) and the identity laws for P and Q :

$$\begin{aligned} \text{expect to equal id : } \text{id}^{\uparrow L} &= \begin{vmatrix} \text{id}^{\uparrow P} & \mathbf{0} \\ \mathbf{0} & \text{id}^{\uparrow Q} \end{vmatrix} = \begin{vmatrix} \text{id} & \mathbf{0} \\ \mathbf{0} & \text{id} \end{vmatrix} \\ \text{identity function in matrix notation : } &= \text{id} \quad . \end{aligned}$$

To verify the composition law:

$$\begin{aligned} \text{expect to equal } (f \circ g)^{\uparrow L} : \quad f^{\uparrow L} \circ g^{\uparrow L} &= \begin{vmatrix} f^{\uparrow P} & \mathbf{0} \\ \mathbf{0} & f^{\uparrow Q} \end{vmatrix} \circ \begin{vmatrix} g^{\uparrow P} & \mathbf{0} \\ \mathbf{0} & g^{\uparrow Q} \end{vmatrix} \\ \text{matrix composition : } &= \begin{vmatrix} f^{\uparrow P} \circ g^{\uparrow P} & \mathbf{0} \\ \mathbf{0} & f^{\uparrow Q} \circ g^{\uparrow Q} \end{vmatrix} \\ \text{composition laws for } P, Q : &= \begin{vmatrix} (f \circ g)^{\uparrow P} & \mathbf{0} \\ \mathbf{0} & (f \circ g)^{\uparrow Q} \end{vmatrix} = (f \circ g)^{\uparrow L} \quad . \end{aligned}$$

The last two statements show that any type constructor built up only using primitive types, type parameters, products and co-products, such as $L^A \triangleq \mathbb{1} + (\text{String} + A) \times A \times \text{Int} + A$, is a functor. Functionors of this kind are called **polynomial functors** because they are analogous 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 products and disjunctions. 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 $L^A \triangleq C^A \Rightarrow P^A$ is a functor, called an **exponential functor**, with fmap defined by

$$\begin{aligned} \text{fmap}_L^{A,B}(f:A \Rightarrow B) : (C^A \Rightarrow P^A) &\Rightarrow C^B \Rightarrow P^B , \\ \text{fmap}_L(f:A \Rightarrow B) = f^{\uparrow L} &\triangleq h:C^A \Rightarrow P^A \Rightarrow f^{\downarrow C} ; h ; f^{\uparrow P} . \end{aligned} \quad (6.15)$$

The corresponding Scala code is

```
def fmap_L[A, B](f: A => B)(h: C[A] => P[A]): C[B] => P[B] = {
    contrafmap_C(f) andThen h andThen fmap_P(f)
}
```

A type diagram for fmap_L can be drawn as

$$\begin{array}{ccccc} & & C^A & \xrightarrow{h} & P^A \\ & \nearrow \text{contrafmap}_C(f:A \Rightarrow B) & & & \searrow \text{fmap}_P(f:A \Rightarrow B) \\ C^B & \xrightarrow{\text{fmap}_L(f:A \Rightarrow B)(h:C^A \Rightarrow P^A)} & & & P^B \end{array}$$

Proof Since the types are already checked, we can use Eq. (6.15) without type annotations,

$$h \triangleright f^{\uparrow L} = f^{\downarrow C} ; h ; f^{\uparrow P} . \quad (6.16)$$

To verify the identity law for L , show that $\text{id}^{\uparrow L}(h) = h$:

$$\begin{aligned} \text{expect to equal } h : & h \triangleright \text{id}^{\uparrow L} \\ \text{definition (6.16) of } \uparrow^L : & = \underline{\text{id}}^{\downarrow C} ; h ; \underline{\text{id}}^{\uparrow P} \\ \text{identity laws for } C \text{ and } P : & = \underline{\text{id}} ; h ; \underline{\text{id}} \\ \text{definition of id} : & = h . \end{aligned}$$

To verify the composition law for L :

$$\begin{aligned} \text{expect to equal } h \triangleright f^{\uparrow L} ; g^{\uparrow L} : & h \triangleright (f ; g)^{\uparrow L} \\ \text{definition (6.16) of } \uparrow^L : & = (\underline{f} ; \underline{g})^{\downarrow C} ; h ; (\underline{f} ; \underline{g})^{\uparrow P} \\ \text{composition laws for } C \text{ and } P : & = g^{\downarrow C} ; f^{\downarrow C} ; h ; f^{\uparrow P} ; g^{\uparrow P} \\ \text{definition (6.16) of } \uparrow^L : & = \underline{g}^{\downarrow C} ; (h \triangleright f^{\uparrow L}) ; \underline{g}^{\uparrow P} \\ \text{definition (6.16) of } \uparrow^L : & = (h \triangleright f^{\uparrow L}) \triangleright g^{\uparrow L} = h \triangleright f^{\uparrow L} ; g^{\uparrow L} . \end{aligned}$$

It is important for this proof that C is a contrafunctor and so the order of lifted function compositions is reversed, $(f ; g)^{\downarrow C} = g^{\downarrow C} ; f^{\downarrow C}$. If C were a functor, the proof would not work: we would have obtained $f^{\uparrow C} ; g^{\uparrow C}$ instead of $g^{\downarrow C} ; f^{\downarrow C}$, and we would not be able to group $f^{\downarrow C} ; h ; f^{\uparrow P}$ together (the order of composition cannot be permuted for arbitrary functions f, g).

Examples of functors obtained via the exponential functor construction are $L^A \triangleq Z \Rightarrow A$ (with the contrafunctor C^A chosen as the constant contrafunctor Z , where Z is a fixed type) and $L^A \triangleq (A \Rightarrow Z) \Rightarrow A$ (with the contrafunctor $C^A \triangleq A \Rightarrow Z$). Statement 6.2.3.5 generalizes those examples to arbitrary contrafunctors C^A used as arguments of function types.

In a similar way, one can prove that $P^A \Rightarrow C^A$ is a contrafunctor (Exercise 6.3.1.2). Together with the results of Statements 6.2.3.3–6.2.3.5, this establishes the rules of reasoning about covariance and contravariance of type parameters in arbitrary type expressions. Every function arrow flips the variance from covariant to contravariant and back. For instance, the identity functor $L^A \triangleq A$ is covariant in A , while $A \Rightarrow Z$ is contravariant in A , and $(A \Rightarrow Z) \Rightarrow Z$ is again covariant in A . As we

have seen, $A \Rightarrow A \Rightarrow Z$ is contravariant in A , so any number of curried arrows count as one in this consideration (and, in any case, $A \Rightarrow A \Rightarrow Z \cong A \times A \Rightarrow Z$). Products and disjunctions do not change variance, so $(A \Rightarrow Z_1) \times (A \Rightarrow Z_2) + (A \Rightarrow Z_3)$ is still contravariant in A . We will see more examples of such reasoning below (Section 6.2.5).

The remaining constructions set a type parameter to another type constructor. 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 type constructors. Viewed in this way, type constructors are **type-level functions** (i.e. maps on the set of types). For this reason, functor composition can be also denoted by $P \circ Q$, similar to 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 $f: A \Rightarrow B$ into a function of type `List[Option[A]] \Rightarrow List[Option[B]]`. In this way, we may perform the `.map` operation on the composite 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 $L^A \triangleq P^Q^A$ 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

$$\begin{aligned} \text{fmap}_L : f: A \Rightarrow B \Rightarrow P^Q^A \Rightarrow P^Q^B &, \\ \text{fmap}_L(f) \triangleq \text{fmap}_P(\text{fmap}_Q(f)) &, \quad \text{fmap}_L \triangleq \text{fmap}_Q \circ \text{fmap}_P &, \\ \text{in a shorter notation : } f^{\uparrow L} \triangleq (f^{\uparrow Q})^{\uparrow P} \triangleq f^{\uparrow Q \uparrow P} &. \end{aligned} \quad (6.17)$$

Proof To verify the identity law for L , use the identity laws for P and Q :

$$\text{id}^{\uparrow L} = (\text{id}^{\uparrow Q})^{\uparrow P} = \text{id}^{\uparrow P} = \text{id} .$$

To verify the composition law for L , use the composition laws for P and Q :

$$(f \circ g)^{\uparrow L} = ((f \circ g)^{\uparrow Q})^{\uparrow P} = (f^{\uparrow Q} \circ g^{\uparrow Q})^{\uparrow P} = f^{\uparrow Q \uparrow P} \circ g^{\uparrow Q \uparrow P} .$$

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

$$\text{List}^A \triangleq \mathbb{1} + A \times \text{List}^A .$$

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]
```

Description	Type definition	Bifunctor $S^{A,R}$
list	$L^A \triangleq \mathbb{1} + A \times L^A$	$S^{A,R} \triangleq \mathbb{1} + A \times R$
non-empty list	$\text{NEL}^A \triangleq A + A \times \text{NEL}^A$	$S^{A,R} \triangleq A + A \times R$
list of odd length	$L^A \triangleq A + A \times A \times L^A$	$S^{A,R} \triangleq A + A \times A \times R$
binary tree	$L^A \triangleq A + L^A \times L^A$	$S^{A,R} \triangleq A + R \times R$
rose tree	$L^A \triangleq A + \text{NEL}^{L^A}$	$S^{A,R} \triangleq A + \text{NEL}^R$
regular-shaped tree	$L^A \triangleq A + L^{A \times A}$	not possible
abstract syntax tree	$L^A \triangleq P^A + Q^{L^A}$	$S^{A,R} = P^A + Q^R$

Table 6.3: Recursive disjunctive types defined using type equations.

is defined by

$$\text{Tree}_2^A \triangleq A + \text{Tree}_2^A \times \text{Tree}_2^A .$$

These definitions of recursive types look like “type equations”. We can generalize these examples to a recursive definition

$$L^A \triangleq S^{A,L^A} , \quad (6.18)$$

where $S^{A,R}$ is a suitably chosen type constructor with two type parameters A, R . If the type constructor $S^{\bullet,\bullet}$ 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]])
```

We must use a case 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
      type L[A] = Either[A, L[A]]
                  ^

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. (6.18) always defines a functor when $S^{\bullet,\bullet}$ is a bifunctor.

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 functor,

$$L^A \triangleq S^{A,L^A} .$$

The `fmap` method for L is a recursive function implemented as

$$\text{fmap}_L(f:A \Rightarrow B) \triangleq \text{bimap}_S(f)(\text{fmap}_L(f)) . \quad (6.19)$$

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 contramap(f).
```

```
L(newX)           // Need to wrap the value of type S[B, L[B]] into the type constructor L.
}
```

Proof This is the first time we prove a property of a recursive function (fmap_L). Since the implementation of fmap_L is recursive, its code contains some calls to itself. When we are proving some property of fmap_L , we may *assume* in the proof that the property already holds for all recursive calls of fmap_L . As long as recursion is not infinite, this kind of proof will be equivalent to a proof by induction: The base case corresponds to the final non-recursive evaluation within the code of fmap_L , and the inductive assumption states that the recursive calls to fmap_L already satisfy the property we are trying to prove.

For clarity, we add an overline to recursive calls in the code:

$$\text{fmap}_L(f) \triangleq \text{bimap}_S(f)(\overline{\text{fmap}_L}(f)) .$$

To prove the identity law:

$$\begin{aligned} \text{expect to equal id : } & \text{fmap}_L(\text{id}) \\ \text{definition of } \text{fmap}_L : & = \text{bimap}_S(\text{id})(\overline{\text{fmap}_L}(\text{id})) \\ (\text{inductive assumption}) \text{ the law holds for } \overline{\text{fmap}_L} : & = \text{bimap}_S(\text{id})(\text{id}) \\ \text{identity law for } S : & = \text{id} . \end{aligned}$$

To prove the composition law:

$$\begin{aligned} \text{expect to equal } \text{fmap}_L(f \circ g) : & \text{fmap}_L(f) \circ \text{fmap}_L(g) \\ \text{definition of } \text{fmap}_L : & = \text{bimap}_S(f)(\overline{\text{fmap}_L}(f)) \circ \text{bimap}_S(g)(\overline{\text{fmap}_L}(g)) \\ \text{composition law for } S : & = \text{bimap}_S(f \circ g)(\overline{\text{fmap}_L}(f) \circ \overline{\text{fmap}_L}(g)) \\ \text{inductive assumption : } & = \text{bimap}_S(f \circ g)(\overline{\text{fmap}_L}(f \circ g)) \\ \text{definition of } \text{fmap}_L : & = \text{fmap}_L(f \circ g) . \end{aligned}$$

For the regular-shaped binary tree, the construction (6.18) 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 regular-shaped trees, we need to generalize the construction by adding another arbitrary functor, P^\bullet , in the type argument of L^\bullet :

$$L^A \triangleq S^{A,L^{P^A}} . \quad (6.20)$$

Regular-shaped trees are defined by Eq. (6.20) with $S^{A,R} \triangleq A + R$ and $P^A \triangleq A \times A$. 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]]]) // Equivalently, case class L[A](s: Either[A, L[(A, A)]])
```

Different choices of P will define regular-shaped trees with different kinds of branching.

6.2.4 Type constructions for contrafunctors

The previous section performed **structural analysis** for functors: a systematic search for type constructions that create new functors. *Mutatis mutandis*, similar constructions work for contrafunctors,

Construction	Type notation	Comment
tuple	$C^A \triangleq P^A \times Q^A$	the product contrafunctor; P and Q must be contrafunctors
disjunctive type	$C^A \triangleq P^A + Q^A$	the co-product contrafunctor; P and Q must be contrafunctors
function type	$C^A \triangleq L^A \Rightarrow H^A$	the exponential contrafunctor; L is a functor and H a contrafunctor
primitive type	$C^A \triangleq Z$	the constant contrafunctor; Z is a fixed type
type constructor	$C^A \triangleq P^Q^A$	the composition; P is a functor and Q a contrafunctor (or vice versa)
type recursion	$C^A \triangleq S^{A,C^A}$	$S^{A,B}$ must be a contrafunctor w.r.t. A and functor w.r.t. B

Table 6.4: Type constructions defining a contrafunctor C^A .

as shown in Table 6.4. One difference with respect to Table 6.2 is that the identity type constructor, $L^A \triangleq A$, is missing because it is a functor and not a contrafunctor. However, the constant type constructor, $L^A \triangleq Z$, 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 $C^A \triangleq Z$ is a contrafunctor (the **constant contrafunctor**) whose `contrafmap` returns an identity function of type $Z \Rightarrow Z$:

```
type Const[Z, A] = Z
def contrafmap[Z, A, B](f: B => A): Const[Z, A] => Const[Z, B] = identity[Z]
```

Proof All laws hold because `contrafmap` returns an identity function:

$$\begin{aligned} \text{identity law : } & \text{contrafmap(id)} = \text{id} \\ \text{composition law : } & \text{contrafmap}(f) ; \text{contrafmap}(g) = \text{id} ; \text{id} = \text{id} = \text{contrafmap}(g ; f) \end{aligned} .$$

Statement 6.2.4.2 If P^A is a functor and Q^A is a contrafunctor then $L^A \triangleq P^Q^A$ is a contrafunctor with `contrafmap` defined by

```
def contrafmap[A, B](f: B => A): P[Q[A]] => P[Q[B]] = fmap_P(contrafmap_Q(f))
```

where lawful implementations of `fmap_P` and `contrafmap_Q` are assumed to be given.

Proof Convert the Scala implementation of `contrafmap_L` into the code notation:

$$\text{contrafmap}_L(f^{B \Rightarrow A}) \triangleq \text{fmap}_P(\text{contrafmap}_Q(f)) .$$

It is easier to reason about this function if we rewrite it as

$$f^{\downarrow L} \triangleq (f^{\downarrow Q})^{\uparrow P} .$$

The contrafunctor laws for L are then proved like this:

$$\begin{aligned} \text{identity law : } & \text{id}^{\downarrow L} = (\text{id}^{\downarrow Q})^{\uparrow P} = \text{id}^{\uparrow P} = \text{id} \\ \text{composition law : } & f^{\downarrow L} ; g^{\downarrow L} = (f^{\downarrow Q})^{\uparrow P} ; (g^{\downarrow Q})^{\uparrow P} \\ \text{use } P\text{'s composition law : } & = (\underline{f^{\downarrow Q}} ; \underline{g^{\downarrow Q}})^{\uparrow P} = ((g ; f)^{\downarrow Q})^{\uparrow P} = (g ; f)^{\downarrow L} \end{aligned} .$$

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

$$S^{A,R} \triangleq (A \Rightarrow \text{Int}) + R \times R . \quad (6.21)$$

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]
```

$$\begin{aligned} \text{xmap}_S : (B \Rightarrow A) \Rightarrow (Q \Rightarrow R) \Rightarrow S^{A,Q} \Rightarrow S^{B,R} &, \\ \text{xmap}_S(f: B \Rightarrow A)(g: Q \Rightarrow R) &\triangleq \text{fmap}_{S^{A,\bullet}}(g) \circ \text{contrafmap}_{S^{\bullet,R}}(f) \quad . \end{aligned}$$

The function `xmap` obeys the laws of identity and composition:

$$\text{identity law : } \text{xmap}_S(\text{id})(\text{id}) = \text{id} \quad , \quad (6.22)$$

$$\text{composition law : } \text{xmap}_S(f_1)(g_1) \circ \text{xmap}_S(f_2)(g_2) = \text{xmap}_S(f_2 \circ f_1)(g_2 \circ g_1) \quad . \quad (6.23)$$

These laws are similar to the identity and composition laws for bifunctors (Section 6.2.2), except for inverting the order of the composition ($f_2 \circ f_1$). The laws hold automatically whenever the functor and contrafunctor methods for S ($\text{fmap}_{S^{A,\bullet}}$ and $\text{contrafmap}_{S^{\bullet,R}}$) 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 equation

$$L^A \triangleq S^{A,L^A} \triangleq (A \Rightarrow \text{Int}) + L^A \times L^A \quad ,$$

we obtain a contrafunctor in the shape of a binary tree whose leaves are functions of type $A \Rightarrow \text{Int}$. 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,

$$C^A \triangleq S^{A,C^A} \quad .$$

Given the functions `contrafmap_{S^{\bullet,R}}` and `fmap_{S^{A,\bullet}}` for S , we implement `contrafmap_C` as

$$\begin{aligned} \text{contrafmap}_C(f: B \Rightarrow A) : C^A \Rightarrow C^B &\cong S^{A,C^A} \Rightarrow S^{B,C^B} \quad , \\ \text{contrafmap}_C(f: B \Rightarrow A) &\triangleq \text{xmap}_S(f)(\overline{\text{contrafmap}}_C(f)) \quad . \end{aligned}$$

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 contrafmap[A, B](f: B => A): C[A] => C[B] = { case C(x) =>
  val sbcb: S[B, C[B]] = xmap_S(f)(contrafmap(f))(x) // Recursive call to contrafmap(f).
  C(sbcb) // Need to wrap the value of type S[B, C[B]] into the type constructor C.
}
```

Proof The code of `contrafmap` is recursive, and the recursive call is marked by an overline:

$$\text{contrafmap}_C(f) \triangleq f \downarrow^C \triangleq \text{xmap}_S(f)(\overline{\text{contrafmap}}_C(f)) \quad .$$

To verify the identity law:

$$\begin{aligned} \text{expect to equal id : } \text{contrafmap}_C(\text{id}) &= \text{xmap}_S(\text{id})(\overline{\text{contrafmap}}_C(\text{id})) \\ \text{inductive assumption : } &= \text{xmap}_S(\text{id})(\text{id}) \\ \text{identity law for xmap}_S : &= \text{id} \quad . \end{aligned}$$

To verify the composition law:

$$\begin{aligned} \text{expect to equal } (g \downarrow^C ; f \downarrow^C) : & (f: D \Rightarrow B ; g: B \Rightarrow A)^{\downarrow C} = \text{xmap}_S(f \circ g)(\overline{\text{contrafmap}}_C(f \circ g)) \\ \text{inductive assumption : } &= \text{xmap}_S(f \circ g)(\overline{\text{contrafmap}}_C(g) ; \overline{\text{contrafmap}}_C(f)) \\ \text{composition laws for xmap}_S : &= \text{xmap}_S(g)(\overline{\text{contrafmap}}_C(g)) ; \text{xmap}_S(f)(\overline{\text{contrafmap}}_C(f)) \\ \text{definition of } \downarrow^C : &= g \downarrow^C ; f \downarrow^C \quad . \end{aligned}$$

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 type constructions (Section 5.1.2) gives either a new functor or a new contrafunctor. So, we expect to be able to decide, for *every* type expression built from primitive types, type parameters, product types, sum types, function (exponential) types, and type parameter substitutions, whether it 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 all their type parameters. 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, $\text{Int} \Rightarrow A$ is covariant in A .
- Each time a type parameter is placed to the left of an *uncurried* function arrow \Rightarrow , the variance is reversed: covariant becomes contravariant and vice versa. For example,

$$\begin{aligned} \text{this is covariant in } A : & \quad \mathbb{1} + A \times A \quad , \\ \text{this is contravariant in } A : & \quad (\mathbb{1} + A \times A) \Rightarrow \text{Int} \quad , \\ \text{this is covariant in } A : & \quad ((\mathbb{1} + A \times A) \Rightarrow \text{Int}) \Rightarrow \text{Int} \quad , \\ \text{this is contravariant in } A : & \quad (((\mathbb{1} + A \times A) \Rightarrow \text{Int}) \Rightarrow \text{Int}) \Rightarrow \text{Int} \quad . \end{aligned}$$

- Repeated curried function arrows work as one arrow: $A \Rightarrow \text{Int}$ is contravariant in A , and $A \Rightarrow A \Rightarrow A \Rightarrow \text{Int}$ is still contravariant in A . This is so because the type $A \Rightarrow A \Rightarrow A \Rightarrow \text{Int}$ is equivalent to $A \times A \times A \Rightarrow \text{Int}$, which is of the form $F^A \Rightarrow \text{Int}$ with a type constructor $F^A \triangleq A \times A \times A$. Exercise 6.3.1.1 will show that $F^A \Rightarrow \text{Int}$ is contravariant in A .
- Nested type constructors combine their variances: e.g. if we know that F^A is contravariant in A then $F^{A \Rightarrow \text{Int}}$ is covariant in A , while $F^{A \times A \times A}$ is contravariant in A .

For any exponential-polynomial type expression, such as Eq. (6.7),

$$Z^{A,R} \triangleq ((A \Rightarrow A \Rightarrow R) \Rightarrow R) \times A + (\mathbb{1} + R \Rightarrow A + \text{Int}) + A \times A \times \text{Int} \times \text{Int} \quad ,$$

we can mark the position of each type parameter as either covariant (+) or contravariant (-), according to the number of nested function arrows:

$$((A \underset{+}{\Rightarrow} A \underset{-}{\Rightarrow} R) \underset{+}{\Rightarrow} R) \times A + (\mathbb{1} + R \underset{-}{\Rightarrow} A \underset{+}{+} \text{Int}) + A \underset{+}{\times} A \underset{+}{\times} \text{Int} \underset{+}{\times} \text{Int} \quad .$$

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

$$\text{map}_Z : Z^{A,R} \Rightarrow (A \Rightarrow B) \Rightarrow Z^{B,R} \quad ,$$

we could derive a fully parametric, information-preserving implementation of `map`. We could then look for equational 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` and proofs of laws for each new functor and contrafunctor. The next examples illustrate the procedure for a simpler type constructor.

Example 6.2.5.1 Rewrite this Scala definition in the type notation and decide whether it is covariant or contravariant with respect to each type parameter:

```
final case class G[A, Z](p: Either[Int, A], q: Option[Z => Int => Z => (Int, A)])
```

Solution The type notation is

$$G^{A,Z} \triangleq (\text{Int} + A) \times (\mathbb{1} + (Z \Rightarrow \text{Int} \Rightarrow Z \Rightarrow \text{Int} \times A)) .$$

Mark the covariant and the contravariant positions in this type expression:

$$(\text{Int} + A) \underset{+}{\times} (\mathbb{1} \underset{-}{+} (Z \Rightarrow \text{Int} \Rightarrow Z \underset{-}{\Rightarrow} \text{Int} \times A)) .$$

All Z positions in the sub-expression $Z \Rightarrow \text{Int} \Rightarrow Z \Rightarrow \text{Int} \times A$ are contravariant since the function arrows are curried rather than nested. We see that A is always in covariant positions (+) while Z is always in contravariant positions (-). It follows that $G^{A,Z}$ is covariant in A and 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 “product functor” construction (Statement 6.2.3.3),

$$G^{A,Z} \cong G_1^A \times G_2^{A,Z} ,$$

$$\text{where } G_1^A \triangleq \text{Int} + A \quad \text{and} \quad G_2^{A,Z} \triangleq \mathbb{1} + (Z \Rightarrow \text{Int} \Rightarrow Z \Rightarrow \text{Int} \times A) .$$

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 “co-product functor” construction (Statement 6.2.3.3) produces a `map` implementation for G_1^A together with a proof that it satisfies the functor laws:

$$\text{fmap}_{G_1}(f:A \Rightarrow B) = f^{\uparrow G_1} \triangleq \begin{array}{c|cc} & \text{Int} & B \\ \hline \text{Int} & \text{id} & \mathbf{0} \\ A & \mathbf{0} & f \end{array} .$$

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 `Int` \times `A`. This tells us to use the “co-product functor”, the “exponential functor”, and the “product functor” constructions. Write down the functor constructions needed at each step as we decompose $G_2^{A,Z}$:

$$G_2^{A,Z} \triangleq \mathbb{1} + (Z \Rightarrow \text{Int} \Rightarrow Z \Rightarrow \text{Int} \times A) .$$

$$\text{co-product functor : } G_2^{A,Z} \cong \mathbb{1} + G_3^{A,Z} \quad \text{where } G_3^{A,Z} \triangleq Z \Rightarrow \text{Int} \Rightarrow Z \Rightarrow \text{Int} \times A .$$

$$\text{exponential functor : } G_3^{A,Z} \cong Z \Rightarrow G_4^{A,Z} \quad \text{where } G_4^{A,Z} \triangleq \text{Int} \Rightarrow Z \Rightarrow \text{Int} \times A .$$

$$\text{exponential functor : } G_4^{A,Z} \cong \text{Int} \Rightarrow G_5^{A,Z} \quad \text{where } G_5^{A,Z} \triangleq Z \Rightarrow \text{Int} \times A .$$

$$\text{exponential functor : } G_5^{A,Z} \cong Z \Rightarrow G_6^A \quad \text{where } G_6^A \triangleq \text{Int} \times A .$$

$$\text{product functor : } G_6^A \cong \text{Int} \times A \cong \text{Const}^{\text{Int}, A} \times \text{Id}^A .$$

Each of the type constructors G_1, \dots, G_6 is a functor in A because all of the functor constructions preserve 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 $f^{\uparrow L} \triangleq \text{fmap}_L(f)$ and $x \triangleright f^{\uparrow L}$ instead of the Scala code `x.map(f)` throughout the derivations:

$$\begin{aligned} \text{product functor : } G^{A,Z} &\cong G_1^A \times G_2^{A,Z} \quad , \quad (g_1 \times g_2) \triangleright f^{\uparrow G} = (g_1 \triangleright f^{\uparrow G_1}) \times (g_2 \triangleright f^{\uparrow G_2}) \quad . \\ \text{co-product functor : } G_1^A &\triangleq \text{Int} + A \quad , \quad f^{\uparrow G_1} = \begin{vmatrix} \text{id} & 0 \\ 0 & f \end{vmatrix} \quad . \\ \text{co-product functor : } G_2^{A,Z} &\triangleq \text{Int} + G_3^{A,Z} \quad , \quad f^{\uparrow G_2} = \begin{vmatrix} \text{id} & 0 \\ 0 & f^{\uparrow G_3} \end{vmatrix} \quad . \\ \text{exponential functor : } G_3^{A,Z} &\triangleq Z \Rightarrow G_4^{A,Z} \quad , \quad g_3 \triangleright f^{\uparrow G_3} = g_3 ; f^{\uparrow G_4} = (z^Z \Rightarrow g_3(z)) \triangleright f^{\uparrow G_4} \quad . \end{aligned}$$

Applying the exponential functor construction three times, we finally obtain

$$\begin{aligned} G_3^{A,Z} &\triangleq Z \Rightarrow \text{Int} \Rightarrow Z \Rightarrow G_6^A \quad , \quad g_3 \triangleright f^{\uparrow G_3} = z_1^Z \Rightarrow n^{\text{Int}} \Rightarrow z_2^Z \Rightarrow g_3(z_1)(n)(z_2) \triangleright f^{\uparrow G_6} \quad . \\ G_6^A &\triangleq \text{Int} \times A \quad , \quad (i \times a) \triangleright f^{\uparrow G_6} = i \times f(a) \quad . \end{aligned}$$

We can now write the corresponding Scala code for `fmapG`:

```
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.
}
```

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 `contrafmap` function that satisfies the appropriate laws.
- Use constructions to derive the correct code of `fmap` or `contrafmap` 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 $C \triangleq L^A \Rightarrow H^A$ is a contrafunctor with the `contrafmap` method defined by the following code:

```
def contrafmap[A, B](f: B => A)(c: L[A] => H[A]): L[B] => H[B] = {
    1b: L[B] => contrafmap_H(f)(c(fmap_L(f)(1b)))
}
```

Here, `contrafmap_H` and `fmap_L` are the methods already defined for H and L .

Exercise 6.3.1.2 In each case, implement the required `fmap` or `contrafmap` function for the new type constructor 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 the appropriate laws.

- (a) $L^A \triangleq F^A \times G^A$ is a contrafunctor if F^A and G^A are contrafunctors.
- (b) $L^A \triangleq F^A + G^A$ is a contrafunctor if F^A and G^A are contrafunctors.
- (c) $L^A \triangleq F^{G^A}$ is a functor when both F and G are contrafunctors.
- (d) $L^A \triangleq F^{G^A}$ is a contrafunctor when F is a contrafunctor and G is a functor.

Exercise 6.3.1.3 Show that the type constructor L defined by Eq. (6.20) is a functor for any given bifunctor S and functor P .

Exercise 6.3.1.4 Show that $L^A \triangleq F^A \Rightarrow G^A$ is, in general, neither a functor nor a contrafunctor when both F^A and G^A are functors or both are contrafunctors (an example of suitable F^A and G^A is sufficient).

Exercise 6.3.1.5 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 6.3.1.6 Using the known constructions, determine which of the following are functors or contrafunctors (or neither) and implement `fmap` or `contrafmap` if appropriate. Answer this question with respect to each type parameter separately.

- (a) $F^A \triangleq \text{Int} \times A \times A + (\text{String} \Rightarrow A) \times A$.
- (b) $G^{A,B} \triangleq (A \Rightarrow \text{Int} \Rightarrow \text{Int}) + (A \Rightarrow \text{Int} \Rightarrow A \Rightarrow \text{Int})$.
- (c) $H^{A,B,C} \triangleq (A \Rightarrow A \Rightarrow B \Rightarrow C) \times C + (B \Rightarrow A)$.
- (d) $P^{A,B} \triangleq (((A \Rightarrow B) \Rightarrow A) \Rightarrow B) \Rightarrow A$.

Exercise 6.3.1.7 Show that the recursive type constructor L^\bullet defined by

$$L^A \triangleq \text{Int} + A + L^A$$

is a functor, and implement a `map` or `fmap` function for L in Scala.

6.4 Discussion

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

$$P^A \triangleq A + (A \Rightarrow \text{Int}) \quad .$$

It is not possible to define either a `map` or a `contramap` function for P : the required type signatures cannot be implemented. However, we can implement a function called `xmap`, with the type signature

$$\text{xmap}_P : (B \Rightarrow A) \Rightarrow (A \Rightarrow B) \Rightarrow P^A \Rightarrow P^B \quad .$$

To see why, let us temporarily rename the contravariant occurrence of A to Z and define a new type constructor \tilde{P} by

$$\tilde{P}^{Z,A} \triangleq A + (Z \Rightarrow \text{Int}) \quad .$$

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 `xmap_P` as a composition of `fmap` with respect to A and `contrafmap`

with respect to Z , similarly to what we saw in the proof of Statement 6.2.4.3. The function xmap_P will satisfy the identity and composition laws (6.22)–(6.23). Setting the type parameter $Z = A$, we will obtain the xmap_P function for P . The identity and composition laws for xmap_P will hold, since the laws of $\tilde{P}^{Z,A}$ hold for all type parameters:

$$P\text{'s identity law : } \text{xmap}_P(\text{id}^{A \Rightarrow A})(\text{id}^{A \Rightarrow A}) = \text{id} ,$$

$$P\text{'s composition law : } \text{xmap}_P(f_1^{B \Rightarrow A})(g_1^{A \Rightarrow B}) ; \text{xmap}_P(f_2^{C \Rightarrow B})(g_2^{B \Rightarrow C}) = \text{xmap}_P(f_2 ; f_1)(g_1 ; g_2) .$$

A type constructor P^A with these properties ($P^A \cong \tilde{P}^{A,A}$ where $\tilde{P}^{Z,A}$ has a lawful xmap_P) is called a **profunctor**. Sometimes the type constructor $\tilde{P}^{Z,A}$ is also called a profunctor.

Consider an exponential-polynomial type constructor P^A , no matter how complicated, such as

$$P^A \triangleq (\mathbb{1} + A \times A \Rightarrow A) \times A \Rightarrow \mathbb{1} + (A \Rightarrow A + \text{Int}) .$$

Each copy of a type parameter will occur either in covariant or in a contravariant position because no other possibility is available in exponential-polynomial type expressions. 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 $\text{xmap}_{\tilde{P}}$ satisfying the identity and composition laws. Setting $Z = A$, we will obtain a lawful xmap_P , which makes P a profunctor. Thus, *every* exponential-polynomial type constructor is a profunctor.

An unfuncor, 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.

6.4.2 Subtyping with injective or surjective conversion functions

In some cases, P is a subtype of Q when the set of values of P is a *subset* of values of Q . In other words, the conversion function $P \Rightarrow Q$ 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, $\mathbb{0} + A <: \mathbb{1} + A$). The set of all values of type `Some[A]` is a subset of the set of values of type `Option[A]`, and the conversion function is injective because it is an identity function, $\mathbb{0} + x:A \Rightarrow \mathbb{0} + x$, that merely reassigns types.

However, subtyping does not necessarily imply that the conversion function is injective. An example of a subtyping relation with a *surjective* conversion function is between the function types $P \triangleq \mathbb{1} + A \Rightarrow \text{Int}$ and $Q \triangleq \mathbb{0} + A \Rightarrow \text{Int}$ (in Scala, $P = \text{Option[A] } \Rightarrow \text{Int}$ and $Q = \text{Some[A] } \Rightarrow \text{Int}$). We have $P <: Q$ because $P \cong C^{\mathbb{1}+A}$ and $Q \cong C^{\mathbb{0}+A}$, where $C^X \triangleq X \Rightarrow \text{Int}$ is a contrafunctor. The conversion function $P \Rightarrow Q$ 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, $p \Rightarrow x \Rightarrow p(x)$ is easily seen to be the same as $p \Rightarrow p$.

Nevertheless, it is not true that all information from a value of type P is preserved in a value of type Q : the type P describes functions that also accept `None` as an argument, while functions of type Q do not. So, there is strictly more information in the type P than in Q . The conversion function $\text{p2q} : P \Rightarrow Q$ is surjective.

We have now seen examples of either injective or surjective type conversions. Suppose $P_1 <: Q_1$ and $P_2 <: Q_2$, and consider the product types $P_1 \times P_2$ and $Q_1 \times Q_2$. Since the product type is one of the functor constructions, the product $A \times B$ is covariant in both type parameters. It follows that $P_1 \times P_2 <: Q_1 \times Q_2$. If $r_1 : P_1 \Rightarrow Q_1$ is injective but $r_2 : P_2 \Rightarrow Q_2$ is surjective, the function product $r_1 \boxtimes r_2 : P_1 \times Q_1 \Rightarrow P_2 \times Q_2$ is neither injective nor surjective. So, type conversion functions are not necessarily injective or surjective; they can also be anything “in between”.

An important property of functor liftings is that they preserve injectivity and surjectivity: if a function $f:A \Rightarrow B$ is injective, it is lifted to an injective function $f^{\uparrow L} : L^A \Rightarrow L^B$; and similarly for surjective functions f . Let us prove this property for injective functions; the proof for surjective functions is quite similar.

Statement 6.4.2.1 If L^A is a lawful functor and $f:A \Rightarrow B$ is an injective function then $\text{fmap}_L(f)$ is also an injective function of type $L^A \Rightarrow L^B$.

Proof We begin by noting that an injective function $f:A \Rightarrow B$ must somehow embed all information from a value of type A into a value of type B . The **image** of f (the subset of all values of type B that can be obtained as $f(a)$ for some $a:A$) thus contains a distinct value of type B for each distinct value of type A . So, there exists a function that maps any b from the image of f back to a value $a:A$ it came from; call that function $g:B \Rightarrow A$. The function g must satisfy

$$\forall a:A. g(f(a)) = a ;$$

equivalently written as

$$g \circ f = \text{id} .$$

It is important that g is a partial function. The function g is partial because it is defined only for a subset of values of type B , which are in the image of f . Despite the equation $g \circ f = \text{id}$, the function g is not an inverse for f . An inverse function for f must be a *total* (not a partial) function h satisfying $h \circ f = \text{id}$ and $f \circ h = \text{id}$. The function g is called a **left inverse** for f because $f \circ g \neq \text{id}$, since $f \circ g$ is only a partial function.

The fact that f has a left inverse is *equivalent* to the assumption 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 $f(x) = f(y)$, we get

$$x = g(f(x)) = g(f(y)) = y .$$

Now we apply this trick to functions lifted into 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 $g \circ f = \text{id}$ to get

$$\begin{aligned} & \text{fmap}_L(g) \circ \text{fmap}_L(f) \\ \text{composition law for } L : &= \text{fmap}_L(g \circ f) \\ \text{use } g \circ f = \text{id} : &= \text{fmap}_L(\text{id}) \\ \text{identity law for } L : &= \text{id} . \end{aligned}$$

It follows that $\text{fmap}_L(g) \circ \text{fmap}_L(f) = \text{id}$, i.e. $\text{fmap}_L(g)$ is a left inverse for $\text{fmap}_L(f)$. Since $\text{fmap}_L(f)$ has a left inverse, it is injective.

7 Typeclasses and functions of types

7.1 Motivation and first examples

7.1.1 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
```

Computing the average value of a sequence of numbers can be written as

```
def avg(s: Seq[Double]): Double = s.sum / s.length
```

We would like to generalize the averaging function from `Double` to other numeric types, e.g.

```
def avg[T](s: Seq[T]): T = ???
```

But this is impossible because averaging works only for certain types T , not for arbitrary types T as would be implied by that type signature. 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. `BigDecimal` or `Double`).

Another example of a similar situation is a function with the type signature $A \times F^B \Rightarrow F^{A \times B}$,

```
def inject[F[_], A, B](a: A, f: F[B]): F[(A, B)] = f.map(b => (a, b))
```

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 look like? Consider the function

```
def f(x: Int): Int = x + 1
```

In this code, the syntax `x: Int` constrains the values of the argument `x` to be integers. It is a type error to apply `f` to a non-integer argument.

Using a similar syntax for *type parameters*, we would 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 types from that subset have certain properties that we require. A subset of types, together with the required properties that those types must satisfy, is called a **typeclass**. The syntax such as `[T: Fractional]` is a **typeclass constraint**.

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.

7.1.2 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 τ and a value s) and returns a value. We can also view `avg` as a function from a *type* τ 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 in Scala 2 code, we may write the type signature of `avg` as

```
def avg[T: Fractional]: Seq[T] => T
```

A type constructor such as `Seq[_]` can be seen as a **type-to-type** function (TTF) because it takes a type τ and returns a new type `Seq[T]`.

In principle, functions can map from values or from types to either values or types. The possibilities are shown in the following table:

	from value	from type
to value	VVF: def f(x:Int):Int	TVF: def point[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, “value-dependent types”). An example in Scala is:

```
val x = new { type T = Int }
val y: x.T = 123
```

In this example, `x.T` is a dependent type because it depends on the *value* `x`. For the value `x` defined in this code, the type `x.T` evaluates to the type `Int`.

We will not use dependent types (VTFs) in this chapter because typeclasses only require a combination of a TTF and a TVF.

7.1.3 Partial functions of types and values

The function `avg[T: Fractional]` is a TVF that is defined only on a subset of possible types τ . This is similar to a **partial function**, i.e. a function defined only for a subset of values. The function `avg[T]` can be applied only to certain types τ . 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 apply. For example, the PVVF `p` defined as

```
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.

It is better to use total functions rather than partial functions. The partial function `p` can be converted into a total function by changing its type to `Left[Int, String] => Int`. As another example, the operation `.head` on a `List` is unsafe but we can use the non-empty list type to guarantee at compile time that the first element exists:

```
def f(xs: NonEmptyList[Int]) = {
  val h = xs.head // Safe and checked at compile time.
}
```

In these cases, we achieve safety by refactoring the code to use more strictly constrained types.

Similarly, partial functions of types become safe to use if we impose the required typeclass constraints on the type parameters. Typeclasses can be viewed as a systematic way of safely managing partial type-to-value functions (PTVFs).

7.2 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 would we be able to reproduce that functionality?

7.2.1 Creating a partial function on types

The code needs to specify that the type parameter must belong to a certain set of allowed types. To simplify the task, we will assume that the allowed types are `BigDecimal` and `Double`. One example of such a constraint is given by unfunctors (see Section 6.1.6), which are type constructors whose type parameters are restricted to specific types. In this code,

```
sealed trait Frac[A] // Unfunctor.
final case class FracBD() extends Frac[BigDecimal]
final case class FracD() extends Frac[Double]
```

us to create a value of type, say, `Frac[String]` or `Frac[Boolean]`. Although the Scala compiler will not detect any errors in this code,

```
type T = Frac[String]
type U = Frac[Boolean]
```

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

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]    // Type error.
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 will be unable to use that value `y` in any further computations.

The type `FracA` 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 partial 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 ∈ {BigDecimal, Double}` is equivalent to the requirement that a value of type `FracT` should exist. So, we will implement the type constraint by including an *additional argument* of type `FracT` 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 will have to be passed to every application of `avg[T]`, we will be unable to use types `T` for which `Frac[T]` is void.

In this way, we implemented the typeclass constraint in 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 unfuncor. 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 implement 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 wrap the required functions. So, let us redefine `Frac` as a named tuple 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 this 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 the correct typeclass instance corresponding to the type `T`:

```
scala> avg(Seq(1.0, 2.0, 3.0), fracD) // It would be a type error to use fracBD 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 add another supported type `T`, we need to write one more line of code similar to `val fracD =`. To implement another function as a PTVF with the same type domain, we need to add an extra argument of type `Frac[T]`.

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
}
```

implementations require significant extra work:

- All calls to `func[T](args)` need to be rewritten as `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 each place in the code where we need to use them.

This extra work can be reduced (and sometimes avoided) by using Scala's "implicit value" feature.

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 our typeclass im-

7.2.2 Scala's implicit values

An **implicit value** is made automatically available as an “implicit argument” for any function that needs a value of that type. Scala supports implicit values using the syntax such as

```
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 “`implicit`” argument of type `Int`:

```
def f(a: String)(implicit n: Int) = s"$a with $n"

scala> f("xyz")
res0: String = xyz with 123
```

We still need to declare the arguments as “`implicit`” in the function’s type signature, and the implicit arguments must be in a separate argument group.

The simplest useful function using an implicit argument is the identity function whose argument is declared as `implicit`. In the standard Scala library, this function is called `implicitly`. Compare its code with the code for 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, we can apply `implicitly` and obtain 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 is specified by its type alone. The Scala compiler will not be able to set implicit arguments of functions automatically when more than one implicit value of the required type can be found. 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 this example, 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]
           ^
```

A compile-time error also occurs if more than one implicit value of the required type is found:

```
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]
           ^
```

7.2.3 Implementing typeclasses by making instances implicit

The main idea is to declare typeclass instance values as `implicit`. Typeclass instance arguments of functions are also declared as `implicit`. As a result, PTVFs can be applied without explicit typeclass instances, and will pass typeclass instances to other PTVFs automatically. This makes typeclasses easier to use because typeclass instances do not need to be handled passed around nearly as 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 a shorter syntax for typeclass instances: the function declaration

```
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]' defines some typeclass instances.
  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` these typeclass instance values explicitly:

```
scala> avg(Seq(1.0, 2.0, 3.0))
res0: Double = 2.0
```

7.2.4 Extension methods

In Scala, function applications can use three kinds of syntax:

- The “function” syntax: arguments are to the right of the function as in `plus(x, y)` or `plus(x)(y)`.
- The “method” syntax: the first argument is to the left of the function, the other arguments (if any) are to the right, as in `x.plus(y)` or `xs.foldLeft(0)(updater)`.
- The “infix method” syntax (only applies to functions with two *explicit* arguments): no dot character is used, as in `x plus y`, `xs map {x => x + 1}`, or `Set(1,2,3) contains 1`.

The last two syntax possibilities are often used when writing chains of function applications, such as `x plus y plus z` or `xs.map(f).filter(g)`, because the code becomes easier to read.

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, say

```
def func(x: X, y: Y): Z = { ... }
```

into an extension method on the type `x` with the syntax `x.func(y)` in addition to `func(x, y)`. To implement that, we need to define a new helper `class` that has a method `func`. The class’s constructor needs to 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 code, 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`.

As an example, let us implement the function `avg[T]` as an extension method on the type `Seq[T]`. Both the type parameter `T` and its typeclass constraint need to be used in the helper class:

```
implicit class AvgSyntax[T: Frac](xs: Seq[T]) {
  def average: T = avg(xs) // Use a different name to avoid ambiguity.
}
```

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[Seq]]).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 correct types.

This convenience comes at a cost: helper classes such as `AvgSyntax` often need to be explicitly imported in every scope where extension methods are used. If the helper class is defined in some library and Scala compiler cannot automatically find it, the programmer will have to look at the library’s source code to determine the name of the helper class to be imported.

7.2.5 Solved examples: Implementing typeclasses in practice

We will now see some practical examples of programming tasks implemented via typeclasses.

Example 7.2.5.1: metadata extractors Suppose that an application needs to work with data structures implemented in various third-party libraries. All those data structures are case classes that must contain certain metadata: “name” (a `String`) and “count” (a `Long` integer). However, the specific data structures declare the metadata differently, each in its own way:

```
final case class Data1(p: String, q: String, r: Long) // "name" = p, "count" = z
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 define 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 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 a typeclass constraint. 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)
```

To use the shorter syntax for the typeclass constraints, we need to replace `ti` by `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 partial type-to-value functions (PTVFs) with the type domain that contains the three types `Data1`, `Data2`, `Data3`. If we need to add support for more data types, say `Data4`, we will need to declare the appropriate 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 {
  implicit val data4 = HasMetadata[Data4](_.message, _.x.toLong)
}

scala> getName(Data4(1, "abc"))
res2: String = abc
```

For convenience, let us define 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 code such as

```
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 to notice that the new methods were added *without* any changes to the implementations of `Data1`, `Data2`, or `Data3`. Those implementations may reside in an external library whose code we do not need to modify. The typeclass pattern enables us to add those externally defined types to a type domain and to implement new PTVFs for them.

Example 7.2.5.2: observability counter A certain application needs to count the number of processed files and to expose this number to external services for observability. 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 of type `TestCounter` that has 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)
```

Having decided on this data structure, we can now implement the typeclass instances:

```
object Bumpable {
  implicit val b1 = Bumpable[Counter](c => c.inc())
  implicit val b2 = Bumpable[TestCounter](c => c.incr())
}
```

The implementation of `bump[C]` is now straightforward:

```
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 7.2.5.3: 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 the default value is 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 τ , 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 τ :

```
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
```

With the shorter typeclass syntax, the code is

```
def default[T: HasDefault]: T = implicitly[HasDefault[T]].value
```

What if we want to define 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 will), we need to write the typeclass instance as a `def`:

```
implicit def defaultList[A] = HasDefault[List[A]](List())
```

Another example of a typeclass instance with a type parameter is a default value for functions of type $A \Rightarrow A$:

```
implicit def defaultFunc[A] = HasDefault[A => A](identity)
```

Types that have default values are also called **pointed** types. However, this book defines the typeclass `Pointed` for pointed *functors* (Section 7.3.5) because they are more widely used than pointed types.

Example 7.2.5.4: mergeable data (semigroup) 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 “merger” operation can be defined as a function `combine[T]` taking two arguments of type τ and returning a new value of type τ . We will denote this operation by \oplus , for instance $x \oplus y$. In all examples we just saw (integers, strings, lists, etc.), that operation is associative:

$$(x \oplus y) \oplus z = x \oplus (y \oplus z) \quad . \quad (7.1)$$

This associativity law makes parentheses in the expression $x \oplus y \oplus z$ unnecessary.

A type τ 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 $T \times T \Rightarrow T$. So, we define the typeclass as a wrapper over this 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 (7.1), the result of $x \mid+| y \mid+| z$ does not depend on the choice of parentheses: $(x \mid+| y) \mid+| z == x \mid+| (y \mid+| z)$. 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 (7.1) should hold for each of the supported types; otherwise, 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 enforce the associativity law. It is the responsibility of the programmer to verify that the implementation of each typeclass instance is lawful.

The associativity law for integers is a standard arithmetic identity

$$(x + y) + z = x + (y + z) .$$

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 clear 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 recursive definition of the `concat` function, requires significant work (see Section 7.3.3).

Example 7.2.5.5: alternative semigroup implementation The implementations of the semigroup operation as concatenation for strings and addition for integers may appear to be “natural”. However, alternative implementations become useful in certain applications. As long as the associativity law holds, *any* function of type $T \times T \Rightarrow T$ 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 $x \oplus y \triangleq x$ (ignoring the value of y).

(b) For pair types $T \triangleq A \times B$, define the operation \oplus by

$$(a_1 \times b_1) \oplus (a_2 \times b_2) \triangleq a_1 \times b_2 .$$

(c) For $T \triangleq \text{String}$, define $x \oplus y$ as the longer of the strings x and y .

(d) For $T \triangleq S \Rightarrow S$ (the type S is fixed), define $x \oplus y \triangleq x \circ y$ (the forward function composition) or $x \oplus y \triangleq x \circ y$ (the backward function composition).

Solution (a) To verify the associativity law, use the definition $x \oplus y = x$ to compute

$$(x \oplus y) \oplus z = x \oplus z = x , \quad x \oplus (y \oplus z) = x .$$

So the associativity law holds: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ for any x, y, z .

It is clear that $x \oplus y \oplus z \oplus \dots = x$ 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 $x \oplus y = y$ gives an associative binary operation for a (different) semigroup.

(b) To verify the associativity law:

$$\begin{aligned} ((a_1 \times b_1) \oplus (a_2 \times b_2)) \oplus (a_3 \times b_3) &= (a_1 \times b_2) \oplus (a_3 \times b_3) = a_1 \times b_3 , \\ (a_1 \times b_1) \oplus ((a_2 \times b_2) \oplus (a_3 \times b_3)) &= (a_1 \times b_1) \oplus (a_2 \times b_3) = a_1 \times b_3 . \end{aligned}$$

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 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 $x \oplus y \oplus z = x \circ y \circ z$ or $x \oplus y \oplus z = z \circ y \circ x$, 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 7.2.5.6: mergeable data with default values (monoid) 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 and any other set does not change the other set.

A semigroup with such a default value is called a **monoid**. Formally, a type T is a monoid when it has an associative binary operation \oplus_T and a chosen default or “empty” value e^T such that

$$\text{for all } x^T : e \oplus_T x = x , \quad x \oplus_T e = x . \quad (7.2)$$

The laws (7.2) are called the **identity laws** of 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. 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 7.3.4 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.

Example 7.2.5.7: 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 $T \times T \Rightarrow T$ 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 short 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 then define this function as an `implicit def monoidOf...`, 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

$$\text{Semigroup}^T \triangleq T \times T \Rightarrow T , \quad \text{HasDefault}^T \triangleq T , \quad \text{Monoid}^T \triangleq (T \times T \Rightarrow T) \times T .$$

It is clear that

$$\text{Monoid}^T \cong \text{Semigroup}^T \times \text{HasDefault}^T .$$

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 continues to hold if the `Semigroup` typeclass instance was already lawful. However, the value stored in the `HasDefault` instance is not guaranteed to satisfy the identity laws (7.2) with respect to the `combine` operation stored in the `Semigroup` instance. The programmer must verify that those laws hold.

Are there alternative implementations of the `Monoid` typeclass instance given `Semigroup` and `HasDefault` instances? The function `monoidOf` needs to produce a value of type $(T \times T \Rightarrow T) \times T$ given values of type $T \times T \Rightarrow T$ and a value of type T :

```
monoidOf :  $(T \times T \Rightarrow T) \times T \Rightarrow (T \times T \Rightarrow T) \times T$  .
```

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 t1: Semigroup[T], t2: HasDefault[T]): Monoid[T] = {
  Monoid((x, y) => t1.combine(x, t1.combine(x, y)), t2.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.

7.2.6 Typeclasses for type constructors

An example of a function parameterized by a type constructor is

```
def inject[F[_]: Functor, A, B](a: A, f: F[B]): F[(A, B)] = ???
```

The intention is to require the type parameter `F` to be a functor.

We can use a suitable typeclass to implement this constraint. Since the type parameter `F` is itself a type constructor, the typeclass constructor will be of the form `Functor[F[_]]`.

What is the necessary information 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

$$\text{map} : \forall(A, B). F^A \Rightarrow (A \Rightarrow B) \Rightarrow F^B .$$

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:  $\forall(A, B). F[A] \Rightarrow (A \Rightarrow B) \Rightarrow F[B]$ ) // 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 $\forall(A, B). F^A \Rightarrow (A \Rightarrow B) \Rightarrow F^B$.

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 also convenient to define `.map` as an infix method on the `Functor` type constructors,

```
implicit class FunctorSyntax[F[_]: Functor, A](fa: F[A]) {
  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) }
```

To use this function, we write code such as

```
scala> inject("abc", Seq(1, 2, 3))
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 implementer's responsibility to verify that the laws hold.

One way of checking the laws is to use the `scalacheck` library¹ that automatically runs random tests for a given equation. 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. The code looks like this (the required `import` declarations are omitted for brevity):

```
def checkFunctorLaws[F[_], A, B, C](implicit ff: Functor[F],
  // The 'Arbitrary' typeclass is provided by scalacheck.
  fa: Arbitrary[F[A]], ab: Arbitrary[A => B], bc: Arbitrary[B => C]) = {
  // Identity law. The equation must hold for all 'fa'.
  forAll { (fa: F[A]) => fa.map(identity[A]) shouldEqual fa }
  // Composition law. The equation must hold for all 'fa', 'f', and 'g'.
  forAll { (f: A => B, g: B => C, fa: F[A]) =>
    fa.map(f).map(g) shouldEqual fa.map(f andThen g)
  }
}
// Check the laws for 'Seq[_]' using specific types instead of type parameters A, B, C.
checkFunctorLaws[Seq, Int, String, Double]
```

The `scalacheck` library will substitute a large number of random values into the given conditions, hoping to discover a set of values for which some condition fails. While a test of this kind is useful (it might find a bug), it does not provide a rigorous proof that the laws hold, because the laws are tested only in a finite number of cases and with type parameters set to specific types.

7.3 Deriving typeclass instances via structural analysis of types

In Chapter 6 we analyzed the structure of functors by checking whether the six standard type constructions can make new functors and contrafunctors 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 some cases, write code that creates the new typeclass instances automatically.

In the following sections, we will show how to use this approach for some simple typeclasses. Later chapters of this book will systematically apply structural analysis to all typeclasses (filterable functors, monads, applicative functors, and so on).

7.3.1 Extractors

Example 7.2.5.1 showed a typeclass that extracts metadata of fixed types from a value of type τ . The typeclass instance wraps 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

$$\text{HasMetadata}^T \triangleq (T \rightarrow \text{String}) \times (T \rightarrow \text{Long}) .$$

¹<https://www.scalacheck.org>

Construction	Type expression to implement	Results
The Unit type, or other fixed type C	Extractor^1 or Extractor^C	Extractor^Z
Product of extractor type A and any B	$\text{Extractor}^A \Rightarrow \text{Extractor}^{A \times B}$	one possibility
Co-product of two extractor types	$\text{Extractor}^A \times \text{Extractor}^B \Rightarrow \text{Extractor}^{A+B}$	one possibility
Function from or to another type C	$\text{Extractor}^A \Rightarrow \text{Extractor}^{A \Rightarrow C}$ or $\text{Extractor}^{C \Rightarrow A}$	$\text{Extractor}^{C \times (C \Rightarrow A)}$
Recursive types	$\text{Extractor}^T \Rightarrow \text{Extractor}^{S^T}$ where $T \triangleq S^T$	Extractor^T

Table 7.1: Type constructions for the `Extractor` typeclass.

Using the standard type equivalences (see Table 5.6), we find that this type is equivalent to $T \Rightarrow \text{String} \times \text{Long}$. So, let us denote $\text{String} \times \text{Long}$ by Z and consider a more general typeclass whose instances are values of type $T \Rightarrow Z$. We may call this typeclass a “ Z -extractor” since types T from its type domain permit us to extract values of type Z . With a fixed type Z , we denote the typeclass by

$$\text{Extractor}^T \triangleq T \Rightarrow Z .$$

```
final case class Extractor[Z, T](extract: T => Z)
```

To apply structural analysis, we check whether any of the standard type constructions produce new typeclass instances. The results are summarized in Table 7.1. 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 $\text{Extractor}^1 = 1 \Rightarrow Z$ 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 $\text{Extractor}^C = C \Rightarrow Z$ 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 $Z \Rightarrow Z$:

```
implicit def extractorZ[Z] = Extractor[Z, Z](identity)
```

Type parameters Creating a typeclass instance Extractor^A for an arbitrary type A means to compute $\forall A. A \Rightarrow Z$; 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

$$\forall(A, B, Z). \text{Extractor}^A \times \text{Extractor}^B \Rightarrow \text{Extractor}^{A \times B} = (A \Rightarrow Z) \times (B \Rightarrow Z) \Rightarrow A \times B \Rightarrow Z .$$

We can derive only two fully parametric implementations of this type signature:

$$f:A \Rightarrow Z \times g:B \Rightarrow Z \Rightarrow a \times b \Rightarrow f(a) \quad \text{and} \quad f:A \Rightarrow Z \times g:B \Rightarrow Z \Rightarrow a \times b \Rightarrow g(b) .$$

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 ,

$$\text{extractorPair} : \forall(A, B, Z). \text{Extractor}^A \Rightarrow \text{Extractor}^{A \times B} .$$

```
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 $P \times Q \times Z$ (where, as we know, the type Z 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

$$\forall(A, B, Z). \text{Extractor}^A \times \text{Extractor}^B \Rightarrow \text{Extractor}^{A+B} = (A \Rightarrow Z) \times (B \Rightarrow Z) \Rightarrow A + B \Rightarrow Z .$$

From a known type equivalence (Table 5.6), we have a unique implementation of this function:

$$\text{extractorEither} \triangleq f:A \Rightarrow Z \times g:B \Rightarrow Z \Rightarrow \begin{array}{c|c|c} & & Z \\ \hline & A & a \Rightarrow f(a) \\ \hline & B & b \Rightarrow g(b) \end{array} .$$

```
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.

Function types We need to investigate whether $C \Rightarrow A$ or $A \Rightarrow C$ 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

$$\text{Extractor}^A \Rightarrow \text{Extractor}^{A \Rightarrow C} \quad \text{or} \quad \text{Extractor}^A \Rightarrow \text{Extractor}^{C \Rightarrow A} .$$

Writing out the types, we find

$$(A \Rightarrow Z) \Rightarrow (A \Rightarrow C) \Rightarrow Z \quad \text{or} \quad (A \Rightarrow Z) \Rightarrow (C \Rightarrow A) \Rightarrow Z .$$

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 $C \times (C \Rightarrow A)$ and find that its `Extractor` instance, i.e. a value of type $C \times (C \Rightarrow A) \Rightarrow Z$, can be derived from a value of type $A \Rightarrow Z$ as

$$f:A \Rightarrow Z \Rightarrow c:C \times g:C \Rightarrow A \Rightarrow f(g(c)) .$$

```
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 $C \times (C \Rightarrow Z)$ and $D \times (D \Rightarrow Z \times P)$.

Another situation where an `Extractor` instance exists for the type $C \Rightarrow A$ 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 $C \times (C \Rightarrow A)$ and instead substitute the default value when necessary.

Recursive types We can apply type recursion to any of the non-recursive constructions that create a new type with an `Extractor` instance out of a previous extractor type. For clarity, let us use a *type constructor* denoted F^A for describing the produced new type out of a previous extractor type A .

For the product construction, we define $F_1^{B,A} \triangleq A \times B$. For the co-product construction, $F_2^{B,A} \triangleq B + A$ (where B must be also an extractor type). For the function construction, $F_3^{C,A} \triangleq C \times (C \Rightarrow A)$.

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 of such a recursive type is a composition of F_2 and F_1 defined by

$$T \triangleq F_2^{Z \times P, F_1^{T,Q}} = Z \times P + Q \times T . \quad (7.3)$$

We can visualize this recursive type as an “infinite disjunction”

$$\begin{aligned} T &\cong Z \times P + Q \times T \\ &\cong Z \times P + Q \times (Z \times P + Q \times (Z \times P + \dots)) \\ &\cong Z \times P \times (\mathbb{1} + Q + Q \times Q + \dots) \\ &\cong Z \times P \times \text{List}^Q . \end{aligned}$$

Since the resulting type is equivalent to $Z \times C$ where $C \triangleq P \times \text{List}^Q$, we find that the recursive equation (7.3) 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 . A polynomial in Z will have an `Extractor` instance only if it is of the form $Z \times C$ for some type C .

Recursive equations involving F_3 will produce new examples of `Extractor` types, such as

$$T \triangleq F_3^{C, F_2^{Z, F_1^{T,P}}} = C \times (C \Rightarrow Z + P \times T) . \quad (7.4)$$

Heuristically, this type can be seen as an “infinite” exponential-polynomial type expression

$$T = C \times (C \Rightarrow Z + P \times C \times (C \Rightarrow Z + P \times C \times (C \Rightarrow Z + \dots))) .$$

It remains to be seen whether types of this form are useful in applications.

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

$$T \triangleq S^T ,$$

where S^\bullet is a type constructor that is built up by composing F_1 , F_2 , and/or F_3 in some way. (The type constructor S^\bullet may use Z as well as other fixed types, which is not indicated for brevity.) For each of F_1 , F_2 , and/or F_3 , we implemented a function with type

$$\text{extractorF} : \text{Extractor}^A \Rightarrow \text{Extractor}^{F^A} .$$

Since S^\bullet is a composition of F_1 , F_2 , and/or F_3 , we are able to implement a function

$$\text{extractorS} : \text{Extractor}^A \Rightarrow \text{Extractor}^{S^A} .$$

The `Extractor` instance for the recursive type T is then defined recursively as

$$x : T \Rightarrow Z \triangleq \text{extractorS}(x) .$$

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. (7.4).

The type constructor S^\bullet is defined by

$$S^A \triangleq C \times (C \Rightarrow Z + P \times A) .$$

```
type S[A] = (C, C => Either[Z, (P, A)])
final case class TypeT(t: S[TypeT]) // Define the recursive type 'T'.
```

To implement the function of type `Extractor[T] => Extractor[S[T]]`, which is $(T \Rightarrow Z) \Rightarrow S^T \Rightarrow Z$, we begin with a typed hole

$$f:T \Rightarrow Z \Rightarrow s:C \times (C \Rightarrow Z + P \times T) \Rightarrow ???^Z .$$

To fill $???^Z$, we could apply $f:T \Rightarrow Z$ to some value of type T ; but the only value of type T can be obtained if we apply the function of type $C \Rightarrow Z + P \times T$ to the given value of type C . So we write

$$f:T \Rightarrow Z \Rightarrow c:C \times g:C \Rightarrow Z + P \times T \Rightarrow g(c) \triangleright ???^{Z+P \times T \Rightarrow Z} .$$

The new typed hole has a function type. We can write the code in matrix notation as

$$\text{extractorS} \triangleq f:T \Rightarrow Z \Rightarrow c:C \times g:C \Rightarrow Z + P \times T \Rightarrow g(c) \triangleright \begin{array}{c|c|c} & & Z \\ \hline Z & & \text{id} \\ \hline P \times T & p:P \times t:T \Rightarrow f(t) & \end{array} .$$

```
def extractorS[A](f: Extractor[A]): Extractor[S[A]] = Extractor[S[A]] { case (c, g) =>
  g(c) match {
    case Left(z)      => z
    case Right((p, t)) => f.extract(t)
  }
}
```

The recursive construction defines an `Extractor` instance for T by a recursive equation,

$$\text{extractorT} \triangleq \text{extractorS}(\text{extractorT}) . \quad (7.5)$$

```
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, perhaps surprisingly, that the recursive definition (7.5) terminates. How does that work? One would expect that a recursive definition of the form $x \triangleq f(x)$ generates an infinite loop, as it does in this example:

```
def f(x: Int): Int = x + 1
def x: Int = f(x)

scala> x
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. (7.5) in the form of an “expanded function”,

$$\text{extractorT} \triangleq t \Rightarrow \text{extractorS}(\text{extractorT})(t) .$$

```
def extractorT: Extractor[TypeT] = Extractor { case TypeT(t) => extractorS(extractorT).extract(t) }
```

Although the “expanded function” $t \Rightarrow f(t)$ is equivalent to just f , there is an important difference: using “expanded functions” will make recursive definitions valid and terminating.

As an example, consider a recursive function f defined via an equation $f \triangleq k(f)$, where k is a suitable function. If we write Scala code directly in this way (using functions as values), we will create an infinite loop:

```
def k(f: Int => Int): Int => Int = { x => if (x <= 0) 1 else 2 * f(x - 1) }
def f_bad: Int => Int = k(f_bad)      // This definition is invalid!

scala> f_bad(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

$$f \triangleq t \Rightarrow k(f)(t)$$

instead of $f \triangleq k(f)$, the code will become valid:

```
def f: Int => Int = { x => k(f)(x) }    // This defines  $f(n) = 2^n$  for  $n \geq 0$ .
                                                    

scala> f(4)
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 the type expression

$$K^{Z,P,Q,R,S} \triangleq Z \times P + Q \times (Q \Rightarrow Z + R \times (R \Rightarrow Z \times S)) \quad .$$

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]] = {           // Create Extractor for types:
  implicit val e1 = extractorPair[Z, Z, S]                                //  $Z \times S$ .
  implicit val e2 = extractorFunc[Z, (Z, S), R]                            //  $R \times (R \Rightarrow Z \times S)$ .
  implicit val e3 = extractorEither[Z, Z, (R, R => (Z, S))]               //  $Z + R \times (R \Rightarrow Z \times S)$ .
  implicit val e4 = extractorFunc[Z, Either[Z, (R, R => (Z, S))], Q]       //  $Q \times (Q \Rightarrow Z + R \times (R \Rightarrow Z \times S))$ .
  implicit val e5 = extractorPair[Z, Z, P]                                    //  $Z \times P$ .
  extractorEither[Z, (Z,P), (Q, Q => Either[Z, (R, R => (Z, S))])]     // Extractor for type K.
}
```

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”.

7.3.2 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 $A \times A \Rightarrow \text{Boolean}$ (where Boolean denotes the `Boolean` type). A function of that type is wrapped by typeclass instances of `Eq`. 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)
  }
  implicit val eqInt: Eq[Int] = Eq[Int](_ == _)
  implicit val eqString: Eq[String] = Eq[String](_ == _)
  // Other typeclass instances to be defined here.
}
```

We may, if desired, define an `Eq` typeclass instance for all types at once through the code

```
implicit def eqTypeA[A] = Eq[A](_ == _)
```

However, this will not always work as expected:

```
import Eq._

scala> ((n: Int) => n) === ((n: Int) => n)
res1: Boolean = false
```

The operations `==` and `===` do not seem to compare function values correctly. The equality comparison must satisfy the laws of identity, reflexivity and transitivity. The law of reflexivity states that $x = x$ for every x ; so the comparison $x === x$ should always return `true`. The example shown above clearly violates that law when we choose $x \triangleq n: \text{Int} \Rightarrow n$.

Let us perform structural analysis for the `Eq` typeclass, defining $\text{Eq}^A \triangleq A \times A \Rightarrow 2$. The results of the analysis will show which types can be meaningfully 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:

```
implicit 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, reflexivity, and transitivity laws hold for the new comparison operation if they hold for comparisons of A and B separately.

Co-products If A and B have equality comparisons, we can compare values of type $A + B$ while ensuring that a value of type $A + \emptyset$ is never equal to a value of type $\emptyset + B$:

```
implicit def eqEither[A: Eq, B: Eq] = Eq[Either[A, B]] {
  case (Left(a1), Left(a2))    => a1 === a2      // Compare a1 + \emptyset and a2 + \emptyset.
  case (Right(b1), Right(b2))   => b1 === b2      // Compare \emptyset + b1 and \emptyset + b2.
  case _                        => false        // a + \emptyset is never equal to \emptyset + b.
}
```

The laws hold for the new operation because the code makes sure that values of type `Either[A, B]` can be equal only when they are all of type `Left` or all of type `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`.

Since both products and co-products preserve equality, any polynomial type expression made of primitive types will also have an `Eq` typeclass instance.

Functions If A has equality comparisons, can we create an `Eq` instance for $R \Rightarrow A$ where R is some other type? This would be possible with a function of type

$$\forall A. (A \times A \Rightarrow 2) \Rightarrow (R \Rightarrow A) \times (R \Rightarrow A) \Rightarrow 2 \quad . \quad (7.6)$$

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 $R \Rightarrow A$ can be simplified to a polynomial type, e.g. $\mathbb{1} \Rightarrow A \cong A$ and $\mathbb{2} \Rightarrow A \cong A \times A$, 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 (7.6) 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. So, functions of type $R \Rightarrow A$ cannot have an `Eq` instance for a general type R . A similar argument shows that functions of type $A \Rightarrow R$ also cannot have a useful `Eq` instance.

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.

Consider a recursive polynomial type T defined using a polynomial functor S^\bullet ,

$$T \triangleq S^T \quad .$$

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

$$\text{eqS} : \text{Eq}^A \Rightarrow \text{Eq}^{S^A} \quad .$$

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:

$$\text{eqT} : \text{Eq}^T \quad , \quad \text{eqT} \triangleq \text{eqS}(\text{eqT}) \quad .$$

The recursive equation for `eqT` needs to be implemented as an expanded function,

$$\text{eqT} \triangleq t:T \times t:T \Rightarrow \text{eqS}(\text{eqT})(t \times t) \quad ,$$

and then, as we have seen in the previous section, the recursion will terminate.

As an example, let us define an `Eq` instance for the type T defined by $T \triangleq \text{Int} + T + \text{Int} \times T \times T$.

```
type S[A] = Either[Either[Int, A], (Int, (A, A))]
final case class T(s: S[T]) // Recursive equation T \triangleq Int + T + Int \times T \times 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]
  implicit val e2 = eqPair[A, A]
  implicit val e3 = eqPair[Int, (A, A)]
  eqEither[Either[Int, A], (Int, (A, A))]
```

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))))))

scala> t === t
res0: Boolean = true
```

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 tools such as the `kittens`², `magnolia`³, or `scalaz-deriving`⁴ libraries.

7.3.3 Semigroups

A type T has an instance of `Semigroup` when an associative binary operation of type $T \times T \Rightarrow T$ is available. Let us apply structural analysis to this typeclass.

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 7.2.5.4 and 7.2.5.5 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).

²<https://github.com/typelevel/kittens>

³<https://github.com/propensive/magnolia>

⁴<https://github.com/scalaz/scalaz-deriving>

Type parameters A semigroup instance parametric in type T means a value of type $\forall T. T \times T \Rightarrow T$. There are two implementations of this type signature: $a^T \times b^T \Rightarrow a$ and $a^T \times b^T \Rightarrow b$. Both provide an associative binary operation, as we showed in Example 7.2.5.5(a). In this way, any type T can be made into a “trivial” semigroup in one of these two ways. (“Trivial” semigroups are rarely useful.)

Products If types A and B are semigroups, the product $A \times B$ can be also given a `Semigroup` typeclass instance. To compute that instance means to implement a function with type

$$\text{semigroupPair} : \forall(A, B). \text{Semigroup}^A \times \text{Semigroup}^B \Rightarrow \text{Semigroup}^{A \times B} .$$

Writing out the type expressions, we get the type signature

$$\text{semigroupPair} : \forall(A, B). (A \times A \Rightarrow A) \times (B \times B \Rightarrow B) \Rightarrow (A \times B \times A \times B \Rightarrow A \times B) .$$

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

$$\text{semigroupPair} \triangleq f^{A \times A \Rightarrow A} \times g^{B \times B \Rightarrow B} \Rightarrow a_1^A \times b_1^B \times a_2^A \times b_2^B \Rightarrow ???^A \times ???^B .$$

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

$$f^{A \times A \Rightarrow A} \times g^{B \times B \Rightarrow B} \Rightarrow a_1^A \times b_1^B \times a_2^A \times b_2^B \Rightarrow f(a_1, a_2) \times g(b_1, b_2) .$$

This code defines a new binary operation $\oplus_{A \times B}$ via the previously given \oplus_A and \oplus_B as

$$(a_1^A \times b_1^B) \oplus_{A \times B} (a_2^A \times b_2^B) = (a_1 \oplus_A a_2) \times (b_1 \oplus_B b_2) . \quad (7.7)$$

```
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:

$$\begin{aligned} ((a_1 \times b_1) \oplus_{A \times B} (a_2 \times b_2)) \oplus_{A \times B} (a_3 \times b_3) &= ((a_1 \oplus_A a_2) \times (b_1 \oplus_B b_2)) \oplus_{A \times B} (a_3 \times b_3) \\ &= (a_1 \oplus_A a_2 \oplus_A a_3) \times (b_1 \oplus_B b_2 \oplus_B b_3) , \\ (a_1 \times b_1) \oplus_{A \times B} ((a_2 \times b_2) \oplus_{A \times B} (a_3 \times b_3)) &= (a_1 \times b_1) \oplus_{A \times B} ((a_2 \oplus_A a_3) \times (b_2 \oplus_B b_3)) \\ &= (a_1 \oplus_A a_2 \oplus_A a_3) \times (b_1 \oplus_B b_2 \oplus_B b_3) . \end{aligned}$$

Co-products To compute a `Semigroup` instance for the co-product $A + B$ of two semigroups, we need

$$\text{semigroupEither} : \forall(A, B). \text{Semigroup}^A \times \text{Semigroup}^B \Rightarrow \text{Semigroup}^{A+B} .$$

Writing out the type expressions, we get the type signature

$$\text{semigroupEither} : \forall(A, B). (A \times A \Rightarrow A) \times (B \times B \Rightarrow B) \Rightarrow (A + B) \times (A + B) \Rightarrow A + B .$$

Begin by writing a function with a typed hole:

$$\text{semigroupEither} \triangleq f^{A \times A \Rightarrow A} \times g^{B \times B \Rightarrow B} \Rightarrow c^{(A+B) \times (A+B)} \Rightarrow ???^{A+B} .$$

Transforming the type expression $(A + B) \times (A + B)$ into an equivalent disjunctive type,

$$(A + B) \times (A + B) \cong A \times A + A \times B + B \times A + B \times B ,$$

we can continue to write the function's code in matrix notation,

$$f:A \times A \Rightarrow A \times g:B \times B \Rightarrow B \Rightarrow \begin{array}{c|cc} & A & B \\ \hline A \times A & ???:A \times A \Rightarrow A & ???:A \times A \Rightarrow B \\ A \times B & ???:A \times B \Rightarrow A & ???:A \times B \Rightarrow B \\ B \times A & ???:B \times A \Rightarrow A & ???:B \times A \Rightarrow B \\ B \times B & ???:B \times B \Rightarrow A & ???:B \times B \Rightarrow B \end{array} .$$

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 $A \times A \Rightarrow A$ and $B \times B \Rightarrow B$, and so it is natural to fill them with f and g :

$$f:A \times A \Rightarrow A \times g:B \times B \Rightarrow B \Rightarrow \begin{array}{c|cc} f:A \times A \Rightarrow A & \emptyset \\ \hline ???:A \times B \Rightarrow A & ???:A \times B \Rightarrow B \\ ???:B \times A \Rightarrow A & ???:B \times A \Rightarrow B \\ \emptyset & g:B \times B \Rightarrow B \end{array} .$$

The remaining two rows can be filled in four different ways:

$$f \times g \Rightarrow \begin{array}{c|cc} f:A \times A \Rightarrow A & \emptyset \\ a:A \times b:B \Rightarrow a & \emptyset \\ b:B \times a:A \Rightarrow a & \emptyset \\ \emptyset & g:B \times B \Rightarrow B \end{array}, \quad f \times g \Rightarrow \begin{array}{c|cc} f & \emptyset \\ \emptyset & a:A \times b:B \Rightarrow b \\ \emptyset & b:B \times a:A \Rightarrow b \\ \emptyset & g \end{array},$$

$$f \times g \Rightarrow \begin{array}{c|cc} f:A \times A \Rightarrow A & \emptyset \\ \emptyset & a:A \times b:B \Rightarrow b \\ b:B \times a:A \Rightarrow a & \emptyset \\ \emptyset & g:B \times B \Rightarrow B \end{array}, \quad f \times g \Rightarrow \begin{array}{c|cc} f & \emptyset \\ a:A \times b:B \Rightarrow a & \emptyset \\ \emptyset & b:B \times a:A \Rightarrow b \\ \emptyset & g \end{array} .$$

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 semigroupEither1[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 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 last" - discard x in x ⊕ y.
    case (Right(b), Left(a))   => a
}

```

The different choices of the binary operation can be described as:

1. (“Take A”) Discard all data of type B : $(a:A + \emptyset) \oplus (\emptyset + b:B) = a$ and $(\emptyset + b:B) \oplus (a:A + \emptyset) = a$.
2. (“Take B”) Discard all data of type A : $(a:A + \emptyset) \oplus (\emptyset + b:B) = b$ and $(\emptyset + b:B) \oplus (a:A + \emptyset) = b$.
3. (“Take first”) $x \oplus y$ discards y : $(a:A + \emptyset) \oplus (\emptyset + b:B) = a$ and $(\emptyset + b:B) \oplus (a:A + \emptyset) = b$.
4. (“Take last”) $x \oplus y$ discards x : $(a:A + \emptyset) \oplus (\emptyset + b:B) = b$ and $(\emptyset + b:B) \oplus (a:A + \emptyset) = a$.

Does the semigroup law (7.1) 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 the values of expressions 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 eight possible combinations of `Left` and `Right` types. Each of them needs to be checked against each of the four implementations of \oplus . Instead of doing 32 separate derivations, let us perform a quicker and intuitively clearer proof via case by case reasoning in Scala syntax.

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))
```

The same argument applies to the second implementation (“take B”). Thus, both implementations are associative.

The implementation “take first” will select the first value whenever the types are mixed. Therefore

```
Left(x) |+| Left(y) |+| Right(z) == Left(x |+| y) == Left(x) |+| (Left(y) |+| Right(z))
Left(x) |+| Right(y) |+| z == Left(x) == Left(x) |+| (Right(y) |+| z)      // Regardless of z.
Right(x) |+| Left(y) |+| z == Right(x) == Right(x) |+| (Left(y) |+| z)      // Regardless of z.
Right(x) |+| Right(y) |+| Left(z) == Right(x |+| y) == Right(x) |+| (Right(y) |+| Left(z))
```

In all cases, the results do not depend on the order of parentheses.

The same argument applies to the “take last” implementation.

Functions If A is a semigroup and E is any fixed type, are the types $A \Rightarrow E$ and/or $E \Rightarrow A$ semigroups? To create a `Semigroup` instance for $E \Rightarrow A$ means to implement the type signature

$$\text{Semigroup}^A \Rightarrow \text{Semigroup}^{E \Rightarrow A} = (A \times A \Rightarrow A) \Rightarrow (E \Rightarrow A) \times (E \Rightarrow A) \Rightarrow E \Rightarrow A .$$

The implementation that preserves information is

$$\text{semigroupFunc} \triangleq f:A \times A \Rightarrow A \Rightarrow g_1^{E \Rightarrow A} \times g_2^{E \Rightarrow A} \Rightarrow e:E \Rightarrow f(g_1(e), g_2(e)) .$$

This defines the new \oplus operation by $g_1 \oplus g_2 \triangleq e \Rightarrow g_1(e) \oplus_A g_2(e)$.

```
def semigroupFunc[E, A: Semigroup] = Semigroup[E => A] { case (g1, g2) => e => g1(e) |+| g2(e) }
```

The type $A \Rightarrow E$ only allows semigroup operations that discard the left or the right element: the type signature $f:A \times A \Rightarrow A \Rightarrow h_1:A \Rightarrow E \times h_2:A \Rightarrow E \Rightarrow ???^A \Rightarrow E$ can be implemented only by discarding f and one of h_1 or h_2 . Either choice makes $A \Rightarrow E$ 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 existing semigroups is again a semigroup.

Recursive types A type T defined by a recursive type equation $T \triangleq S^T$ 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

$$\text{semigroupS} : \text{Semigroup}^A \Rightarrow \text{Semigroup}^{S^A} .$$

Then the semigroup instance for T is defined recursively as

$$\text{semigroupT} \triangleq \text{semigroupS}(\text{semigroupT}) .$$

The recursive definition will terminate as long as we implement it in code as an expanded function.

Summary All type constructions preserve semigroups, so any type expression whatsoever can have a `Semigroup` instance. Since the typeclass instances have several inequivalent implementations, automatic derivation of `Semigroup` instances is not often useful.

7.3.4 Monoids

Since a monoid is a semigroup with a default value, a `Monoid` instance is a value of type

$$\text{Monoid}^A \triangleq (A \times A \Rightarrow A) \times A .$$

For the binary operation $A \times A \Rightarrow A$, 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.

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).

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

$$\text{monoidPair} : \forall(A, B). \text{Monoid}^A \times \text{Monoid}^B \Rightarrow \text{Monoid}^{A \times B} .$$

The empty value for the monoid $A \times B$ is the pair of empty values from the monoids A and B ,

$$e_{A \times B} \triangleq e_A \times e_B .$$

The new binary operation is defined by Eq. (7.7) 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 :

$$(a_1 \times b_1) \oplus (e_A \times e_B) = (a_1 \oplus_A e_A) \times (b_1 \oplus_B e_B) = a_1 \times b_1 , \\ (e_A \times e_B) \oplus (a_2 \times b_2) = (e_A \oplus_A a_2) \times (e_B \oplus_B b_2) = a_2 \times b_2 .$$

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:

$$e_{A+B} \triangleq e_A + \mathbb{0}^{:B} \quad \text{or} \quad e_{A+B} \triangleq \mathbb{0}^{:A} + e_B \quad .$$

It remains to see which of the eight combinations will satisfy the monoid identity laws,

$$\begin{aligned} (a + \mathbb{0}) \oplus e_{A+B} &= a + \mathbb{0} \quad , \quad e_{A+B} \oplus (a + \mathbb{0}) = a + \mathbb{0} \quad , \\ (\mathbb{0} + b) \oplus e_{A+B} &= \mathbb{0} + b \quad , \quad e_{A+B} \oplus (\mathbb{0} + b) = \mathbb{0} + b \quad . \end{aligned}$$

First choose $e_{A+B} = e_A + \mathbb{0}$; the reasoning for the other case will be quite similar. The first line above,

$$(a + \mathbb{0}) \oplus (e_A + \mathbb{0}) = a + \mathbb{0} \quad , \quad (e_A + \mathbb{0}) \oplus (a + \mathbb{0}) = a + \mathbb{0} \quad ,$$

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 ”:

$$(a^{:A} + \mathbb{0}) \oplus (\mathbb{0} + b^{:B}) = b \quad \text{and} \quad (\mathbb{0} + b^{:B}) \oplus (a^{:A} + \mathbb{0}) = b \quad .$$

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 $e_{A+B} \triangleq \mathbb{0}^{:A} + e_B$ forces us to choose the version “take A ” of the \oplus operation.

Function types The semigroup construction for function types works for monoids. Exercise 7.4.2.3 is to show that for any type R and any monoid A , the function type $R \Rightarrow A$ is a lawful monoid.

Recursive types Can we define a `Monoid` instance for a type T defined by $T \triangleq S^T$, 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 when A is a monoid:

$$\text{monoidS} : \text{Monoid}^A \Rightarrow \text{Monoid}^{S^A} \quad .$$

A monoid instance for T is then defined recursively by

$$\text{monoidT} \triangleq \text{monoidS}(\text{monoidT}) \quad .$$

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 $(A \times A \Rightarrow A) \times A$. To obtain a working implementation of `monoidT`, we need to rewrite that type into an equivalent function type,

$$\text{Monoid}^A = (A \times A \Rightarrow A) \times A \cong (A \times A \Rightarrow A) \times (\mathbb{1} \Rightarrow A) \cong (\mathbb{1} + A \times A \Rightarrow A) \quad ,$$

where we used the known type equivalences $A \cong \mathbb{1} \Rightarrow A$ and $(A \Rightarrow C) \times (B \Rightarrow C) \cong A + B \Rightarrow C$.

```
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^A defined as

$$S^A \triangleq (\text{Int} + A) \times \text{Int} + \text{String} \times (A \Rightarrow (A \Rightarrow \text{Int}) \Rightarrow A) .$$

```
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 $T \triangleq S^T$ 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 all methods of the `trait` are declared as `def` (not as a `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. Figure 7.1 shows the full code.

Summary A `Monoid` instance can be implemented for *any* type expression built from primitive types, products, co-products, and function types. For recursive types T defined by a type equation $T \triangleq S^T$, a `Monoid` instance can be implemented for any exponential-polynomial type constructor S^A .

7.3.5 Pointed functors: motivation and laws

Section 7.2.6 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 two examples of typeclasses (pointed and co-pointed functors) that extend `Functor` with further methods. Chapter 6 performed a structural analysis of functors, which we will now 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

```

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 will have 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 the string "abcabc".
res0: T = T(Right((abcabc,<function1>)))

```

Figure 7.1: Using a `trait` for implementing the `Monoid` typeclass and a recursive instance.

7 Typeclasses and functions of types

```
def pure[A]: A => F[A] = ???
```

The code notation for this function is pu_F , and the type signature is written as $\text{pu}_F : \forall A. A \Rightarrow F^A$.

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 computed.
```

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 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 `Pointed` 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 a `.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 an arbitrary function $f:A \Rightarrow B$ applied to the wrapped value via `map`:

```
pure(x).map(f) // pu_F(x) ∘ f^F
```

We now expect that the result should be the same as a wrapped $f(x)$. This expectation can be formulated as a law,

$$\text{pu}_F(x) \triangleright f^F = \text{pu}_F(f(x)) .$$

```
pure(x).map(f) = pure(f(x))
```

This law is called the **naturality law** of `pure`; it must hold for any $f:A \Rightarrow B$. Using the \triangleright -notation, we reformulate this law as $x \triangleright \text{pu}_F \triangleright f^F = x \triangleright f \triangleright \text{pu}_F$ or equivalently as $x \triangleright \text{pu}_F ; f^F = x \triangleright f ; \text{pu}_F$. Since both sides of the law are functions applied to an arbitrary x , we can omit x and write

$$\text{pu}_F ; f^F = f ; \text{pu}_F . \quad (7.8)$$

$$\begin{array}{ccc} A & \xrightarrow{\text{pu}_F} & F^A \\ f \downarrow & & \downarrow \text{fmap}_F(f) \\ B & \xrightarrow{\text{pu}_F} & F^B \end{array}$$

This motivates the rigorous definition: A functor F^\bullet is **pointed** if there exists a fully parametric function $\text{pu}_T : \forall A. A \Rightarrow F^A$ satisfying the naturality law (7.8) for any function $f:A \Rightarrow B$.

It turns out that we can avoid checking the naturality law for 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 (7.8) are functions of type $A \Rightarrow F^B$. The trick is to set $A = \mathbb{1}$ and $f : \mathbb{1} \Rightarrow B \triangleq _ \Rightarrow b$, where $b : B$ is any fixed value. Both sides of the naturality law can then be applied to the unit value 1 and must evaluate to the same result,

$$1 \triangleright \text{pu}_F \triangleright (_ \Rightarrow b)^{\uparrow F} = 1 \triangleright f \triangleright \text{pu}_F \quad .$$

Since $1 \triangleright f = f(1) = b$, we find

$$\text{pu}_F(1) \triangleright (_ \Rightarrow b)^{\uparrow F} = \text{pu}_F(b) \quad . \quad (7.9)$$

The naturality law (7.8) applies to all types A, B and to any function $f : A \Rightarrow B$. Thus, Eq. (7.9) must apply to an arbitrary value $b : B$ for any type B . That formula expresses the entire 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.
```

So far we have seen that if pu_F satisfies the naturality law then the single “wrapped unit” value of that function, $\text{pu}_F(1)$, is sufficient to recover the entire function as

$$\text{pu}_F = b \Rightarrow \text{pu}_F(1) \triangleright (_ \Rightarrow b)^{\uparrow F}$$

Now, if someone gives us just the “wrapped unit” value (denoted wu_F) of type $F^{\mathbb{1}}$, we can define a new function pu_F by

$$\text{pu}_F^{A \Rightarrow F^A} \triangleq x : A \Rightarrow \text{wu}_F \triangleright (_ \Rightarrow x)^{\uparrow F} \quad . \quad (7.10)$$

```
def pure[A](x: A): F[A] = wu.map { _ => x }
```

Does this function satisfy the naturality law with respect to an arbitrary $f : A \Rightarrow B$? It does:

$$\begin{aligned} \text{expect to equal } x \triangleright f \circ \text{pu}_F : & x \triangleright \text{pu}_F \circ f^{\uparrow F} \\ \text{definition of } \text{pu}_F : & = \text{wu}_F \triangleright (_ \Rightarrow x)^{\uparrow F} \circ f^{\uparrow F} \\ \text{functor composition law of } F : & = \text{wu}_F \triangleright ((_ \Rightarrow x) \circ f)^{\uparrow F} \\ \text{compute function composition} : & = \text{wu}_F \triangleright (_ \Rightarrow f(x))^{\uparrow F} \\ \text{definition of } \text{pu}_F : & = \text{pu}_F(f(x)) \\ \triangleright\text{-notation} : & = x \triangleright f \triangleright \text{pu}_F = x \triangleright f \circ \text{pu}_F \quad . \end{aligned}$$

This new function will satisfy $\text{pu}_F(1) = \text{wu}_F$ since

$$\begin{aligned} \triangleright\text{-notation} : & \text{pu}_F(1) = 1 \triangleright \text{pu}_F \\ \text{definition of } \text{pu}_F \text{ via } \text{wu}_F : & = \text{wu}_F \triangleright (_ \Rightarrow 1)^{\uparrow F} \\ \text{the function } (_ \Rightarrow 1) \text{ is the identity function } \text{id}^{\mathbb{1} \Rightarrow \mathbb{1}} : & = \text{wu}_F \triangleright \text{id}^{\uparrow F} \\ \text{functor identity law of } F : & = \text{wu}_F \triangleright \text{id} = \text{wu}_F \quad . \end{aligned}$$

To summarize our results: for any functor F ,

- If F is a lawful pointed functor, its `pure` method satisfies Eq. (7.10) where $\text{wu}_F \triangleq \text{pu}_F(1)$ 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. (7.10) 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 function pu_F and the value wu_F are **computationally equivalent**: each one can be converted into the other and back, with no loss of information. We can define a pointed functor equivalently as a functor with a chosen value wu_F of type F^\perp . When reasoning about pointed functors, it is simpler to use the definition via the “wrapped unit” wu because it has a simple type and no laws. For practical programming, however, the `pure` method is preferable.

7.3.6 Pointed functors: structural analysis

To perform structural analysis, we begin with the constructions from Chapter 6 that are known to build lawful functors and impose an additional requirement that a “wrapped unit” value $\text{wu}_F : F^\perp$ should exist. As we have seen in the previous section, no additional laws need to be checked.

```
final case class Pointed[F[_]](wu: F[Unit])
def pure[F[_]: Pointed : Functor, A](a: A): F[A] = implicitly[Pointed[F]].wu.map(_ => a)
```

Fixed types A constant functor $\text{Const}^{Z,\bullet}$ is defined as $\text{Const}^{Z,A} \triangleq Z$, 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 $\mathbb{1} + \mathbb{0}^U$. In that case, we set $\text{wu}_{\text{Const}^{Z,\bullet}} = \mathbb{1} + \mathbb{0}^U$.

```
type Const[Z, A] = Z
def pointedOption[U]: Pointed[Const[Option[U], ?]] = Pointed(None: Const[Option[U], Unit])
```

Type parameters The identity functor $\text{Id}^A \triangleq A$ is pointed since $\text{Id}^\perp = \mathbb{1}$, and we can set $\text{wu}_{\text{Id}} = \mathbb{1}$.

```
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 . If both F and G are pointed, we can apply F ’s `pure` method to $\text{wu}_G : G^\perp$ and obtain a value of type F^G .

```
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 7.3.8 below). A pointed contrafunctor has a “wrapped unit” value of type F^\perp , which can be transformed into F^A for any type A by using `contramap` with a constant function $A \Rightarrow \mathbb{1}$:

```
def purec[F[_]: Pointed : Contrafunctor, A]: F[A] = implicitly[Pointed[F]].wu.cmap(_ => ())
```

In this way, we can create a value of type F^G . The contrafunctor G does not need to be pointed.

```
def pointedCFOG[F[_]: Pointed : Contrafunctor, G[_]: Pointed]: Pointed[Lambda[X => F[G[X]]]] =
  Pointed[Lambda[X => F[G[X]]]](purec[F, G[Unit]])
```

Products If F and G are two pointed functors, is the functor product $L^A \triangleq F^A \times G^A$ a pointed functor? We need to produce a value $\text{wu}_L : F^\perp \times G^\perp$, and we have values $\text{wu}_F : F^\perp$ and $\text{wu}_G : G^\perp$. 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 $L^A \triangleq F^A + G^A$ a pointed functor? We need to produce a value $\text{wu}_L : F^\perp + G^\perp$, and we have values $\text{wu}_F : F^\perp$ and $\text{wu}_G : G^\perp$. There are two solutions: $\text{wu}_L = \text{wu}_F + \mathbb{0}^{G^\perp}$ and $\text{wu}_L = \mathbb{0}^{F^\perp} + \text{wu}_G$. Both choices define L^\bullet as a

pointed functor.

It is sufficient for only F^\bullet to be a pointed functor; $wu_L \triangleq wu_F + \mathbb{0}^{G^\dagger}$ then provides 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))
```

In general, when the type F^\dagger has several distinct values, the choice of wu_F is application-dependent. For example, if $F^A = \text{List}^A$, the type List^\dagger has values such as an empty list `List[Unit]()`, a list with a single `Unit` value, a list with two `Unit` values, etc. All of these choices will give a `Pointed` typeclass instance to the `List` functor. It is up to the programmer to choose the `Pointed` instance that will be useful for a specific application. In the case of `List`, the standard choice `wu == List(())` and correspondingly `pure(x) = List(x)` is motivated by the usage of the `List` type constructor to represent several possibilities, e.g. in a search problem; then the “pure” list represents the situation with only one possibility.

Function types If C is any contrafunctor and F is a pointed functor, the exponential functor $L^A \triangleq C^A \Rightarrow F^A$ will be pointed if we produce a value $wu_L : C^\dagger \Rightarrow F^\dagger$. We already have a value $wu_F : F^\dagger$, and there is no way we could have used a value of type C^\dagger because we know nothing about the contrafunctor C . So, we have to set $wu_L \triangleq (_ \Rightarrow wu_F)$, i.e. wu_L is a function that ignores its argument and always returns 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 assumes a bifunctor $S^{\bullet,\bullet}$ and defines a recursive functor F^\bullet via the type equation $F^A \triangleq S^{A,F^A}$. The functor F^A will be pointed if we can compute a value wu_F of type F^\dagger . The type F^\dagger is a recursive type defined via the type equation $F^\dagger \triangleq S^{\dagger,F^\dagger}$. 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 7.5.1 below). For *polynomial* bifunctors S (which is the most often used kind of bifunctors), a necessary and sufficient condition is that the type $S^{1,0}$ should be non-void. If we can create a value of type $S^{1,0}$, constructions shown in Section 7.5.1 will guarantee that we can also create a value of type F^\dagger and so the recursive functor F will be pointed.

As an example, consider the polynomial bifunctor $S^{A,R} \triangleq A + A \times R$. The corresponding recursive functor $F^A \triangleq S^{A,F^A} = A + A \times F^A$ 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”

$$F^A = A + A \times (A + A \times (A + \dots)) \cong A + A \times A + A \times A \times A + \dots .$$

Since the type $S^{1,0} = \mathbb{1} + \mathbb{1} \times \mathbb{0} \cong \mathbb{1}$ is non-void, the necessary and sufficient condition holds, so we expect that the recursive construction will work. The type F^\dagger is defined by

$$F^\dagger \triangleq S^{\dagger,F^\dagger} = \mathbb{1} + \mathbb{1} \times F^\dagger \cong \mathbb{1} + F^\dagger .$$

This type can be (non-rigorously) viewed as an “infinite disjunction”

$$F^\dagger = \mathbb{1} + \mathbb{1} + \mathbb{1} + \dots .$$

It is clear that a value of that type can be computed, for example, as

$$wu_F = \mathbb{1} + \mathbb{0} + \mathbb{0} + \dots \cong \mathbb{1} + \mathbb{0}^{F^\dagger} .$$

In Scala, this is `Left()`. So, a `Pointed` typeclass instance for F is implemented by

```
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 $\text{wu}_F = 1 + \mathbb{0} + \mathbb{0} + \dots$ into

$$a + \mathbb{0} + \mathbb{0} + \dots : A + A \times A + A \times A \times A + \dots \quad .$$

So, the `pure` method of F creates a non-empty list with a single element $a:A$.

7.3.7 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”:

$$\text{ex} : \forall A. F^A \Rightarrow A \quad .$$

```
def extract[F[_], A]: F[A] => A = ???
```

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` operation must be a fully parametric function that obeys the naturality law,

$$\text{ex}_F ; f = f^{\uparrow F} ; \text{ex}_F \quad . \quad (7.11)$$

$$\begin{array}{ccc} F^A & \xrightarrow{\text{ex}_F} & A \\ \downarrow \text{fmap}_F(f) & & \downarrow f \\ F^B & \xrightarrow{\text{ex}_F} & B \end{array}$$

The naturality law formulates our expectation that the extractor function somehow “selects” a value of type A among all the values possibly wrapped by F^A , and the “selection” works independently of the values. If all wrapped values are transformed by a function f into 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 selected a value of type A and then transformed it with f .

Both sides of the law (7.11) are functions of type $F^A \Rightarrow B$. We saw in the previous section that the `pure` method of the `Pointed` typeclass is computationally equivalent to a single chosen value of type $F^\mathbb{1}$ when F is a functor. For co-pointed functors, there is no simpler form of the `extract` method. If we set $A = \mathbb{1}$ and $f: \mathbb{1} \Rightarrow B \triangleq (1 \Rightarrow b)$ in the naturality law, both sides will become functions of type $F^\mathbb{1} \Rightarrow B$. Now, 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 a pointed functor, we *will* have a chosen value $\text{wu}_F : F^\mathbb{1}$ to which we may then apply the naturality law (7.11) and obtain

$$\text{wu}_F \triangleright \text{ex}_F ; f = \text{wu}_F \triangleright f^{\uparrow F} ; \text{ex}_F \quad .$$

Calculating both sides separately, we find

$$\begin{aligned} \text{wu}_F \triangleright \text{ex}_F \triangleright f &= \text{wu}_F \triangleright \text{ex}_F \triangleright f = 1 \triangleright f = b \quad . \\ \text{wu}_F \triangleright f^{\uparrow F} ; \text{ex}_F &= b \triangleright \text{pu}_F ; \text{ex}_F \quad . \end{aligned}$$

The result is $b = b \triangleright \text{pu}_F ; \text{ex}_F$. This can hold for all $b:B$ only if

$$\text{pu}_F ; \text{ex}_F = \text{id} \quad .$$

This additional **compatibility law** is a consequence of naturality laws if the functor F is pointed and co-pointed at the same time.

Fixed types A constant functor $\text{Const}^{Z,A} \triangleq Z$ is *not* co-pointed because we cannot implement $\forall A. Z \Rightarrow A$ (a value of an arbitrary type A cannot be computed from values of a fixed type Z).

Type parameters The identity functor $\text{Id}^A \triangleq A$ is co-pointed with $\text{ex} \triangleq \text{id}^{A \Rightarrow A}$. An identity function will always satisfy any naturality law.

Composition of two co-pointed functors F, G is co-pointed:

$$\text{ex}_{F \circ G} \triangleq h^{F^G} \Rightarrow \text{ex}_G(\text{ex}_F(h)) \quad \text{or equivalently} \quad \text{ex}_{F \circ G} = \text{ex}_F ; \text{ex}_G .$$

The naturality law holds for $\text{ex}_{F \circ G}$ because

$$\begin{aligned} \text{expect to equal } \text{ex}_{F \circ G} ; f &: f^{\uparrow F \circ G} ; \text{ex}_{F \circ G} \\ \text{definition of } f^{\uparrow F \circ G}, \text{ see Eq. (6.17)} &: = (f^{\uparrow G})^{\uparrow F} ; \text{ex}_F ; \text{ex}_G \\ \text{naturality law for } \text{ex}_F &: = \text{ex}_F ; f^{\uparrow G} ; \text{ex}_G \\ \text{naturality law for } \text{ex}_G &: = \text{ex}_F ; \text{ex}_G ; f = \text{ex}_{F \circ G} ; f . \end{aligned}$$

Products If functors F and G are co-pointed, we can implement a function of type $F^A \times G^A \Rightarrow A$ in two different ways: by discarding F^A or by discarding G^A . With either choice, the functor product $F^\bullet \times G^\bullet$ is made into a co-pointed functor. For instance, if we choose to discard G^A , the code for the `extract` method will be

$$\text{ex}_{F \times G} \triangleq f^{F^A} \times g^{G^A} \Rightarrow \text{ex}_F(f) = \nabla_1 ; \text{ex}_F ,$$

where we used the pair projection function $\nabla_1 \triangleq (a \times b \Rightarrow a)$. The following calculation verifies the naturality law 7.11 for this definition of $\text{ex}_{F \times G}$:

$$\begin{aligned} \text{expect to equal } \text{ex}_{F \times G} ; f &: f^{\uparrow F \times G} ; \text{ex}_{F \times G} \\ \text{definition of } f^{\uparrow F \times G}, \text{ see Eq. (6.13)} &: = (f^{\uparrow F} \boxtimes f^{\uparrow G}) ; \text{ex}_{F \times G} \\ \text{definition of } \text{ex}_{F \times G} &: = (f^{\uparrow F} \boxtimes f^{\uparrow G}) ; \nabla_1 ; \text{ex}_F \\ \text{use the property } (p \boxtimes q) ; \nabla_1 &= \nabla_1 ; p : = \nabla_1 ; f^{\uparrow F} ; \text{ex}_F \\ \text{naturality law of } \text{ex}_F &: = \nabla_1 ; \text{ex}_F ; f \\ \text{definition of } \text{ex}_{F \times G} &: = \text{ex}_{F \times G} ; f . \end{aligned}$$

Co-products For co-pointed functors F and G , the implementation of $\text{ex}_{F+G} : F^A + G^A \Rightarrow A$ is

$$\text{ex}_{F+G} \triangleq \left| \begin{array}{c|c} & A \\ \hline F^A & \text{ex}_F \\ G^A & \text{ex}_G \end{array} \right| .$$

This is the only possible implementation for this type signature, given that we have functions ex_F and ex_G . To verify that ex_{F+G} satisfies the naturality law, we compute

$$\begin{aligned} \text{expect to equal } \text{ex}_{F+G} ; f &: f^{\uparrow F+G} ; \text{ex}_{F+G} \\ \text{definition of } f^{\uparrow F+G}, \text{ see Eq. (6.14)} &: = \left\| \begin{array}{cc} f^{\uparrow F} & \mathbf{0} \\ \mathbf{0} & f^{\uparrow G} \end{array} \right\| ; \left\| \begin{array}{c} \text{ex}_F \\ \text{ex}_G \end{array} \right\| \\ \text{matrix function composition} &: = \left\| \begin{array}{c} f^{\uparrow F} ; \text{ex}_F \\ f^{\uparrow G} ; \text{ex}_G \end{array} \right\| \\ \text{naturality laws for } \text{ex}_F \text{ and } \text{ex}_G &: = \left\| \begin{array}{c} \text{ex}_F ; f \\ \text{ex}_G ; f \end{array} \right\| = \left\| \begin{array}{c} \text{ex}_F \\ \text{ex}_G \end{array} \right\| ; f = \text{ex}_{F+G} ; f . \end{aligned}$$

Function types An exponential functor of the form $L^A \triangleq C^A \Rightarrow P^A$ (where C is a contrafunctor and P is a functor) will be co-pointed if we can implement a function of type $\forall A. (C^A \Rightarrow P^A) \Rightarrow A$,

$$\text{ex}_L \triangleq h : C^A \Rightarrow P^A \Rightarrow ???^A \quad .$$

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 `pure` of type $\forall A. C^A$ a **pointed contrafunctor**. Assuming that C is pointed and denoting its `pure` method by pu_C , we can thus compute a value of type P^A as $h(\text{pu}_C)$. To extract A from P^A , we need to assume additionally that P is co-pointed and use its method $\text{ex}_P : P^A \Rightarrow A$. Finally we have

$$\text{ex}_L \triangleq h : C^A \Rightarrow P^A \Rightarrow \text{ex}_P(h(\text{pu}_C)) \quad \text{or equivalently} \quad h : C^A \Rightarrow P^A \triangleright \text{ex}_L = \text{pu}_C \triangleright h \triangleright \text{ex}_P \quad . \quad (7.12)$$

To verify the naturality law, we apply both sides to an arbitrary $h : C^A \Rightarrow P^A$ and compute

$$\begin{aligned} \text{expect to equal } h \triangleright \text{ex}_L \circ f &: h \triangleright f^{\uparrow L} \circ \text{ex}_L = (h \triangleright f^{\uparrow L}) \triangleright \text{ex}_L \\ \text{use Eq. (7.12)} &: = \text{pu}_C \triangleright (h \triangleright f^{\uparrow L}) \triangleright \text{ex}_P = \text{pu}_C \triangleright (h \triangleright f^{\uparrow L}) \circ \text{ex}_P \\ \text{definition of } f^{\uparrow L}, \text{ see Eq. (6.16)} &: = \text{pu}_C \triangleright f^{\downarrow C} \circ h \circ f^{\uparrow P} \circ \text{ex}_P \\ \text{naturality law for } \text{ex}_P &: = \text{pu}_C \triangleright f^{\downarrow C} \circ h \circ \text{ex}_P \circ f \quad . \end{aligned}$$

We expect the last expression to equal

$$h \triangleright \text{ex}_L \circ f = \text{pu}_C \triangleright h \triangleright \text{ex}_P \circ f = \text{pu}_C \triangleright h \circ \text{ex}_P \circ f \quad .$$

This is possible only if $\text{pu}_C \triangleright f^{\downarrow C} = \text{pu}_C$ for all f . This motivates us to assume this law as the **naturality law** of pu_C for pointed contrafunctors. With this last assumption, we have finished proving the naturality law for ex_L . The code for ex_L is

`def ???`

We will analyze pointed contrafunctors in Section 7.3.8.

Recursive types Consider a functor F defined by a recursive equation $F^A \triangleq S^{A,F^A}$ where S is a bifunctor (see Section 6.2.2). The functor F is pointed if a method $\text{ex}_F : (F^A \Rightarrow A) \cong (S^{A,F^A} \Rightarrow A)$ 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 $F^A \Rightarrow A$ is already available when applied to the recursively used F^A . Then we can use the `bimap` method of S to map $S^{A,F^A} \Rightarrow S^{A,A}$. 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 method $\text{ex}_S : S^{A,A} \Rightarrow A$ exists satisfying the corresponding naturality law

$$\text{ex}_S \circ f = \text{bimap}_S(f)(f) \circ \text{ex}_S \quad . \quad (7.13)$$

Assuming that S is co-pointed, we can finally define ex_F by recursion,

$$\text{ex}_F \triangleq s : S^{A,F^A} \Rightarrow s \triangleright (\text{bimap}_S(\text{id})(\text{ex}_F)) \triangleright \text{ex}_S \quad \text{or equivalently} \quad \text{ex}_F = \text{bimap}_S(\text{id})(\text{ex}_F) \circ \text{ex}_S \quad .$$

To verify the naturality law for ex_F , we denote recursive uses by an overline and compute:

$$\begin{aligned}
 \text{expect to equal } \text{ex}_F ; f &: \underline{f^{\uparrow F}} ; \text{ex}_F \\
 \text{definition of } f^{\uparrow F}, \text{ see Eq. (6.19)} &: = \text{bimap}_S(f)(\overline{f^{\uparrow F}}) ; \underline{\text{ex}_F} \\
 \text{definition of } \text{ex}_F &: = \underline{\text{bimap}_S(f)(\overline{f^{\uparrow F}})} ; \text{bimap}_S(\text{id})(\overline{\text{ex}_F}) ; \underline{\text{ex}_S} \\
 \text{bifunctor composition law (6.10)} &: = \text{bimap}_S(f ; \text{id})(\underline{\overline{f^{\uparrow F}} ; \text{ex}_F}) ; \underline{\text{ex}_S} \\
 \text{naturality law of } \text{ex}_F &: = \underline{\text{bimap}_S(f)(\overline{\text{ex}_F ; f})} ; \underline{\text{ex}_S} \\
 \text{bifunctor composition law in reverse} &: = \text{bimap}_S(\text{id})(\overline{\text{ex}_F}) ; \text{bimap}_S(f)(\overline{f}) ; \underline{\text{ex}_S} \\
 \text{naturality law (7.13) of } \text{ex}_S &: = \underline{\text{bimap}_S(\text{id})(\overline{\text{ex}_F})} ; \underline{\text{ex}_S ; f} \\
 \text{definition of } \text{ex}_F &: = \text{ex}_F ; f \quad .
 \end{aligned}$$

7.3.8 Pointed contrafunctors

In the previous section, the function-type construction required a contrafunctor C to have a method pu_C of type $\forall A. C^A$; we called such contrafunctors **pointed**. It was also necessary to assume that the naturality law holds for all functions $f : A \Rightarrow B$,

$$\text{pu}_C \triangleright f^{\downarrow C} = \text{pu}_C \quad (7.14)$$

$$\begin{array}{ccc}
 \text{pu}_C : C^B & & B \\
 \text{contramap}_C(f) \downarrow & & \uparrow f \\
 \text{pu}_C : C^A & & A
 \end{array}$$

Beginning with this definition of pointed contrafunctors, we may simplify the formulation of the typeclass by setting $B = \mathbb{1}$ in the naturality law (7.14) and denoting $\text{wu}_C \triangleq \text{pu}_C^{\mathbb{1}}$. The law (7.14) then gives

$$\text{pu}_C^A = \text{wu}_C \triangleright (_ : A \Rightarrow 1)^{\downarrow C} \quad . \quad (7.15)$$

In this way, we express the `pure` method through a chosen value $\text{wu}_C : C^{\mathbb{1}}$. For the same reasons as in the case of pointed functors, pu_C and wu_C are computationally equivalent. The law (7.14) for pu_C will be satisfied automatically if pu_C is defined via Eq. (7.15). To verify that, compute

$$\begin{aligned}
 \text{expect to equal } \text{pu}_C^A &: \text{pu}_C^B \triangleright f^{\downarrow C} \\
 \text{use definition (7.15)} &: = \text{wu}_C \triangleright (_ : B \Rightarrow 1)^{\downarrow C} \triangleright f^{\downarrow C} \\
 \text{composition law for contrafunctor } C &: = \text{wu}_C \triangleright (f ; (_ : B \Rightarrow 1))^{\downarrow C} \\
 \text{compute function composition} &: = \text{wu}_C \triangleright (_ : A \Rightarrow 1)^{\downarrow C} = \text{pu}_C^A \quad .
 \end{aligned}$$

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 purec[F[_]: Pointed : Contrafunctor, A]: F[A] = implicitly[Pointed[F]].wu.cmap(_ => ())
```

We now apply structural analysis to pointed contrafunctors, similarly to Section 6.2.4.

Fixed types This construction is the same as for pointed functors (Section 7.3.6). A fixed type Z gives a constant contrafunctor $C^A \triangleq Z$. 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 $\text{Id}^A \triangleq A$ is not a contrafunctor, it remains to consider the functor compositions C^{F^A} and F^{C^A} where C is a contrafunctor and F is a functor.

If C is pointed, we can always obtain a value pu_C of type C^A for any type A , in particular for $A = F^\perp$ (whether or not a value of type F^\perp 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]]]](purec[C, F[Unit]])
```

Creating a value of type F^{C^\perp} requires F to have a `pure` method that could be applied to a value of type C^\perp to compute a value of type F^{C^\perp} . 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^\perp and D^\perp , we can compute the pair $C^\perp \times D^\perp$. This makes the product contrafunctor $L^A \triangleq C^A \times D^A$ 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 $L^A \triangleq C^A + D^A$ as either $\text{wu}_L = \text{wu}_C + \mathbb{0}^{D^\perp}$ or $\text{wu}_L = \mathbb{0}^{C^\perp} + \text{wu}_D$.

Function types The exponential contrafunctor construction is $L^A \triangleq F^A \Rightarrow C^A$, where C^\bullet is a contrafunctor and F^\bullet is a functor. To create a value of type L^\perp means to create a function of type $F^\perp \Rightarrow C^\perp$. That function actually cannot use its argument of type F^\perp for computing a value C^\perp since F is an arbitrary functor. So, the only possibility is to create a constant function $(_ : F^\perp \Rightarrow \text{wu}_C)$ where we have assumed that C^\bullet is pointed, so that a value $\text{wu}_C : C^\perp$ is available. Therefore, $F^A \Rightarrow C^A$ 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 is $C^A \triangleq S^{A,C^A}$ where $S^{A,R}$ is a contrafunctor in A and a functor in R . A value of type C^\perp will exist when the recursive type equation $T \triangleq S^{\perp,T}$ defines a non-void type T . This condition is similar to what we saw for pointed functors, and the resulting construction is the same.

7.4 Summary

***What problems can we solve now?

Define arbitrary PTTFs and use them to define type classes (PTVFs)

Define them together or separately, combine them at will

Use the Cats library to define instances for standard type classes

Derive type class instances automatically from previous ones

Reason about higher-order type functions, types, and kinds as necessary

What problems cannot be solved with these tools?

Automatically derive type class instances for polynomial data types

see [The guide to “shapeless”, chapter 3](#)

Derive a recursive type generically from an arbitrary type function

Given a type function $F[_]$, define a recursive type R via $R = F[R]$

This R will be a function of F ; denote that type function by $Y[F[_]]$

This Y must be defined by a type equation like this,

```
type Y[F[_]] = F[Y[F]] // Does not compile ("cyclic type").
```

Automatically derive type class instances for such recursive types

That requires type-level recursion (type-level fixpoints), see [matryoshka](#)

This and other advanced topics are found in [this blog post from 2010](#)

7.4.1 Solved examples

Example 7.4.1.1 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

Example 7.4.1.2 Define a `Monoid` instance for the type `1 + (String ⇒ String)`.

Solution

Example 7.4.1.3 Show that if A is a monoid and B is a semigroup then $A + B$ is a monoid.

Solution

Example 7.4.1.4 Using the `cats` library, define a `Functor` instance for `type F[T] = Seq[Try[T]]`.

Solution

Example 7.4.1.5 Using the `cats` library, define a `Bifunctor` instance for $Q^{X,Y} \triangleq X + X \times Y$.

Solution

Example 7.4.1.6 Define a `Contrafunctor` type class having the method `contramap`:

```
def contramap[A, B](c: C[A])(f: B => A): C[B]
```

Implement a `Contrafunctor` instance for the type constructor $C^A \triangleq A \Rightarrow \text{Int}$.

Solution

Example 7.4.1.7 Define `Functor` instance for recursive type $Q^A \triangleq (\text{Int} \Rightarrow A) + \text{Int} + Q^A$.

Solution

Example 7.4.1.8* Assuming that F^A and G^A are functors, define a `Functor` instance for $F^A + G^A$ using a function parameterized by the type constructors F and G .

Solution

7.4.2 Exercises

Exercise 7.4.2.1 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 7.4.2.2 Implement a `Monoid` instance for the type `String × (1 + Int)`.

Exercise 7.4.2.3 If A is a monoid and R any type, implement a `Monoid` instance for $R \Rightarrow A$:

```
def monoidFunc[A: Monoid, R]: Monoid[R => A] = ???
```

With `R = Boolean`, use the type equivalence $(R \Rightarrow A) = (2 \Rightarrow A) \cong A \times A$ and verify that `monoidFunc[A, Boolean]` is the same as the monoid instance for $A \times A$ computed by `monoidPair[A, A]` (see Section 7.3.4).

Exercise 7.4.2.4 Show that if s is a semigroup then `Option[S]` is a monoid.

Exercise 7.4.2.5 Using the `cats` library, implement a `Functor` instance for `type F[T] = Future[Seq[T]]`.

Exercise 7.4.2.6 Using the `cats` library, implement a `Bifunctor` instance for $B^{X,Y} \triangleq (\text{Int} \Rightarrow X) + Y \times Y$.

Exercise 7.4.2.7 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 $P^A \triangleq A \Rightarrow (\text{Int} \times A)$.

Exercise 7.4.2.8 Define a `Functor` instance for the recursive type $Q^A \triangleq \text{String} + A \times A \times Q^A$.

Exercise 7.4.2.9 Show explicitly that a value $wu_C : C^1$ is computationally equivalent to a value $pu_C : \forall A. C^A$ satisfying the naturality law (7.14).

Exercise 7.4.2.10* Assuming that F^A and G^A are functors, define a `Functor` instance for $F^A \times G^A$. Use a function parameterized by the type constructors F and G .

Exercise 7.4.2.11* Define a `Functor` instance for $F^A \Rightarrow G^A$ where F^A is a contrafunctor and G^A is a functor. Implement that as a function parameterized by the type constructors F and G . For the contrafunctor F , use either the `Contrafunctor` typeclass from Example 7.4.1.6 or the `cats` library's `Contravariant` type class for F^A .

7.5 Discussion

7.5.1 Recursive types and recursive values

A recursive type is defined by a type equation such as $T \triangleq \mathbb{1} + \text{Int} + T \times T$. A recursive value (usually a function) is defined by a formula ***

There must be a clearly defined base case and inductive step must be clearly defined.

7.5.2 Inductive typeclasses and their properties

We have seen examples of typeclasses that have a similar form: *** and similar properties.

A typeclass is called **inductive** if instances for a type A are values of type $P^A \Rightarrow A$ with some functor P . The value of type $P^A \Rightarrow A$ represents all the methods of the typeclass. 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 instances have type $\mathbb{1} + A \times A \Rightarrow A$ ***can be Product of two inductive typeclasses always has an instance:

$$(P^A \Rightarrow A) \times (P^B \Rightarrow B) \Rightarrow P^{A \times B} \Rightarrow A \times B .$$

Recursive type construction works for inductive typeclasses. If $T \triangleq S^T$ then assuming we can implement $f : (P^A \Rightarrow A) \Rightarrow P^{S^A} \Rightarrow S^A$, we define

$$x : P^T \Rightarrow T \triangleq p : P^T \Rightarrow f(x)(p) .$$

This recursive definition is valid because 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 $f(x)(p)$ can be converted from type S^T to type T . In Scala code, the conversions between T and S^T are implemented by the constructor and accessor methods of the case class that wraps the type T .

The typeclasses `HasDefault`, `Semigroup`, and `Monoid` are inductive because they can be written as

$$\text{HasDefault}^A \cong \mathbb{1} \Rightarrow A , \quad \text{Semigroup}^A \cong A \times A \Rightarrow A , \quad \text{Monoid}^A \cong (\mathbb{1} + A \times A) \Rightarrow A .$$

We can implement these typeclasses via the general `InductiveTypeclass`. For example, to implement `Semigroup`:

```
type SemigroupStructure[A] = (A, A)
implicit val functorSemigroupStructure = new Functor[SemigroupStructure] { ... }
type Semigroup[A] = InductiveTypeclass[SemigroupStructure, A]
```

***The `Functor` and `Pointed` typeclasses are inductive if we look at the level of type constructors (higher-order functions on types).

The `Extractor` and `Copointed` typeclasses are **co-inductive**, because they have the form $A \Rightarrow S^A$. Co-product works for co-inductive typeclasses, rather than the product construction. The `Eq` typeclass is neither inductive nor co-inductive.

7.5.3 Inheritance and automatic conversions of typeclasses**7.5.4 Typeclasses with more than one type parameter (type relations)**

Higher-order type functions

8 Computations in functor blocks. I. Filterable functors

8.1 Slides

8.1.1 Computations within a functor context

Example:

$$\sum_{x \in \mathbb{Z}; 0 \leq x \leq 100; \cos x > 0} \sqrt{\cos x} \approx 38.71$$

Scala code:

```
(0 to 100).map(math.cos(_)).filter(_ > 0).map(math.sqrt).sum
```

Using Scala's `for/yield` syntax ("functor block", "for comprehension")

```
(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 block" is a syntax for manipulating data within a container

Container must be a functor (has `map` such that the laws hold)

Data changes but remains within the same container

A **filterable functor** is a functor that has a `withFilter` method

Can use "`if`" when `withFilter(p: A⇒Boolean): F[A] ⇒ F[A]` is defined

What are the required laws for `withFilter`?

What data types are filterable functors?

8.1.2 Filterable functors: Intuitions I

Intuition: the `filter` call *may decrease* the number of data items held
a filterable container can hold *more or fewer* data items of type T

Examples:

`Option[T] ≡ 1 + T`

`Some(123).filter(_ > 0)` returns `Some(123)`

`Some(123).filter(_ == 1)` returns `None`

`Some(123).withFilter(_ == 1).map(identity)` returns `None`

`List[T] ≡ 1 + T + T × T + T × T × T + ...`

`List(10, 20, 30).filter(_ > 10)` returns `List(20, 30)`

`List(10, 20, 30).filter(_ == 1)` returns `List()`

What we learn from these examples:

The data type must contain a *disjunction* having different counts of T

When the predicate `p` returns `false` on some T values, the remaining data goes to a part of the disjunction that has fewer T values

Values `x` are *algebraically* replaced by 1 (a `Unit`) when `p(x) = false`

The container can become "empty" as a result of filtering

8.1.3 Examples of filterable functors I

Consider these business requirements:

An order can be placed on Tuesday and/or on Friday

An order is approved under certain conditions (amount < \$1000, etc.)

```
final case class Orders[A](tue: Option[A], fri: Option[A]) {
  def withFilter(p: A ⇒ Boolean): Orders[A] =
    Orders(tue.filter(p), fri.filter(p))
}
Orders(Some(500), Some(2000)).withFilter(_ < 1000)
// returns Orders(Some(500), None) – see example code
```

This functor type is written as $F^A = (1 + A) \times (1 + A)$

When a value does not pass the filter, the A is replaced by 1

Filtering is applied independently to both parts of the product type

What if additional business requirements were given:

(a) both orders must be approved, or else no orders can be placed

or

(b) both orders can be placed if at least one of them is approved

Does this still make sense as “filtering”?

Need mathematical laws to decide this

8.1.4 Filterable functors: Intuitions II

Intuition: computations in the functor block should “make sense”

we should be able to reason correctly by looking at the program text

A schematic example of a functor block program using `map` and `filter`:

```
for { // computations lifted into the List functor
  x ← List(...) // the first line has "←", other lines do not
  y = f(x) // will become a "map(f)" after compilation
  if p1(y) // will become a "withFilter(p1)"
  if p2(y)
  z = g(x, y)
  if q(x, y, z) // – more conditions, etc.; see example code
} yield // for all x in list, such that conditions hold, compute this:
  k(x, y, z) // all the new values will stay within the container
```

What we intuitively expect to be true about such programs:

`y = f(x); if p(y);` is equivalent to `if p(f(x)); y = f(x);`

`if p1(y); if p2(y);` is equivalent to `if p1(y) && p2(y)`

When a filter predicate `p(x)` returns `true` for all `x`, we can delete the line “`if p(x)`” from the program with no change to the results

When a filter predicate `p(x)` returns `false` for some `x` then *that x* will be excluded from computations performed after “`if p(x)`”

8.1.5 Examples of filterable functors II: Checking the laws

Properties 1 – 4 are expressed as laws for $\text{filter}(p \Rightarrow \text{Boolean}) \Rightarrow F^A \Rightarrow F^A$:

$\text{fmap } f^{A \Rightarrow B} ; \text{ filter } p^{B \Rightarrow \text{Boolean}} = \text{filter } (f ; p) ; \text{ fmap } f^{A \Rightarrow B}$

$\text{filter } p_1^{A \Rightarrow \text{Boolean}} ; \text{ filter } p_2^{A \Rightarrow \text{Boolean}} = \text{filter } (x \Rightarrow p_1(x) \wedge p_2(x))$

$\text{filter } (x^A \Rightarrow \text{true}) = \text{id}^{F^A \Rightarrow F^A}$

$\text{filter } p ; \text{ fmap } f^{A \Rightarrow B} = \text{filter } p ; \text{ fmap } (f|_p)$ where $f|_p$ is the *partial function* defined as `{ case x if p(x) = f(x) }` – only works if $p(x)$ holds

Can define a type class `Filterable`, method `withFilter`

Check the laws for the `Orders` functor (see example code)

Laws hold for the `Orders` functor with / without business rule (a)

Another filterable functor: $F^A \equiv 1 + A \times A$ (“collapsible product”)

Examples of functors that are *not* filterable:

"Orders" with additional business rule (b) – breaks law 2 for some $p_{1,2}$

F^A defining `filter` in a special way e.g. for $A = \text{Int}$ – breaks law 1

$F^A \equiv 1 + A$ defining filter $(p)(x) \equiv 1 + 0$ breaks law 3

$F^A \equiv A$ – must define filter $(p^{A \Rightarrow \text{Boolean}})(x^A) = x$, breaking law 4

$F^A \equiv A \times (1 + A)$ – unable to remove the first A , breaking law 4

The equational laws 1–4 specify *rigorously* what it means to "filter data"!

8.1.6 Worked examples I: Programming with filterables

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 setup.

A server received a sequence of requests. Each request must be authenticated. Once a non-authenticated request is found, no further requests are accepted. Is this setup described by a filterable functor?

For each of these functors, determine whether they are filterable, and if so, implement `withFilter` via a type class:

`final case class P[T](first: Option[T], second: Option[(T, T)])`

$F^A \equiv \text{Int} + \text{Int} \times A + \text{Int} \times A \times A + \text{Int} \times A \times A \times A$

$F^A = \text{NonEmptyList}^A$ defined recursively as $F^A \equiv A + A \times F^A$

$F^{Z,A} \equiv Z + \text{Int} \times Z \times A \times A$ (with respect to the type parameter A)

$F^{Z,A} \equiv 1 + Z + \text{Int} \times A \times \text{List}^A$ (w.r.t. the type parameter A)

* Show that $C^{Z,A} = A \Rightarrow 1 + Z$ is a filterable *contrafunctor* w.r.t. A (implement `withFilter` with the same type signature; no law checking)

8.1.7 Exercises I

Confucius gave 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 setup described by a filterable functor?

Define `evenFilter(p)` on an `IndexedSeq[T]` such that a value $x: T$ is retained if $p(x)=\text{true}$ and only if the sequence has an *even* number of elements y for which $p(y)=\text{false}$. Does this define a filterable functor?

Implement `filter` for these functors if possible (law checking optional):

$F^A \equiv \text{Int} + \text{String} \times A \times A \times A$

`final case class Q[A, Z](id: Long, user1: Option[(A, Z)], user2: Option[(A, Z)])` – with respect to the type parameter A

$F^A = \text{MyTree}^A$ defined recursively as $F^A \equiv 1 + A \times F^A \times F^A$

`final case class R[A](x: Int, y: Int, z: A, data: List[A])`, where the standard functor `List` already has `withFilter` defined

* Show that $C^A \equiv A + A \times A \Rightarrow 1 + Z$ is a filterable contrafunctor

8.1.8 Filterable functors: The laws in depth I

Is there a shorter formulation of the laws that is easier to remember?

Intuition: When $p(x) = \text{false}$, replace $x: A$ by $1: \text{Unit}$ in $F[A]$

(1) How to replace x by 1 in $F[A]$ without breaking the types?

(2) How to transform the resulting type back to $F[A]$?

We could do (1) if instead of F^A we had F^{1+A} i.e. `F[Option[A]]`

Now use `filter` to replace A by 1 in each item of type $1 + A$

Get F^{1+A} from F^A using `inflate : F^A \Rightarrow F^{1+A} = \text{fmap}(\text{Some}^{A \Rightarrow 1+A})`

Filter $F^{1+A} \Rightarrow F^{1+A}$ using $\text{fmap}(x^{1+A} \Rightarrow \text{filter}_{\text{Opt}}(p^{\text{A} \Rightarrow \text{Boolean}})(x))$

$$\text{filter } p : F^A \xrightarrow{\text{inflate}} F^{1+A} \xrightarrow{\text{fmap}(\text{filter}_{\text{Opt}} p)} F^{1+A} \xrightarrow{\text{deflate}} F^A$$

Doing (2) means *defining* a function $\text{deflate} : F[\text{Option}[A]] \Rightarrow F[A]$

standard library already has $\text{flatten}[T] : \text{Seq}[\text{Option}[T]] \Rightarrow \text{Seq}[T]$

Simplify $\text{fmap}(\text{Some}^{A \Rightarrow 1+A}) ; \text{fmap}(\text{filter}_{\text{Opt}} p) = \text{fmap}(\text{bop}(p))$ where we defined $\text{bop}(p) : (A \Rightarrow 1+A) \equiv x \Rightarrow$

`Some(x).filter(p)`

In this way, express `filter` through `deflate` (see example code)

$\text{filter } p = \text{fmap}(\text{bop } p) ; \text{deflate}$. – Notation: $\text{bop } p$ is $\text{bop}(p)$, like $\cos x$

$$\text{filter } p : F^A \xrightarrow{\text{fmap}(\text{bop } p)} F^{1+A} \xrightarrow{\text{deflate}} F^A$$

8.1.9 Filterable functors: Using `deflate`

So far we have expressed `filter` through `deflate`

We can also express `deflate` through `filter` (assuming law 4 holds):

$$\text{deflate} : F^{1+A} \xrightarrow{\text{filter}(_.\text{nonEmpty})} F^{1+A} \xrightarrow{\text{fmap}(_.\text{get})} F^A$$

```
def deflate[F[_], A](foa: F[Option[A]]): F[A] =
  foa.filter(_.nonEmpty).map(_.get) // _.get is 0 + x^A => x^A
  // for F = Seq, this would be foa.collect { case Some(x) => x }
  // for arbitrary functor F we need to use the partial function, _.get
```

This means `deflate` and `filter` are **computationally equivalent**

We could specify filterable functors by implementing `deflate`

The implementation of `filter` would then be derived by library

Use `deflate` to verify that some functors are certainly not filterable:

$F^A = A + A \times A$. Write $F^{1+A} = 1 + A + (1+A) \times (1+A)$

cannot map $F^{1+A} \Rightarrow F^A$ because we do not have $1 \rightarrow A$

$F^A = \text{Int} \Rightarrow A$. Write $F^{1+A} = \text{Int} \Rightarrow 1 + A$

type signature of `deflate` would be $(\text{Int} \Rightarrow 1 + A) \Rightarrow \text{Int} \Rightarrow A$

cannot map $F^{1+A} \Rightarrow F^A$ because we do not have $1 + A \rightarrow A$

`deflate` is easier to implement and to reason about

* **Filterable functors: The laws in depth II** We were able to define `deflate` only by assuming that law 4 holds

Now, law 4 is satisfied *automatically* if `filter` is defined via `deflate`!

Denote $\psi_p^{F^A \Rightarrow F^{1+A}} \equiv \text{fmap}(\text{bop } p)$ for brevity, then $\text{filter } p = \psi_p ; \text{deflate}$

Law 4 then becomes: $\psi_p ; \text{deflate} ; \text{fmap} f^{A \Rightarrow B} = \psi_p ; \text{deflate} ; \text{fmap} f|_P$

$$\begin{array}{ccc} & F^{1+A} & \\ \psi_p \nearrow & \xrightarrow{\text{deflate}} & F^A \searrow \text{fmap} f^{A \Rightarrow B} \\ F^A & & \\ & \psi_p \nearrow & \\ & F^{1+A} & \\ & \xrightarrow{\text{deflate}} & F^A \searrow \text{fmap} f^{A \Rightarrow B}|_P \end{array}$$

We would like to interchange `deflate` and `fmap` in both sides

We need a *naturality* law; let's express law 1 through `deflate`:

$$\text{fmap} f^{A \Rightarrow B} ; \psi_p ; \text{deflate}^{F,B} = \psi_{f \circ p} ; \text{deflate}^{F,A} ; \text{fmap} f^{A \Rightarrow B}$$

$$\begin{array}{ccccc} & F^B & & F^{1+B} & \\ \text{fmap} f^{A \Rightarrow B} \nearrow & \xrightarrow{\psi_p} & & \searrow \text{deflate}^{F,B} & \\ F^A & & & & F^B \\ & \psi_{f \circ p} \nearrow & & & \\ & F^{1+A} & \xrightarrow{\text{deflate}^{F,A}} & F^A & \searrow \text{fmap} f^{A \Rightarrow B} \end{array}$$

8 Computations in functor blocks. I. Filterable functors

Can we simplify $\text{fmap } f \circ \psi_p = \text{fmap } f \circ \text{fmap } (\text{bop } p) = \text{fmap } (f \circ \text{bop } p)$?

* **Filterable functors: The laws in depth III** Have property: $f^{A \Rightarrow B} \circ \text{bop } (p^{B \Rightarrow \text{Boolean}}) = \text{bop } (f \circ p) \circ \text{fmap}^{\text{Opt}} f$ (see code)

$$\begin{array}{ccccc} & f^{A \Rightarrow B} & & \text{bop } p & \\ A & \nearrow & B & \searrow & \\ & \text{bop } (f \circ p) & \nearrow & 1 + B & \\ & & 1 + A & \nearrow & \text{fmap}^{\text{Opt}} f \end{array}$$

We can now rewrite Law 1 as

$$\text{fmap } (\text{bop } (f \circ p)) \circ \text{fmap } (\text{fmap}^{\text{Opt}} f) \circ \text{deflate} = \text{fmap } (\text{bop } (f \circ p)) \circ \text{deflate} \circ \text{fmap } f$$

Remove common prefix $\text{fmap } (\text{bop } (f \circ p)) \circ \dots$ from both sides:

$$\text{fmap } (\text{fmap}^{\text{Opt}} f^{A \Rightarrow B}) \circ \text{deflate}^{F, B} = \text{deflate}^{F, A} \circ \text{fmap } f^{A \Rightarrow B} \quad \text{-- law 1 for deflate}$$

$$\begin{array}{ccccc} & \text{fmap } (\text{fmap } f) & & \text{deflate}^{F, B} & \\ F^{1+A} & \nearrow & F^{1+B} & \searrow & \\ & \text{deflate}^{F, A} & \nearrow & F^B & \\ & & F^A & \nearrow & \text{fmap } f^{A \Rightarrow B} \end{array}$$

deflate: $F^{1+A} \Rightarrow F^A$ is a **natural transformation** (has naturality law)

Example: $F^A = 1 + A \times A$

$$F^{1+A} = 1 + (1 + A) \times (1 + A) = 1 + 1 \times 1 + A \times 1 + 1 \times A + A \times A$$

natural transformations map containers $G^A \Rightarrow H^A$ by rearranging data in them

* **Filterable functors: The laws in depth IV** The naturality law for **deflate**:

$$\text{fmap } (\text{fmap}^{\text{Opt}} f^{A \Rightarrow B}) \circ \text{deflate}^{F, B} = \text{deflate}^{F, A} \circ \text{fmap } f^{A \Rightarrow B}$$

Law 4 expressed via **deflate**:

$$\begin{array}{ccccc} & \psi_p & & \text{deflate} & \\ F^A & \nearrow & F^{1+A} & \xrightarrow{\text{deflate}} & F^A \xrightarrow{\text{fmap } f^{A \Rightarrow B}} F^B \\ & \psi_p & \nearrow & & \\ & F^{1+A} & \xrightarrow{\text{deflate}} & F^A & \xrightarrow{\text{fmap } f_{|P}^{A \Rightarrow B}} F^B \end{array}$$

$$\psi_p \circ \text{deflate}^{F, A} \circ \text{fmap } f^{A \Rightarrow B} = \psi_p \circ \text{deflate}^{F, A} \circ \text{fmap } f_{|P}$$

Use naturality to interchange **deflate** and **fmap** in both sides of law 4:

$$\psi_p \circ \text{fmap } (\text{fmap}^{\text{Opt}} f) \circ \text{deflate}^{F, B} = \psi_p \circ \text{fmap } (\text{fmap}^{\text{Opt}} f_{|P}) \circ \text{deflate}^{F, B}$$

[omit $\text{deflate}^{F, B}$ from both sides; expand ψ_p]

$$\text{bop } p \circ \text{fmap}^{\text{Opt}} f = \text{bop } p \circ \text{fmap}^{\text{Opt}} f_{|P} \quad \text{-- check this by hand:}$$

`x => Some(x).filter(p).map(f)`

`x => Some(x).filter(p).map { case x if p(x) => f(x) }`

These functions are equivalent because law 4 holds for **Option**

Filterable functors: The laws in depth V Maybe $\psi_p \circ \text{deflate}$ is easier to handle than **deflate**? Let us define

$$\text{fmapOpt}^{F, A, B}(f^{A \Rightarrow 1+B}) : F^A \Rightarrow F^B = \text{fmap } f \circ \text{deflate}^{F, B}$$

$$\begin{array}{ccccc} & \text{fmap } f^{A \Rightarrow 1+B} & & \text{deflate}^{F, B} & \\ F^A & \nearrow & F^{1+B} & \searrow & \\ & \text{fmapOpt} f^{A \Rightarrow 1+B} & \nearrow & F^B & \end{array}$$

fmapOpt and **deflate** are *equivalent*: $\text{deflate}^{F, A} = \text{fmapOpt}^{F, 1+A, A}(\text{id}^{1+A \Rightarrow 1+A})$

Express laws 1 – 3 in terms of **fmapOpt**: do they get simpler?

Express **filter** through **fmapOpt**: $\text{filter } p = \text{fmapOpt}^{F, A, A}(\text{bop } p)$

Consider the expression needed for law 2: $x \Rightarrow p_1(x) \wedge p_2(x)$

$$\text{bop } (x \Rightarrow p_1(x) \wedge p_2(x)) = x^A \Rightarrow (\text{bop } p_1)(x).flatMap(\text{bop } p_2) \text{ -- see code}$$

Denote this computation by \diamond_{Opt} and write

$$q_1^{A \Rightarrow 1+B} \diamond_{\text{Opt}} q_2^{B \Rightarrow 1+C} \equiv x^A \Rightarrow q_1(x).flatMap(q_2)$$

Similar to composition of functions, except the types are $A \Rightarrow 1 + A$

This is a particular case of **Kleisli composition**; the general case: $\diamond_M : (A \Rightarrow M^B) \Rightarrow (B \Rightarrow M^C) \Rightarrow (A \Rightarrow M^C)$; we set $M^A \equiv 1 + A$

The **Kleisli identity** function: $\text{id}_{\diamond_{\text{Opt}}}^{A \Rightarrow 1+A} \equiv x^A \Rightarrow \text{Some}(x)$

Kleisli composition \diamond_{Opt} is associative and respects the Kleisli identity!

`fmapOpt` lifts a Kleisli_{Opt} function $f^{A \Rightarrow 1+B}$ into the functor F

Filterable functors: The laws in depth VI Simplifying down to two laws

Only *two* laws are necessary for `fmapOpt`!

Identity law (covers old law 3):

$$\text{fmapOpt}(\text{id}_{\diamond_{\text{Opt}}}^{A \Rightarrow 1+A}) = \text{id}^{F^A \Rightarrow F^A}$$

Composition law (covers old laws 1 and 2):

$$\text{fmapOpt}(f^{A \Rightarrow 1+B}) ; \text{fmapOpt}(g^{B \Rightarrow 1+C}) = \text{fmapOpt}(f \diamond_{\text{Opt}} g)$$

$$\begin{array}{ccc} & F^B & \\ f\text{mapOpt}(f^{A \Rightarrow 1+B}) & \nearrow & \searrow f\text{mapOpt}(g^{B \Rightarrow 1+C}) \\ F^A & \xrightarrow{\text{fmapOpt}(f \diamond_{\text{Opt}} g)} & F^C \end{array}$$

The two laws for `fmapOpt` are very similar to the two functor laws

Both of them use more complicated types than the old laws

Conceptually, the new laws are simpler (lift $f^{A \Rightarrow 1+B}$ into $F^A \Rightarrow F^B$)

* **Filterable functors: The laws in depth VII** Showing that old laws 1 – 3 follow from the identity and composition laws for `fmapOpt`

Old law 3 is *equivalent* to the identity law for `fmapOpt`:

$$\text{filter}(x^A \Rightarrow \text{true}) = \text{fmap}(x^A \Rightarrow 0 + x) ; \text{deflate} = \text{fmapOpt}(\text{id}_{\diamond_{\text{Opt}}}^{A \Rightarrow 1+A}) = \text{id}^{F^A \Rightarrow F^A}$$

Derive old law 2: need to work with $q_{1,2} \equiv \text{bop}(p_{1,2}) : A \Rightarrow 1 + A$

The Boolean conjunction $x \Rightarrow p_1(x) \wedge p_2(x)$ corresponds to $q_1 \diamond_{\text{Opt}} q_2$

Apply the composition law to Kleisli functions of types $A \Rightarrow 1 + A$:

$$\begin{aligned} \text{filter } p_1 ; \text{filter } p_2 &= \text{fmapOpt } q_1 ; \text{fmapOpt } q_2 \\ &= \text{fmapOpt}(q_1 \diamond_{\text{Opt}} q_2) = \text{fmapOpt}(\text{bop}(x \Rightarrow p_1(x) \wedge p_2(x))) \end{aligned}$$

Derive old law 1:

express `filter` through `fmapOpt`; old law 1 becomes

$$\text{fmap } f ; \text{fmapOpt}(\text{bop } p) = \text{fmapOpt}(\text{bop}(f ; p)) ; \text{fmap } f - \text{eq. (*)}$$

lift $f^{A \Rightarrow B}$ to Kleisli_{Opt} by defining $k_f^{A \Rightarrow 1+B} = f ; \text{id}_{\diamond_{\text{Opt}}}$; then we have $\text{fmapOpt}(k_f) = \text{fmap } k_f ; \text{deflate} = \text{fmap } f ; \text{fmap id}_{\diamond_{\text{Opt}}} ; \text{deflate} = \text{fmap } f$

rewrite eq. (*) as $\text{fmapOpt}(k_f \diamond_{\text{Opt}} \text{bop } p) = \text{fmapOpt}(\text{bop}(f ; p) \diamond_{\text{Opt}} k_f)$

it remains to show that $k_f \diamond_{\text{Opt}} \text{bop } p = \text{bop}(f ; p) \diamond_{\text{Opt}} k_f$

use the properties $k_f \diamond_{\text{Opt}} q = f ; q$ and $q \diamond_{\text{Opt}} k_f = q ; \text{fmap}^{\text{Opt}} f$, and $f ; \text{bop } p = \text{bop}(f ; p) ; \text{fmap}^{\text{Opt}} f$ (property from slide 11)

8.1.10 Summary: The methods and the laws

Filterable functors can be defined via `filter`, `deflate`, or `fmapOpt`

All three methods are *equivalent* but have different roles:

The easiest to use in program code is `filter` / `withFilter`

The easiest type signature to implement and reason about is `deflate`

Conceptually, the laws are easiest to remember with `fmapOpt`

* The 2 laws for `fmapOpt` are the 2 functor laws with a Kleisli “twist”

* Category theory accommodates this via a generalized definition of functors as liftings between “twisted” types. Compare:

fmap : $(A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$ – ordinary container (“endofunctor”)
 contrafmap : $(B \Rightarrow A) \Rightarrow F^A \Rightarrow F^B$ – lifting from reversed functions
 fmapOpt : $(A \Rightarrow 1 + B) \Rightarrow F^A \Rightarrow F^B$ – lifting from Kleisli_{Opt}-functions

CT gives us some *intuitions* about how to derive better laws:

look for type signatures that resemble a generalized sort of “lifting”
 look for natural transformations and use the naturality law

However, CT does not directly provide any derivations for the laws
 you will not find the laws for `filter` or `deflate` in any CT book

CT is abstract, only gives hints about possible further directions

investigate functors having “liftings” with different type signatures
 replace `Option` in the Kleisli_{Opt} construction by another functor

8.1.11 Structure of filterable functors

How to recognize a filterable functor by its type?

Intuition from `deflate`: reshuffle data in F^A after replacing some A 's by 1

“reshuffling” usually means reusing different parts of a disjunction

Some constructions of exponential-polynomial filterable functors

$F^A = Z$ (constant functor) for a fixed type Z (define `fmapOpt f = id`)

Note: $F^A = A$ (identity functor) is *not* filterable

$F^A \equiv G^A \times H^A$ for any filterable functors G^A and H^A

$F^A \equiv G^A + H^A$ for any filterable functors G^A and H^A

$F^A \equiv G^{H^A}$ for *any* functor G^A and filterable functor H^A

$F^A \equiv 1 + A \times G^A$ for a filterable functor G^A

Note: *pointed* types P are isomorphic to $1 + Z$ for some type Z

Example of non-trivial pointed type: $A \Rightarrow A$

Example of non-pointed type: $A \Rightarrow B$ when A is different from B

So $F^A \equiv P + A \times G^A$ where P is a pointed type and G^A is filterable

Also have $F^A \equiv P + A \times A \times \dots \times A \times G^A$ similarly

$F^A \equiv G^A + A \times F^A$ (recursive) for a filterable functor G^A

$F^A \equiv G^A \Rightarrow H^A$ if contrafunctor G^A and functor H^A both filterable

Note: the functor $F^A \equiv G^A \Rightarrow A$ is not filterable

8.1.12 * Worked examples II: Constructions of filterable functors I

(2) The `fmapOpt` laws hold for $F^A \times G^A$ if they hold for F^A and G^A

For $f^{A \Rightarrow 1+B}$, get $\text{fmapOpt}_F(f) : F^A \Rightarrow F^B$ and $\text{fmapOpt}_G(f) : G^A \Rightarrow G^B$

Define $\text{fmapOpt}_{F \times G} f \equiv p^{F^A} \times q^{G^A} \Rightarrow \text{fmapOpt}_F(f)(p) \times \text{fmapOpt}_G(f)(q)$

Identity law: $f = \text{id}_{\text{Opt}}$, so $\text{fmapOpt}_F f = \text{id}$ and $\text{fmapOpt}_G f = \text{id}$

Hence we get $\text{fmapOpt}_{F+G}(f)(p \times q) = \text{id}(p) \times \text{id}(q) = p \times q$

Composition law:

$$\begin{aligned} & (\text{fmapOpt}_{F \times G} f_1 ; \text{fmapOpt}_{F+G} f_2)(p \times q) \\ &= \text{fmapOpt}_{F \times G} (f_2) (\text{fmapOpt}_F(f_1)(p) \times \text{fmapOpt}_G(f_1)(q)) \\ &= (\text{fmapOpt}_F f_1 ; \text{fmapOpt}_F f_2)(p) \times (\text{fmapOpt}_G f_1 ; \text{fmapOpt}_G f_2)(q) \\ &= \text{fmapOpt}_F (f_1 \diamond_{\text{Opt}} f_2)(p) \times \text{fmapOpt}_G (f_1 \diamond_{\text{Opt}} f_2)(q) \\ &= \text{fmapOpt}_{F \times G} (f_1 \diamond_{\text{Opt}} f_2)(p \times q) \end{aligned}$$

Exactly the same proof as that for functor property for $F^A \times G^A$

this is because `fmapOpt` corresponds to a generalized functor

New proofs are necessary only when using non-filterable functors

these are used in constructions 4 – 6

* **Worked examples II: Constructions of filterable functors II** (5) The `fmapOpt` laws hold for $F^A \equiv 1 + A \times G^A$ if they hold for G^A

For $f^{A \Rightarrow 1+B}$, get $\text{fmapOpt}_G(f) : G^A \Rightarrow G^B$

Define $\text{fmapOpt}_F(f)(1 + a^A \times q^{G^A})$ by returning $0 + b \times \text{fmapOpt}_G(f)(q)$ if the argument is $0 + a \times q$ and if $f(a) = 0 + b$, and returning $1 + 0$ otherwise

Identity law: $f = \text{id}_{\circ_{\text{Opt}}}$, so $f(a) = 0 + a$ and $\text{fmapOpt}_G f = \text{id}$

Hence we get $\text{fmapOpt}_F(\text{id}_{\circ_{\text{Opt}}})(1 + a \times q) = 1 + a \times q$

Composition law: need only to check for arguments $0 + a \times q$, and only when $f_1(a) = 0 + b$ and $f_2(b) = 0 + c$, in which case $(f_1 \circ_{\text{Opt}} f_2)(a) = 0 + c$; then

$$\begin{aligned} & (\text{fmapOpt}_F f_1 \circ \text{fmapOpt}_F f_2)(0 + a \times q) \\ &= \text{fmapOpt}_F(f_2)(\text{fmapOpt}_F(f_1)(0 + a \times q)) \\ &= \text{fmapOpt}_F(f_2)(0 + b \times \text{fmapOpt}_G(f_1)(q)) \\ &= 0 + c \times (\text{fmapOpt}_G f_1 \circ \text{fmapOpt}_G f_2)(q) \\ &= 0 + c \times \text{fmapOpt}_G(f_1 \circ_{\text{Opt}} f_2)(q) \\ &= \text{fmapOpt}_F(f_1 \circ_{\text{Opt}} f_2)(0 + a \times q) \end{aligned}$$

This is a “greedy filter”: if $f(a)$ is empty, will delete all data in G^A

* **Worked examples II: Constructions of filterable functors III** (6) The `fmapOpt` laws hold for $F^A \equiv G^A + A \times F^A$ if they hold for G^A

For $f^{A \Rightarrow 1+B}$, we have $\text{fmapOpt}_G(f) : G^A \Rightarrow G^B$ and $\text{fmapOpt}'_F(f) : F^A \Rightarrow F^B$ (for use in recursive arguments as the inductive assumption)

Define $\text{fmapOpt}_F(f)(q^{G^A} + a^A \times p^{F^A})$ by returning $0 + \text{fmapOpt}'_F(f)(p)$ if $f(a) = 1 + 0$, and $\text{fmapOpt}_G(f)(q) + b \times \text{fmapOpt}'_F(f)(p)$ otherwise

Identity law: $\text{id}_{\circ_{\text{Opt}}}(x) \neq 1 + 0$, so $\text{fmapOpt}_F(\text{id}_{\circ_{\text{Opt}}})(q + a \times p) = q + a \times p$

Composition law: $(\text{fmapOpt}_F(f_1) \circ \text{fmapOpt}_F(f_2))(q + a \times p) = \text{fmapOpt}_F(f_1 \circ_{\text{Opt}} f_2)(q + a \times p)$

For arguments $q + 0$, the laws for G^A hold; so assume arguments $0 + a \times p$. When $f_1(a) = 0 + b$ and $f_2(b) = 0 + c$, the proof of the previous example will go through. So we need to consider the two cases $f_1(a) = 1 + 0$ and $f_1(a) = 0 + b$, $f_2(b) = 1 + 0$

If $f_1(a) = 1 + 0$ then $(f_1 \circ_{\text{Opt}} f_2)(a) = 1 + 0$; to show $\text{fmapOpt}'_F(f_2)(\text{fmapOpt}'_F(f_1)(p)) = \text{fmapOpt}'_F(f_1 \circ_{\text{Opt}} f_2)(p)$, use the inductive assumption about $\text{fmapOpt}'_F$ on p

If $f_1(a) = 0 + b$ and $f_2(b) = 1 + 0$ then $(f_1 \circ_{\text{Opt}} f_2)(a) = 1 + 0$; to show $\text{fmapOpt}_F(f_2)(0 + b \times \text{fmapOpt}'_F(f_1)(p)) = \text{fmapOpt}'_F(f_1 \circ_{\text{Opt}} f_2)(p)$, rewrite $\text{fmapOpt}_F(f_2)(0 + b \times \text{fmapOpt}'_F(f_1)(p)) = \text{fmapOpt}'_F(f_2)(\text{fmapOpt}'_F(f_1)(p))$ and again use the inductive assumption about $\text{fmapOpt}'_F$ on p

This is a “list-like filter”: if $f(a)$ is empty, will recurse into nested F^A data

8.1.13 Worked examples II: Constructions of filterable functors IV

Use known filterable constructions to show that $F^A \equiv (\text{Int} \times \text{String}) \Rightarrow (1 + \text{Int} \times A + A \times (1 + A) + (\text{Int} \Rightarrow 1 + A + A \times A \times \text{String}))$ is a filterable functor

Instead of implementing `Filterable` and verifying laws by hand, we analyze the structure of this data type and use known constructions

Define some auxiliary functors that are parts of the structure of F^A ,

$$R_1^A = (\text{Int} \times \text{String}) \Rightarrow A \text{ and } R_2^A = \text{Int} \Rightarrow A$$

$$G^A = 1 + \text{Int} \times A + A \times (1 + A) \text{ and } H^A = 1 + A + A \times A \times \text{String}$$

$$\text{Now we can rewrite } F^A = R_1[G^A + R_2[H^A]]$$

G^A is filterable by construction 5 because it is of the form $G^A = 1 + A \times K^A$ with filterable functor $K^A = 1 + \text{Int} + A$

K^A is of the form $1 + A + X$ with constant type X , so it is filterable by constructions 1 and 3 with the `Option` functor $1 + A$

H^A is filterable by construction 5 with $H^A = 1 + A \times (1 + A \times \text{String})$, while $1 + A \times \text{String}$ is filterable by constructions 5 and 1

Constructions 3 and 4 show that $R_1[G^A + R_2[H^A]]$ is filterable

Note that there is more than one way of implementing `Filterable` here

8.1.14 * Exercises II

Implement a `Filterable` instance for `type F[T] = G[H[T]]` assuming that the functor `H[T]` already has a `Filterable` instance (construction 4). Verify the laws rigorously (i.e. by calculations, not tests).

For `type F[T] = Option[Int => Option[(T, T)]]`, implement a `Filterable` instance. Show that the filterable laws hold by using known filterable constructions (avoiding explicit proofs or tests).

Implement a `Filterable` instance for $F^A \equiv G^A + \text{Int} \times A \times A \times F^A$ (recursive) for a filterable functor G^A . Verify the laws rigorously.

Show that $F^A = 1 + A \times G^A$ is in general *not* filterable if G^A is an arbitrary (non-filterable) functor; it is enough to give an example.

Show that $F^A = 1 + G^A + H^A$ is filterable if $1 + G^A$ and $1 + H^A$ are filterable (even when G^A and H^A are by themselves not filterable).

Show that the functor $F^A = A + (\text{Int} \Rightarrow A)$ is not filterable.

Show that one can define `deflate: C1+A ⇒ CA` for any contrafunctor C^A (not necessarily filterable), similarly to how one can define `inflate: FA ⇒ F1+A` for any functor F^A (not necessarily filterable).

8.1.15 * Bonus slide I: Definition of filterable contrafunctors

When is a contrafunctor filterable?

When a contrafunctor C^A with `contrafmap : (B ⇒ A) ⇒ CA ⇒ CB` has also

`filter/withFilter: (A ⇒ Boolean) ⇒ CA ⇒ CA` – same as for functors

`inflate: CA ⇒ C1+A` and `contrafmapOpt: (B ⇒ 1+A) ⇒ CA ⇒ CB`

All three functions are computationally equivalent...

`filter(pA=Boolean) = inflateCA⇒C1+A ; contrafmap(bop p)`

`inflateCA⇒C1+A = contrafmap(0 + xA ⇒ x) ; filter(_ ⇒ true)`

`contrafmapOptB⇒1+A = inflate ; contrafmap f`

`inflate = contrafmapOpt(id1+A⇒1+A)`

but have different laws

4 laws (naturality, conjunction, identity, partial function) for `filter`

3 laws (naturality, conjunction, identity) for `inflate`

2 laws (identity, contracomposition) for `contrafmapOpt`

as before, `contrafmapOpt` is a “twisted” version of `contrafmap`

Examples of filterable contrafunctors

$C^A \equiv A \Rightarrow 1 + Z$ where Z is a fixed type

$C^A \equiv 1 + A \Rightarrow Z$

Examples of non-filterable contrafunctors

$C^A \equiv A \times F^A \Rightarrow Z$ – cannot implement `inflate`

8.1.16 * Bonus slide II: Structure of filterable contrafunctors

How to build up a filterable contrafunctor from parts?

Filterable contrafunctors “can consume fewer data items”

The easiest function to consider first is `inflate`

Some constructions of filterable contrafunctors:

$C^A = Z$ (constant contrafunctor)

Functor constructions (no need to check laws for these):

$F^A \equiv G^A \times H^A$ for any filterable contrafunctor G^A and H^A

$F^A \equiv G^A + H^A$ for any filterable contrafunctor G^A and H^A

$F^A \equiv G^{H^A}$ for H^A a filterable (contra)functor and G^A any (contra)functor – various combinations possible here

$F^A \equiv G^A \Rightarrow H^A$ if functor G^A and contrafunctor H^A both filterable

Special constructions:

$F^A \equiv 1 + A \times G^A \Rightarrow H^A$ where G^A and H^A are filterable

$F^A \equiv A \times G^A \Rightarrow 1 + H^A$ if G^A and H^A are filterable

8.1.17 Addendum

Notes added after the tutorial was complete

A polynomial functor F^A is filterable if and only if it can be expressed (isomorphically) as $F^A = C \times (1 + \dots)$ where C is a constant type. An example of a polynomial functor not of this form is $F^A = P + Q \times A$, where $P \neq Q$ are constant types and P is not pointed (not known to be isomorphic to $1 + Z$ for some type Z).

Given a polynomial functor F^A , one can decide algorithmically whether F^A is filterable, and derive *some* implementation of `deflate` or `withFilter` such that the laws hold. However, typically there will be many legitimate implementations, and the application-specific filtering requirements are not obviously guessable in advance.

8.2 Practical use

8.2.1 Discussion

8.3 Laws and structure

8.3.1 Discussion

9 Computations in functor blocks. II. Semimonads and monads

9.1 Slides, part 1

9.1.1 Computations within a functor context: Semimonads

Intuitions behind adding more “generator arrows”

Example of nested iterations:

$$\sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n f(i, j, k)$$

Using Scala’s `for/yield` syntax (“functor block”)

```
(for { i ← 1 to n
      j ← 1 to n
      k ← 1 to n
    } yield f(i, j, k)
  ).sum
```

```
(1 to n).flatMap { i =>
  (1 to n).flatMap { j =>
    (1 to n).map { k =>
      f(i, j, k)
    }
  }.sum
}
```

`map` replaces the last left arrow, `flatMap` replaces other left arrows

When the functor is *also* filterable, we can use “`if`” as well

Standard library defines `flatMap()` as replacement of `map() ∘ flatten`

`(1 to n).map(j => ...).flatten` is `(1 to n).flatMap(j => ...)`

Functors having `flatMap/flatten` are “flattenable” or **semimonads**

Most of them also have method `pure: A ⇒ F[A]` and so are **monads**

The method `pure` is not relevant in the functor block

We will not need `pure` in this part of the tutorial; focus on semimonads

9.1.2 How `flatMap` works with lists

consider `List(x1, x2, x3).flatMap(x ⇒ f(x))`

assume that

```
f: X ⇒ List[Y]
f(x1) = List(y0, y1)
f(x2) = List(y2)
f(x3) = List(y3, y4, y5, y6)
```

then the result is `List(y0, y1, y2, y3, y4, y5, y6)`

if we first do `.map(f)` then `flatten`:

```
List(x1, x2, x3).map(f).flatten =
List(List(y0, y1), List(y2), List(y3, y4, y5, y6)).flatten =
List(y0, y1, y2, y3, y4, y5, y6)
```

9.1.3 What is `flatMap` doing with the data in a collection?

Consider this schematic code, using `Seq` as the container type:

```
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)
```

Computations are repeated for all i , for all j , etc., from each collection
 All “generator lines” must use the same container type
 Each generator line finally computes a container of *that* type
 The total number of resulting data items is $\leq m * n * p$
 All the resulting data items must fit within *the same* container type!
 The set of *container capacity counts* must be closed under multiplication
 What container types have this property?
`Seq, NonEmptyList, Iterator, Stream` – can hold *any* number of elements $\geq \text{min. count}$
`Option, Either, Try, Future` – can hold 0 or 1 elements (“pass/fail”)
 “Tree-like” containers, e.g. can hold only 3, 6, 9, 12, ... elements
 “Non-standard” containers: $F^A \equiv \text{String} \Rightarrow A; F^A \equiv (A \Rightarrow \text{Int}) \Rightarrow \text{Int}$

9.1.4 Worked examples I: List-like monads

`Seq, NonEmptyList, Iterator, Stream`

Typical tasks for “list-like” monads:

Create a list of all combinations or all permutations of a sequence
 Traverse a “solution tree” with DFS and filter out incorrect solutions
 Can use eager (`Seq`) or lazy (`Iterator, Stream`) evaluation strategies
 Usually, list-like containers have many additional methods
 append, prepend, concat, fill, fold, scan, etc.
 Worked examples: see code
 All permutations of `Seq("a", "b", "c")`
 All subsets of `Set("a", "b", "c")`
 All subsequences of length 3 out of a given sequence
 Generalize examples 1-3 to support arbitrary length n instead of 3
 All solutions of the “8 queens” problem
 Generalize example 5 to solve n -queens problem
 Transform Boolean formulas between CNF and DNF

9.1.5 Intuitions for pass/fail monads

`Option, Either, Try, Future`

Container F^A can hold $n = 1$ or $n = 0$ values of type A
 Such containers will have methods to create “pass” and “fail” values
 Schematic example of a functor block program using the `Try` functor:
`val result: Try[A] = for {` // computations in the Try functor
 `x ← Try(...)` // first computation; may fail
 `y = f(x)` // no possibility of failure in this line
 `if p(y)` // the entire expression will fail if this is false
 `z ← Try(g(x, y))` // may fail here
 `r ← Try(...)` // may fail here as well
`} yield r` // r is of type A, so result is of type Try[A]
 Computations may yield a result ($n = 1$), or may fail ($n = 0$)
 The functor block chains several such computations *sequentially*
 Computations are sequential even if using the `Future` functor!
 Once any computation fails, the entire functor block fails ($0 * n = 0$)
 Only if *all* computations succeed, the functor block returns *one* value
 Filtering can also make the entire functor block fail
 “Flat” functor block replaces a chain of nested `if/else` or `match/case`

9.1.6 Worked examples II: Pass/fail monads

Type constructors:

$$\text{Option}[A] \equiv 1 + A$$

$$\text{Either}[Z, A] \equiv Z + A$$

$$\text{Try}[A] \equiv \text{Either}[\text{Throwable}, A]$$

Typical tasks for pass/fail monads:

Perform a linear sequence of computations that may fail

Avoid crashing on failure, instead return an *error value*

Worked examples: see code

Read values of Java properties, checking that they all exist

Obtain values from `Future` computations in sequence

Make arithmetic safe by returning error messages in `Either`

Pass/fail chain: sequencing computations that may throw an exception

9.1.7 Intuitions for tree-like monads

Examples of tree-like recursive type constructors:

$$F^A \equiv A + F^A \times F^A \text{ (binary tree)}$$

$$F^A \equiv A + S^{F^A} \text{ (S-shaped tree, where } S \text{ is a functor)}$$

$$F^A \equiv A \times A + F^A \times F^A \text{ (binary tree with binary leaves)}$$

$$F^A \equiv S^A + S^{F^A} \text{ (S-shaped tree with } S \text{-shaped leaves)}$$

Implementing `flatMap` for these type constructors is recursive

See example code

Note: trees with *S*-shaped leaves are *semi-monads* but not monads

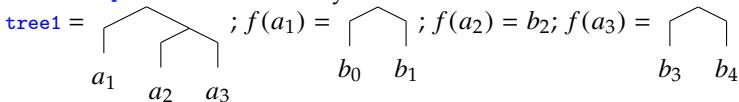
Example of a *non-monadic* tree-like functor:

$$F^A \equiv A + A \times A + A \times A \times A \times A + \dots \text{ (powers of 2)}$$

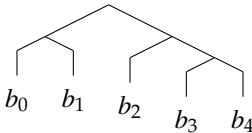
Recursive definition: $F^A = A + F^{A \times A}$

9.1.8 Worked examples III: Tree-like monads

How `flatMap` works for a binary tree: assume `f: A → Tree[B]` and



then `tree1.flatMap(f)` =



grafting subtrees plays the role of “flattening”

Typical tasks for tree-like monads:

Traverse a tree, graft subtrees at leaves

Substitute subexpressions in a syntax tree

Worked examples: see code

Implement a tree of `String` properties with arbitrary branching

Implement variable substitution for a simple arithmetic language

9.1.9 Worked examples IV: Single-value monads

Pretend that container holds exactly 1 value, together with a “context”

Usually, methods exist to insert a value and to work with the “context”

Typical tasks for single-value monads:

Managing extra information about computations along the way

Chaining computations with a nonstandard evaluation strategy

Examples: see code

Writer: Perform computations and log information about each step

$\text{Writer}^A \equiv A \times W$ where W is a monoid or a semigroup

Reader: Read-only context, or dependency injection

$\text{Reader}^A \equiv E \Rightarrow A$ where E represents the “environment”

Eval: Perform a sequence of lazy or memoized computations

$\text{Eval}^A \equiv A + (1 \Rightarrow A)$

Cont: A chain of asynchronous operations

$\text{Cont}^A \equiv (A \Rightarrow R) \Rightarrow R$ where R is the fixed “result” type

State: A sequence of steps that update state while returning results

$\text{State}^A \equiv S \Rightarrow A \times S$ where S is the fixed “state” value type

9.1.10 Deriving the types of single-value monads

Motivation for the choice of the type constructors Writer^A , Reader^A , State^A , Cont^A

We want previous values to be transformed via `flatMap` to next values

Writer: a computation $(A \Rightarrow B)$ and some info (W) about it

$x^A \Rightarrow f(x) : B$ and $x^A \Rightarrow g(x) : W$; the type is $(A \Rightarrow B) \times (A \Rightarrow W)$

this function should have type $A \Rightarrow \text{Writer}^B$, hence $\text{Writer}^B \equiv B \times W$

use the “arithmetic” Curry-Howard to transform types: $b^a w^a = (bw)^a$

Reader: Read-only context, or “environment” of type E

$x^A \Rightarrow f(r, x) : B$ where r^E is fixed; the type is $A \times E \Rightarrow B$

this function should have type $A \Rightarrow \text{Reader}^B$, hence $\text{Reader}^B \equiv E \Rightarrow B$

we used the “arithmetic” Curry-Howard to transform $b^{ae} = (b^e)^a$

Cont: A computation that registers an asynchronous callback

$x^A \Rightarrow f(cb) : 1$ where $cb : B \Rightarrow 1$ (usually, callbacks return `Unit`)

the type is $A \Rightarrow (B \Rightarrow 1) \Rightarrow 1$; this function should have type $A \Rightarrow \text{Cont}^B$, hence $\text{Cont}^B \equiv (B \Rightarrow 1) \Rightarrow 1$

generalize to $\text{Cont}^A \equiv (A \Rightarrow R) \Rightarrow R$ where R is a fixed “result” type

State: A computation can update state (S) while producing a result

$x^A \Rightarrow f(x, s)$ and $s^S := g(x, s)$; the type is $(A \times S \Rightarrow B) \times (A \times S \Rightarrow S)$

this will be $A \Rightarrow \text{State}^B$ if $\text{State}^B \equiv (S \Rightarrow B) \times (S \Rightarrow S) \equiv S \Rightarrow B \times S$

we used the “arithmetic” Curry-Howard: $b^{as} s^{as} = (b^s s^s)^a = ((bs)^s)^a$

9.1.11 Exercises I

For a given `Set[Int]`, compute all subsets (w, x, y, z) of size 4 such that $w < x < y < z$ and $w + z = x + y$

Given 3 sequences xs , ys , zs of type `Seq[Int]`, compute all (x, y, z) such that $x \in xs$, $y \in ys$, $z \in zs$ and $x < y < z$ and $x + y + z < 10$

Solve the n -queens problem on an $3 \times 3 \times 3$ cube

Write a tiny library for arithmetic using `Future`'s; use it to compute $1 + 2 + \dots + 100$ via `for/yield` and verify the result. E.g. implement:

```
const: Int => Future[Int]
add(x: Int): Int => Future[Int]
isEqual(x: Int): Int => Future[Boolean]
```

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 this safe.

Given a semigroup W , make a semimonad out of $F^A \equiv E \Rightarrow A \times W$

Implement a semimonad instance for the (recursive) type constructor $F^A = A + A \times A + F^A + F^A \times F^A$

Find the largest prime number below 1000 via a simple `Sieve of Eratosthenes`; use the `State[S, Int]` monad with `S = Array[Boolean]`

9.2 Slides, part 2

9.2.1 Semimonad laws I: The intuitions

What properties of functor block programs do we expect to have?

In `x ← c`, the value of `x` will go over items held in container `c`

Manipulating items in container is followed by a generator:

<code>x ← cont1</code>	<code>y ← cont1</code>
<code>y = f(x)</code>	<code>.map(x ⇒ f(x))</code>
<code>z ← cont2(y)</code>	<code>z ← cont2(y)</code>

`cont1.flatMap(x ⇒ cont2(f(x))) = cont1.map(f).flatMap(y ⇒ cont2(y))`

Manipulating items in container is preceded by a generator:

<code>x ← cont1</code>	<code>x ← cont1</code>
<code>y ← cont2(x)</code>	<code>z ← cont2(x)</code>
<code>z = f(y)</code>	<code>.map(f)</code>

`cont1.flatMap(cont2).map(f) = cont1.flatMap(x ⇒ cont2(x).map(f))`

Within a generator, `for { ... } yield` can be inlined:

<code>x ← cont</code>	<code>yy ← for { x ← cont</code>
<code>y ← p(x)</code>	<code> y ← p(x) } yield y</code>
<code>z ← cont2(y)</code>	<code> z ← cont2(yy)</code>

`cont.flatMap(x ⇒ p(x).flatMap(cont2)) = cont.flatMap(p).flatMap(cont2)`

9.2.2 Semimonad laws II: The laws for `flatMap`

For brevity, write `fml` instead of `flatMap`

A **semimonad** S^A has $fml^{[A,B]} : (A \Rightarrow S^B) \Rightarrow S^A \Rightarrow S^B$ with 3 laws:

$fml(f^{A \Rightarrow B} ; g^{B \Rightarrow C}) = fml f ; fml g$ (naturality in A)

$$\begin{array}{ccc} & S^B & \\ \xrightarrow{\quad fml f^{A \Rightarrow B} \quad} & \searrow & \swarrow \xrightarrow{\quad fml g^{B \Rightarrow C} \quad} \\ S^A & \xrightarrow{\quad fml(f^{A \Rightarrow B} ; g^{B \Rightarrow C}) \quad} & S^C \end{array}$$

$fml(f^{A \Rightarrow S^B} ; fml g^{B \Rightarrow C}) = fml f ; fml g$ (naturality in B)

$$\begin{array}{ccc} & S^B & \\ \xrightarrow{\quad fml f^{A \Rightarrow S^B} \quad} & \searrow & \swarrow \xrightarrow{\quad fml g^{B \Rightarrow C} \quad} \\ S^A & \xrightarrow{\quad fml(f^{A \Rightarrow S^B} ; fml g^{B \Rightarrow C}) \quad} & S^C \end{array}$$

$fml(f^{A \Rightarrow S^B} ; fml g^{B \Rightarrow S^C}) = fml f ; fml g$ (associativity)

$$\begin{array}{ccc} & S^B & \\ \xrightarrow{\quad fml f^{A \Rightarrow S^B} \quad} & \searrow & \swarrow \xrightarrow{\quad fml g^{B \Rightarrow S^C} \quad} \\ S^A & \xrightarrow{\quad fml(f^{A \Rightarrow S^B} ; fml g^{B \Rightarrow S^C}) \quad} & S^C \end{array}$$

Is there a shorter and clearer formulation of these laws?

9.2.3 Semimonad laws III: The laws for `flatten`

The methods `flatten` (denoted by `ftn`) and `flatMap` are equivalent:

$ftn^{[A]} : S^{S^A} \Rightarrow S^A \equiv fml^{[S^A, A]}(m^{S^A} \Rightarrow m)$

$fml(f^{A \Rightarrow S^B}) \equiv fmap f ; ftn$

$$\begin{array}{ccc} & S^{S^B} & \\ \xrightarrow{\quad fml f^{A \Rightarrow S^B} \quad} & \searrow & \swarrow \xrightarrow{\quad ftn \quad} \\ S^A & \xrightarrow{\quad fml(f^{A \Rightarrow S^B}) \quad} & S^B \end{array}$$

It turns out that `flatten` has only 2 laws:

$$\text{fmap}(\text{fmap } f^{A \Rightarrow B}) \circ \text{ftn}^{[B]} = \text{ftn}^{[A]} \circ \text{fmap } f \text{ (naturality)}$$

$$\begin{array}{ccccc} & & S^{S^B} & & \\ & \nearrow \text{fmap}(\text{fmap } f^{A \Rightarrow B}) & \searrow \text{ftn}^{[B]} & & \\ S^{S^A} & & & & S^B \\ \searrow \text{ftn}^{[A]} & & \nearrow \text{fmap } f^{A \Rightarrow B} & & \\ & S^A & & & \end{array}$$

$$\text{fmap}(\text{ftn}^{[A]}) \circ \text{ftn}^{[A]} = \text{ftn}^{[S^A]} \circ \text{ftn}^{[A]} \text{ (associativity)}$$

$$\begin{array}{ccccc} & & S^{S^A} & & \\ & \nearrow \text{fmap}(\text{ftn}^{[A]}) & \searrow \text{ftn}^{[A]} & & \\ S^{S^{S^A}} & & & & S^A \\ \searrow \text{ftn}^{[S^A]} & & \nearrow \text{ftn}^{[A]} & & \\ & S^{S^A} & & & \end{array}$$

9.2.4 Equivalence of a natural transformation and a “lifting”

Equivalence of `flm` and `ftn`: $\text{ftn} = \text{flm}(\text{id})$; $\text{flm } f = \text{fmap } f \circ \text{ftn}$

We saw this before: $\text{deflate} = \text{fmapOpt}(\text{id})$; $\text{fmapOpt } f = \text{fmap } f \circ \text{deflate}$

Is there a general pattern where two such functions are equivalent?

Let $\text{tr} : F^{G^A} \Rightarrow F^A$ be a natural transformation (F and G are functors)

Define $\text{ftr} : (A \Rightarrow G^B) \Rightarrow F^A \Rightarrow F^B$ by $\text{ftr } f = \text{fmap } f \circ \text{tr}$

It follows that $\text{tr} = \text{ftr}(\text{id})$, and we have equivalence between tr and ftr :

$$\begin{aligned} \text{tr} : F^{G^A} &\Rightarrow F^A = \text{ftr}(m^{G^A} \Rightarrow m) \\ \text{ftr}(f^{A \Rightarrow G^B}) &= \text{fmap } f \circ \text{tr} \end{aligned}$$

$$\begin{array}{ccc} & F^{G^B} & \\ \nearrow \text{fmap } f^{A \Rightarrow G^B} & & \searrow \text{tr} \\ F^A & \xrightarrow{\quad} & F^B \\ \text{ftr}(f^{A \Rightarrow G^B}) & & \end{array}$$

An automatic law for ftr (“naturality in A ”) follows from the definition: $\text{fmap } g \circ \text{ftr } f = \text{fmap } g \circ \text{fmap } f \circ \text{tr} = \text{fmap}(g \circ f) \circ \text{tr} = \text{ftr}(g \circ f)$

This is why tr always has *one law fewer* than ftr

To demonstrate equivalence in the direction $\text{ftr} \rightarrow \text{tr}$: Start with an arbitrary ftr satisfying “naturality in A ”, then obtain $\text{tr} = \text{ftr}(\text{id})$ from it, then verify $\text{ftr } f = \text{fmap } f \circ \text{tr}$ with that tr ; $\text{fmap } f \circ \text{ftr}(\text{id}) = \text{ftr}(f \circ \text{id}) = f$

9.2.5 Semimonad laws IV: Deriving the laws for `flatten`

Denote for brevity $q^\uparrow \equiv \text{fmap } q$ for any function q (“lifting” $q^{A \Rightarrow B}$ to S)

Express $\text{flm } f = f^\uparrow \circ \text{ftn}$ and substitute that into flm ’s 3 laws:

$$\text{flm}(f \circ g) = f^\uparrow \circ \text{flm } g \text{ gives } (f \circ g)^\uparrow \circ \text{ftn} = f^\uparrow \circ g^\uparrow \circ \text{ftn}$$

- this law holds automatically due to functor composition law

$$\text{flm}(f \circ g^\uparrow) = \text{flm } f \circ g^\uparrow \text{ gives } (f \circ g^\uparrow)^\uparrow \circ \text{ftn} = f^\uparrow \circ \text{ftn} \circ g^\uparrow;$$

using the functor composition law, we reduce this to

$$g^{\uparrow\uparrow} \circ \text{ftn} = \text{ftn} \circ g^\uparrow - \text{this is the naturality law}$$

$\text{flm}(f \circ \text{flm } g) = \text{flm } f \circ \text{flm } g$ with functor composition law gives $f^\uparrow \circ g^{\uparrow\uparrow} \circ \text{ftn}^\uparrow \circ \text{ftn} = f^\uparrow \circ \text{ftn} \circ g^\uparrow \circ \text{ftn}$; using ftn ’s naturality and omitting the common factor $f^\uparrow \circ g^{\uparrow\uparrow}$, we get $\text{ftn}^\uparrow \circ \text{ftn} = \text{ftn} \circ \text{ftn} - \text{associativity law}$

`flatten` has the simplest type signature *and* the fewest laws

It is usually easy to check naturality!

Parametricity theorem: Any *pure, fully parametric* code for a function of type $F^A \Rightarrow G^A$ will implement a natural transformation

Checking `flatten`’s associativity needs *a lot* more work!

The `cats` library has a `FlatMap` type class, defining `flatten` via `flatMap`

9.2.6 Checking the associativity law for standard monads

Implement `flatten` for these functors and check the laws (see code):

`Option` monad: $F^A \equiv 1 + A$; $\text{ftn} : 1 + (1 + A) \Rightarrow 1 + A$

`Either` monad: $F^A \equiv Z + A$; $\text{ftn} : Z + (Z + A) \Rightarrow Z + A$

`List` monad: $F^A \equiv \text{List}^A$; $\text{ftn} : \text{List}^{\text{List}^A} \Rightarrow \text{List}^A$

`Writer` monad: $F^A \equiv A \times W$; $\text{ftn} : (A \times W) \times W \Rightarrow A \times W$

`Reader` monad: $F^A \equiv R \Rightarrow A$; $\text{ftn} : (R \Rightarrow (R \Rightarrow A)) \Rightarrow R \Rightarrow A$

`State`: $F^A \equiv S \Rightarrow A \times S$; $\text{ftn} : (S \Rightarrow (S \Rightarrow A \times S) \times S) \Rightarrow S \Rightarrow A \times S$

`Continuation` monad: $F^A \equiv (A \Rightarrow R) \Rightarrow R$; $\text{ftn} : (((A \Rightarrow R) \Rightarrow R) \Rightarrow R) \Rightarrow (A \Rightarrow R) \Rightarrow R$

Code implementing these `flatten` functions is *fully parametric* in A

Naturality of these functions follows from parametricity theorem

Associativity needs to be checked for each monad!

Example of a useful semimonad that is *not* a full monad:

$F^A \equiv A \times V \times W$; $\text{ftn}((a \times v_1 \times w_1) \times v_2 \times w_2) = a \times v_1 \times w_2$

Examples of *non-associative* (i.e. wrong) implementations of `flatten`:

$F^A \equiv A \times W \times W$; $\text{ftn}((a \times v_1 \times v_2) \times w_1 \times w_2) = a \times w_2 \times w_1$

$F^A \equiv \text{List}^A$, but `flatten` concatenates the nested lists in reverse order

9.2.7 Motivation for monads

Monads represent values with a “special computational context”

Specific monads will have methods to create various contexts

Monadic composition will “combine” the contexts associatively

It is generally useful to have an “empty context” available:

$$\text{pure} : A \Rightarrow M^A$$

Adding the empty context to another context should be a no-op

Empty context is followed by a generator:

$$\begin{array}{ll} y \leftarrow \text{pure}(x) & y = x \\ z \leftarrow \text{cont}(y) & z \leftarrow \text{cont}(y) \end{array}$$

$$\text{pure}(x).\text{flatMap}(y \Rightarrow \text{cont}(y)) = \text{cont}(x) \quad \text{pure} ; \text{flm } f = f - \text{left identity}$$

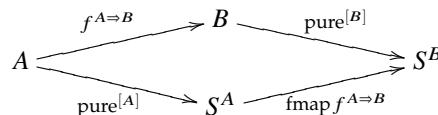
Empty context is preceded by a generator:

$$\begin{array}{ll} x \leftarrow \text{cont} & x \leftarrow \text{cont} \\ y \leftarrow \text{pure}(x) & y = x \end{array}$$

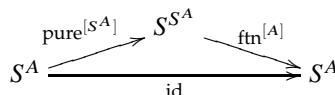
$$\text{cont}.\text{flatMap}(x \Rightarrow \text{pure}(x)) = \text{cont} \quad \text{flm } (\text{pure}) = \text{id} - \text{right identity}$$

9.2.8 The monad laws formulated in terms of `pure` and `flatten`

Naturality law for `pure`: $f ; \text{pure} = \text{pure} ; \text{fmap } f$



Left identity: $\text{pure} ; \text{flm } f = \text{pure} ; \text{fmap } f ; \text{ftn} = f ; \text{pure} ; \text{ftn} = f$ requires that $\text{pure} ; \text{ftn} = \text{id}$ (both sides applied to S^A)



Right identity: $\text{flm}(\text{pure}) = \text{fmap}(\text{pure}) \circ \text{ftn} = \text{id}^{S^A \Rightarrow S^A}$

$$\begin{array}{ccccc} & & S^{S^A} & & \\ & \nearrow \text{fmap}(\text{pure}^{[A]}) & & \searrow \text{ftn}^{[A]} & \\ S^A & \xrightarrow{\text{id}} & S^A & & \end{array}$$

9.2.9 Formulating laws via Kleisli functions

Recall: we formulated the laws of filterables via `fmapOpt`

type signature of `fmapOpt`: $(A \Rightarrow 1 + B) \Rightarrow S^A \Rightarrow S^B$

and then we had to compose functions of types $A \Rightarrow 1 + B$ via \diamond_{Opt}

Here we have $\text{flm} : (A \Rightarrow S^B) \Rightarrow S^A \Rightarrow S^B$ instead of `fmapOpt`

Can we compose **Kleisli functions** with “twisted” type, $A \Rightarrow S^B$?

Use `flm` to define **Kleisli composition**: $f^{A \Rightarrow S^B} \diamond g^{B \Rightarrow S^C} \equiv f \circ \text{flm } g$

Define **Kleisli identity** id_{\diamond} of type $A \Rightarrow S^A$ as $\text{id}_{\diamond} \equiv \text{pure}$

Composition law: $\text{flm}(f \diamond g) = \text{flm } f \circ \text{flm } g$ (same as for `fmapOpt`)

Shows that `flatMap` is a “lifting” of $A \Rightarrow S^B$ to $S^A \Rightarrow S^B$

These laws are similar to functor “lifting” laws...

except that \diamond is used for composing Kleisli functions

What are the properties of \diamond ?

Exactly similar to the properties of function composition $f \circ g$

Reformulate `flm`'s laws in terms of the \diamond operation:

`flm`'s left and right identity laws: $\text{pure} \diamond f = f$ and $f \diamond \text{pure} = f$

Associativity law: $(f \diamond g) \diamond h = f \diamond (g \diamond h)$

Follows from the `flm` law: $f \circ \text{flm}(g \circ \text{flm } h) = f \circ \text{flm } g \circ \text{flm } h$

9.2.10 * Motivation for categories and functors

Compare different “liftings” seen so far, and generalize

Category	Type $A \rightsquigarrow B$	Identity	Composition
plain functions	$A \Rightarrow B$	$\text{id} : A \Rightarrow A$	$f^{A \Rightarrow B} \circ g^{B \Rightarrow C}$
lifted to F	$F^A \Rightarrow F^B$	$\text{id} : F^A \Rightarrow F^A$	$f^{F^A \Rightarrow F^B} \circ g^{F^B \Rightarrow F^C}$
Kleisli over F	$A \Rightarrow F^B$	$\text{pure} : A \Rightarrow F^A$	$f^{A \Rightarrow F^B} \diamond g^{B \Rightarrow F^C}$

Category theory generalizes this situation

Category: a certain class of “twisted functions” $A \rightsquigarrow B$ called **morphisms**

For any two morphisms $f^{A \rightsquigarrow B}$ and $g^{B \rightsquigarrow C}$ the **composition** morphism $f \diamond g$ of type $A \rightsquigarrow C$ must exist

For each type A , the **identity** morphism id_{\diamond} of type $A \rightsquigarrow A$ must exist

Composition respects identity: $\text{id}_{\diamond} \diamond f = f$ and $f \diamond \text{id}_{\diamond} = f$

Composition is associative: $(f \diamond g) \diamond h = f \diamond (g \diamond h)$

General **functor**: a map from one category to another

A functor must `fmap` each morphism from one category to the other

Functor laws: `fmap` must preserve identity and composition

What we call “functor” is called **endofunctor** in category theory

An endofunctor's `fmap` goes from plain functions to F -lifted functions

9.2.11 * From Kleisli back to `fmap`

The Kleisli functions, $A \rightsquigarrow B \equiv A \Rightarrow S^B$, form a category iff S is a monad

`fmap` and `flatMap` are computationally equivalent to Kleisli composition:

Define `flatMap` through Kleisli: $\text{fmap } f^{A \Rightarrow S^B} \equiv \text{id}^{S^A \Rightarrow S^A} \diamond f$

Require two additional laws that connect \diamond , `fmap`, and \circ :

Left naturality: $f^{A \Rightarrow B} ; g^{B \Rightarrow S^C} = (f ; \text{pure}) \diamond g$

Right naturality: $f^{A \Rightarrow S^B} ; \text{fmap } g^{B \Rightarrow C} = f \diamond (g ; \text{pure})$

So, can define `fmap` through Kleisli: $\text{fmap } g^{A \Rightarrow B} \equiv \text{id}^{S^A \Rightarrow S^A} \diamond (g ; \text{pure})$

The laws for `pure` and `flatMap` then follow from category axioms for Kleisli:

Left and right identity laws follow from $\text{id} \diamond \text{pure} = \text{id}$ and $\text{pure} \diamond f = f$

Associativity for `flatMap` follows from $(\text{id} \diamond f) \diamond g = \text{id} \diamond (f \diamond g)$

Use “left naturality”, get: $(f ; g) \diamond h = (f ; \text{pure}) \diamond g \diamond h = f ; (g \diamond h)$

Naturality for `pure`: $\text{pure} ; \text{fmap } f = \text{pure} \diamond (f ; \text{pure}) = f ; \text{pure}$

Define `flatten`: $\text{ftn} = \text{id}^{S^{S^A} \Rightarrow S^A} \diamond \text{id}^{S^A \Rightarrow S^A}$

Naturality for `flatten`: $\text{ftn} ; \text{fmap } f = \text{id} \diamond \text{id} \diamond (f ; \text{pure}) = \text{id} \diamond \text{fmap } f$ and $\text{fmap}(\text{fmap } f) ; \text{ftn} = \text{id} \diamond ((\text{fmap } f) ; \text{pure}) ; \text{id} \diamond \text{id} = \text{id} \diamond \text{fmap } f$

9.2.12 Structure of semigroups and monoids

Semimonad contexts are combined associatively, as in a semigroup

A full monad includes an “empty” context, i.e. the identity element

Semigroup with an identity element is a monoid

Some constructions of semigroups and monoids (see code):

Any type Z is a semigroup with operation $z_1 ; z_2 = z_1$ (or z_2)

$1 + S$ is a monoid if S is (at least) a semigroup (or $S \equiv 0$)

List^A is a monoid (for any type A), also Seq^A etc.

The function type $A \Rightarrow A$ is a monoid (for any type A)

The operation $f ; g$ can be either $f ; g$ or $g ; f$

Any totally ordered type is a monoid, with \circledast defined as max or min

$S_1 \times S_2$ is a semigroup (monoid) if S_1, S_2 are semigroups (monoids)

$S_1 + S_2$ is a semigroup (monoid) if S_1, S_2 are semigroups (monoids)

`M[S]` is a monoid if `M[_]` is a monad and `S` is a monoid

$S \times P$ is a semigroup if S is a semigroup that has an **action on P**

The “action” is $\alpha : S \Rightarrow P \Rightarrow P$ such that $\alpha(s_2) ; \alpha(s_1) = \alpha(s_1 ; s_2)$

$S \times P$ is a “twisted product.” Examples: $(A \Rightarrow A) \times A$; $\text{Bool} \times (1 + A)$

Other examples of monoids: Int (many), String , Set^A , Akka’s `Route`

9.2.13 Structure of (semi)monads

How to recognize a (semi)monad by its type? Open question!

Intuition from `flatten`: reshuffle data in F^{F^A} to fit into F^A

Some constructions of exponential-polynomial (semi)monads:

$F^A \equiv Z$ (constant functor) for a fixed type Z

For a full monad, need to choose $Z = 1$

$F^A \equiv A \times G^A$ for any functor G^A (a full monad only if G^A is a monad)

$F^A \equiv G^A \times H^A$ for any (semi)monads G^A and H^A

but $G^A + H^A$ is generally *not* a semimonad

$F^A \equiv R \Rightarrow G^A$ is a (semi)monad for any (semi)monad G^A

$F^A \equiv A + G^A$ is a monad for a monad G^A (**free pointed** over G)

$F^A \equiv G^{Z+A \times W}$ is a monad if G is a monad and W a monoid

$F^A \equiv A + G^{F^A}$ (recursive) for any functor G^A (**free monad** over G)

Semimonad-only constructions:

$F^A \equiv G^A + G^{F^A}$ (recursive) for any functor G^A

$F^A \equiv H^A \Rightarrow A \times G^A$ for any contrafunctor H^A and functor G^A

Obtain a full monad only when $G^A \equiv 1$, i.e. $F^A \equiv H^A \Rightarrow A$

9.2.14 Exercises II

Show that $M[S]$ is a monoid if $M[_]$ is a monad and S is a monoid.

A framework implements a “route” type R as $R \equiv Q \Rightarrow (E + S)$, where Q is a query, E is an error response, and S is a success response. A server is defined as a “sum” of several routes. For a given query Q , the response is the first route (if it exists) that yields a success. Implement the route “summation” operation and show that it makes R into a semigroup. What would be necessary to make R into a monoid?

Verify the associativity law for the semimonad $F^A \equiv Z + \text{Bool} \times A$.

Show that the functor $F^A \equiv \text{Boolean} \times M^A$ (where M^A is an arbitrary monad) can be made into a semimonad but not into a monad.

If W and R are arbitrary fixed types, which of the functors can be made into a semimonad: $F^A \equiv W \times (R \Rightarrow A)$, $G^A = R \Rightarrow (W \times A)$?

Show that $F^A \equiv (P \Rightarrow A) + (Q \Rightarrow A)$ is not a semimonad (cannot define `flatMap`) when P and Q are arbitrary, different types.

Implement the `flatten` and `pure` methods for $D^A \equiv 1 + A \times A$ (`type D[A] = Option[(A, A)]`) in at least two significantly different ways, and show that the monad laws always fail to hold. (D^A is not a monad!)

9.2.15 Exercises II (continued)

[] A programmer implemented the `fmap` method for $F^A \equiv A \times (A \Rightarrow Z)$ as

```
def fmap[A,B](f: A⇒B): ((A, A⇒Z)) ⇒ (B, B⇒Z) =
  { case (a, az) ⇒ (f(a), (_: B) ⇒ az(a)) }
```

Show that this implementation fails to satisfy the functor laws.

Show that $P^A \equiv Z + W \times A$ is a (full) monad if W is a monoid.

Verify that the full monad laws hold for construction 4.

Implement `flatten` and `pure` for $F^A \equiv A + (R \Rightarrow A)$, where R is a fixed type, and show that all the monad laws hold.

For construction 5, show that an identity law would fail if `pure` were defined as `a ⇒ Right(Monad[G].pure(a))` instead of as `Left(a)`.

Implement the monad methods for $F^A \equiv (Z \Rightarrow 1 + A) \times \text{List}^A$ using the known monad constructions (no need to check the laws).

Implement the semimonad construction 2 by discarding the first effect (not the second), and show that the associativity law is still satisfied.

For semimonad construction 8, show that the associativity law holds.

Verify the identity laws for the State and Continuation monads.

9.2.16 Addendum: Miscellaneous remarks on monads

A non-empty list $F^A \equiv A \times \text{List}^A$ is a semigroup but not a monoid.

Any polynomial functor $F^A \equiv p(A)$ can be made into a monad when $p(x)$ is a polynomial of the form $p(x) = x^{n_1} + x^{n_2} + \dots + x^{n_k}$ for some positive integers n_1, \dots, n_k . Indeed, any F^A of this form may be built from the identity monad via constructions 3 and 5. To illustrate this, denote $E_1 \equiv 1$, $E_{n+1} \equiv 1 + E_n$. Monoid construction 2 makes E_n into monoids. Then the monads $E_n \Rightarrow A$ (reader) and $E_n \times A$ (writer) are equivalent to polynomial monads $A \times \dots \times A$ and $A + \dots + A$.

Contrafunctors cannot be monads or semimonads: if H^A is a contrafunctor then H^{H^A} is a functor, so a natural transformation between H^{H^A} and H^A (in either direction) is impossible.

An example of combining natural transformations: Given functors C, F, G and natural transformations $C^A \Rightarrow F^A$ and $C^A \Rightarrow G^A$ and taking the product, we get a natural transformation $C^A \Rightarrow F^A \times G^A$.

If M^A is a monad then M^{M^A} is not automatically a monad (need counterexample?).

Two monadic values $m_1, m_2 : M^A$ can be merged by ignoring the payload of one of them and merging the effects; and we can merge the effects in any chosen order: for $\{x \leftarrow m_1; _ \leftarrow m_2\}$ yield x or for $\{_ \leftarrow m_1; x \leftarrow m_2\}$ yield x

A curious example: The functor $Q^A \equiv (A \Rightarrow Z) \Rightarrow 1 + A$ is not a monad (and not even a lawful applicative) but $M^A \equiv (A \Rightarrow 1 + 1) \Rightarrow 1 + A$ is a “search monad”. More generally, a “selector monad” is $(A \Rightarrow P^1) \Rightarrow P^A$ for any functor P^A .

9.3 Practical use

9.3.1 Discussion

9.4 Laws and structure

9.4.1 Discussion

10 Applicative functors, contrafunctors, and profunctors

10.1 Slides, Part I

10.1.1 Motivation for applicative functors

Monads are inconvenient for expressing *independent* effects

Monads perform effects *sequentially* even if effects are independent:

```
x ← Future { c1 }
y ← Future { c2 }
z ← Future { c3 }

Future { c1 }.flatMap { x =>
  Future { c2 }.flatMap { y =>
    Future { c3 }.map { z => ... }
  }
}
```

We would like to parallelize independent computations

We would like to accumulate *all* errors, rather than stop at the first one

Changing the order of monad's effects will (generally) change the result:

```
for {
  x ← List(1, 2)
  y ← List(10, 20)
} yield f(x, y)
// f(1, 10), f(1, 20), f(2, 10), f(2, 20)

for {
  y ← List(10, 20)
  x ← List(1, 2)
} yield f(x, y)
// f(1, 10), f(2, 10), f(1, 20), f(2, 20)
```

We would like to express a computation where effects are unordered

This can be done using a method `map2`, *not* defined via `flatMap`: the desired type signature is `map2 : FA × FB ⇒ (A × B ⇒ C) ⇒ FC`

Applicative functor has `map2` and `pure` but is not necessarily a monad

10.1.2 Defining `map2`, `map3`, etc.

Consider 1, 2, 3, ... commutative and independent "effects"

```
for { x1 ← c1
} yield f(x1)                                c1.map(f)



---


for { x1 ← c1
  x2 ← c2
} yield f(x1, x2)                            (c1, c2).map2(f)



---


for { x ← c1
  x2 ← c2
  x3 ← c3
} yield f(x1, x2, x3)                      (c1, c2, c3).map3(f)
```

Generalize from `map`, `map2`, `map3` to `mapN`:

$$\begin{aligned} \text{map}_1 &: F^A \Rightarrow (A \Rightarrow Z) \Rightarrow F^Z \\ \text{map}_2 &: F^A \times F^B \Rightarrow (A \times B \Rightarrow Z) \Rightarrow F^Z \\ \text{map}_3 &: F^A \times F^B \times F^C \Rightarrow (A \times B \times C \Rightarrow Z) \Rightarrow F^Z \end{aligned}$$

10.1.3 Practical examples of using `mapN`

$F^A \equiv Z + A$ where Z is a monoid: collect all errors

$F^A = Z + A$: Create a validated case class out of validated parts

$F^A \equiv \text{Future}[A]$: perform several computations concurrently

$F^A \equiv E \Rightarrow A$: pass standard arguments to functions more easily

$F^A \equiv \text{List}^A$: transposing a matrix by using `map2`

Applicative contrafunctors and applicative profunctors

defining an instance of `Semigroup` type class from `Semigroup` parts

implement `imap2` for non-disjunctive profunctors, e.g. $Z \times A \Rightarrow A \times A$

“Fused `fold`”: automatically merge several `folds` into one

compute several running averages in one traversal (`scala-folds`)

The difference between applicative and monadic functors

define monadic folds using the “free functor” construction

compute running averages that depend on previous running averages

do not confuse this with *monad-valued* folds (`origami`)!

applicative parsers vs. monadic parsers

applicative: parse independent data, collecting all errors

monadic: parse depends on previous results, stops on errors

10.1.4 Exercises I

Implement `map2`, or `imap2` if appropriate, for these type constructors F^A :

$F^A \equiv 1 + A + A \times A$

$F^A \equiv E \Rightarrow A \times A$

$F^A \equiv Z \times A \Rightarrow A$

$F^A \equiv A \Rightarrow A \times Z$ where Z is a `Monoid`

Write a function that defines an instance of the `Monoid` type class for a tuple (A, B) whose parts already have a `Monoid` instance.

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.

Define a “regexp extractor” as a type constructor R^A describing extraction of various data from strings; the extracted data is converted to a value of type `option[A]`. Implement `zip` and `map2` for R^A .

Use parser combinators to implement an evaluator for arithmetic language containing integers and `+` symbols, for example $1 + 321 + 20$.

Use folding combinators to implement a `Fold` that computes the standard deviation of a sequence in one traversal.

10.2 Slides, Part II

10.2.1 Deriving the `ap` operation from `map2`

Can we avoid having to define map_n separately for each n ?

- Use curried arguments, $\text{fmap}_2 : (A \Rightarrow B \Rightarrow Z) \Rightarrow F^A \Rightarrow F^B \Rightarrow F^Z$
 - Set $A \equiv (B \Rightarrow Z)$ and apply fmap_2 to the identity $\text{id}^{(B \Rightarrow Z) \Rightarrow (B \Rightarrow Z)}$: obtain $\text{ap}^{[B,Z]} : F^{B \Rightarrow Z} \Rightarrow F^B \Rightarrow F^Z \equiv \text{fmap}_2(\text{id})$
 - The functions `fmap2` and `ap` are computationally equivalent:

$$\begin{array}{ccc} \text{fmap}_2 f^{A \Rightarrow B \Rightarrow Z} & = & \text{fmap } f \circ \text{ap} \\ \text{fmap } f \nearrow & & \searrow \text{ap} \\ F^A & \xrightarrow{\quad \text{fmap}_2(f^{A \Rightarrow B \Rightarrow Z}) \quad} & (F^B \Rightarrow F^Z) \end{array}$$

- The functions `fmap3`, `fmap4` etc. can be defined similarly:

$$\text{fmap}_3 f^{A \Rightarrow B \Rightarrow C \Rightarrow Z} = \text{fmap } f \circ \text{ap} \circ \text{fmap}_{F^B \Rightarrow ?} \text{ap}$$

$$\begin{array}{c} F^B \Rightarrow C \Rightarrow Z \xrightarrow{\text{ap}[B,C \Rightarrow Z]} (F^B \Rightarrow F^C \Rightarrow Z) \xrightarrow{\text{fmap}_{F^B \Rightarrow ?} \text{ap}[C,Z]} \\ \text{fmap } f \nearrow \qquad \qquad \qquad \searrow \\ F^A \xrightarrow{\text{fmap}_3(f^{A \Rightarrow B \Rightarrow C \Rightarrow Z})} (F^B \Rightarrow F^C \Rightarrow F^Z) \end{array}$$

- Using the infix syntax will get rid of `fmap_{FB \Rightarrow ?} ap` (see example code)
 - * Note the pattern: a natural transformation is equivalent to a lifting

10.2.2 Deriving the `zip` operation from `map2`

- The types $A \Rightarrow B \Rightarrow C$ and $A \times B \Rightarrow C$ are equivalent (curry/uncurry)
 - Uncurry `fmap2` to `fmap2`: $(A \times B \Rightarrow C) \Rightarrow F^A \times F^B \Rightarrow F^C$
 - Compute `fmap2(f)` with $f = \text{id}^{A \times B \Rightarrow A \times B}$, expecting to obtain a simpler natural transformation:

$$\text{zip} : F^A \times F^B \Rightarrow F^{A \times B}$$

- This is quite similar to `zip` for lists:

`List(1, 2).zip(List(10, 20)) = List((1, 10), (2, 20))`

- The functions `zip` and `fmap2` are computationally equivalent:

$$\begin{aligned} \text{zip} &= \text{fmap2(id)} \\ \text{fmap2}(f^{A \times B \Rightarrow C}) &= \text{zip} \circ \text{fmap } f \end{aligned}$$

$$\begin{array}{ccc} F^A \times F^B & \xrightarrow{\text{zip}} & F^{A \times B} & \xrightarrow{\text{fmap } f^{A \times B \Rightarrow C}} & F^C \\ & & \searrow & \nearrow & \\ & & \text{fmap2}(f^{A \times B \Rightarrow C}) & & \end{array}$$

- The functor F is **zippable** if such a `zip` exists (with appropriate laws)
 - * The same pattern: a natural transformation is equivalent to a lifting

10.2.3 * Equivalence of the operations `ap` and `zip`

- Set $A \equiv B \Rightarrow C$, get $\text{zip}^{[B \Rightarrow C, B]} : F^{B \Rightarrow C} \times F^B \Rightarrow F^{(B \Rightarrow C) \times B}$
 - Use `eval`: $(B \Rightarrow C) \times B \Rightarrow C$ and $\text{fmap } (\text{eval}) : F^{(B \Rightarrow C) \times B} \Rightarrow F^C$
 - Uncurry: $\text{app}^{[B, C]} : F^{B \Rightarrow C} \times F^B \Rightarrow F^C \equiv \text{zip} \circ \text{fmap } (\text{eval})$
 - The functions `zip` and `app` are computationally equivalent:
 - * use pair: $(A \Rightarrow B \Rightarrow A \times B) = a^A \Rightarrow b^B \Rightarrow a \times b$
 - * use $\text{fmap } (\text{pair}) \equiv \text{pair}^\uparrow$ on an fa^{F^A} , get $(\text{pair}^\uparrow fa) : F^{B \Rightarrow A \times B}$; then

$$\begin{aligned} \text{zip } (fa \times fb) &= \text{app } ((\text{pair}^\uparrow fa) \times fb) \\ \text{app}^{[B, C]} &= \text{zip}^{[B \Rightarrow C, B]} \circ \text{fmap } (\text{eval}) \end{aligned}$$

$$\begin{array}{ccc} F^{B \Rightarrow C} \times F^B & \xrightarrow{\text{zip}} & F^{(B \Rightarrow C) \times B} & \xrightarrow{\text{fmap } (\text{eval})} & F^C \\ & & \searrow & \nearrow & \\ & & \text{app}^{[B, C]} & & \end{array}$$

- Rewrite this using curried arguments: $\text{fzip}^{[A, B]} : F^A \Rightarrow F^B \Rightarrow F^{A \times B}; \text{ap}^{[B, C]} : F^{B \Rightarrow C} \Rightarrow F^B \Rightarrow F^C$; then $\text{ap } f = \text{fzip } f \circ \text{fmap } (\text{eval})$.
- Now $\text{fzip } p^{F^A} q^{F^B} = \text{ap } (\text{pair}^\uparrow p) q$, hence we may omit the argument q : $\text{fzip} = \text{pair}^\uparrow \circ \text{ap}$. With explicit types: $\text{fzip}^{[A, B]} = \text{pair}^\uparrow \circ \text{ap}^{[B, A \Rightarrow B]}$.

10.2.4 Motivation for applicative laws. Naturality laws for `map2`

Treat `map2` as a replacement for a monadic block with independent effects:

```
for {
  x ← cont1
  y ← cont2
} yield g(x, y)          map2 (
                           cont1,
                           cont2
                           ) { (x, y) ⇒ g(x, y) }
```

- Main idea: Formulate the monad laws in terms of `map2` and `pure`

Naturality laws: Manipulate data in one of the containers

```
for {
  x ← cont1.map(f)
  y ← cont2
} yield g(x, y)          for {
                           x ← cont1
                           y ← cont2
                           } yield g(f(x), y)
```

and similarly for `cont2` instead of `cont1`; now rewrite in terms of `map2`:

- Left naturality for `map2`:

```
map2(cont1.map(f), cont2)(g)
= map2(cont1, cont2){ (x, y) ⇒ g(f(x), y) }
```

- Right naturality for `map2`:

```
map2(cont1, cont2.map(f))(g)
= map2(cont1, cont2){ (x, y) ⇒ g(x, f(y)) }
```

10.2.5 Associativity and identity laws for `map2`

Inline two generators out of three, in two different ways:

```
for {
  x ← cont1
  (y, z) ← for {
    yy ← cont2
    zz ← cont3
  } yield (yy, zz)
} yield g(x, y, z)          for {
                           (x, y) ← for {
                           xx ← cont1
                           yy ← cont2
                           } yield (xx, yy)
                           z ← cont3
                           } yield g(x, y, z)
```

Write this in terms of `map2` to obtain the **associativity law for `map2`**:

```
map2(cont1, map2(cont2, cont3)((_,_)){ case(x,(y,z))⇒g(x,y,z) })
= map2(map2(cont1, cont2)((_,_), cont3){ case((x,y),z)⇒g(x,y,z) })
```

Empty context precedes a generator, or follows a generator:

```
for { x ← pure(a)
      y ← cont
} yield g(x, y)          for {
                           y ← cont
                           } yield g(a, y)
```

Write this in terms of `map2` to obtain the identity laws for `map2` and `pure`:

```
map2(pure(a), cont)(g) = cont.map { y ⇒ g(a, y) }
map2(cont, pure(b))(g) = cont.map { x ⇒ g(x, b) }
```

10.2.6 Deriving the laws for `zip`: naturality law

- The laws for `map2` in a short notation; here $f \otimes g \equiv \{a \times b \Rightarrow f(a) \times g(b)\}$

$$\begin{aligned}
\text{fmap2}\left(g^{A \times B \Rightarrow C}\right)\left(f^\dagger q_1 \times q_2\right) &= \text{fmap2}\left(\left(f \otimes \text{id}\right) ; g\right)(q_1 \times q_2) \\
\text{fmap2}\left(g^{A \times B \Rightarrow C}\right)\left(q_1 \times f^\dagger q_2\right) &= \text{fmap2}\left(\left(\text{id} \otimes f\right) ; g\right)(q_1 \times q_2) \\
\text{fmap2}\left(g_{1,2,3}\right)\left(q_1 \times \text{fmap2}\left(\text{id}\right)(q_2 \times q_3)\right) &= \text{fmap2}\left(g_{1,2,3}\right)\left(\text{fmap2}\left(\text{id}\right)(q_1 \times q_2) \times q_3\right) \\
\text{fmap2}\left(g^{A \times B \Rightarrow C}\right)\left(\text{pure } a^A \times q_2^{F^B}\right) &= (b \Rightarrow g(a \times b))^\dagger q_2 \\
\text{fmap2}\left(g^{A \times B \Rightarrow C}\right)\left(q_1^{F^A} \times \text{pure } b^B\right) &= (a \Rightarrow g(a \times b))^\dagger q_1
\end{aligned}$$

- Express `map2` through `zip`:

$$\begin{aligned} \text{fmap}_2 g^{A \times B \Rightarrow C} (q_1^{F^A} \times q_2^{F^B}) &\equiv (\text{zip} \circ g^\dagger) (q_1 \times q_2) \\ \text{fmap}_2 g^{A \times B \Rightarrow C} &\equiv \text{zip} \circ g^\dagger \end{aligned}$$

- Combine the two naturality laws into one by using two functions f_1, f_2 :

$$\begin{aligned} (f_1^\dagger \otimes f_2^\dagger) \circ \text{fmap}_2 g &= \text{fmap}_2 ((f_1 \otimes f_2)^\dagger \circ g) \\ (f_1^\dagger \otimes f_2^\dagger) \circ \text{zip} \circ g^\dagger &= \text{zip} \circ (f_1 \otimes f_2)^\dagger \circ g^\dagger \end{aligned}$$

- The **naturality law** for `zip` then becomes: $(f_1^\dagger \otimes f_2^\dagger) \circ \text{zip} = \text{zip} \circ (f_1 \otimes f_2)^\dagger$

10.2.7 Deriving the laws for `zip`: associativity law

- Express `map2` through `zip` and substitute into the associativity law:

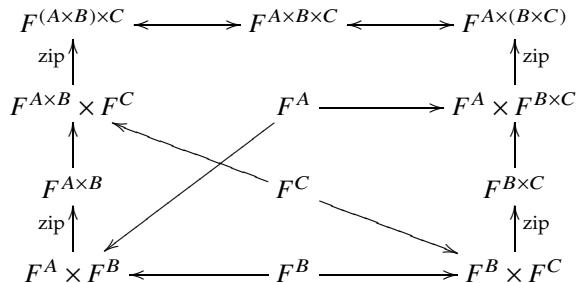
$$g_{1,2,3}^\dagger (\text{zip} (q_1 \times \text{zip} (q_2 \times q_3))) = g_{12,3}^\dagger (\text{zip} (\text{zip} (q_1 \times q_2) \times q_3))$$

- The arbitrary function g is preceded by transformations of the tuples,

$$a \times (b \times c) \equiv (a \times b) \times c \quad (\text{type isomorphism})$$

- Assume that the isomorphism transformations are applied as needed, then we may formulate the **associativity law** for `zip` more concisely:

$$\text{zip} (\text{zip} (q_1 \times q_2) \times q_3) \cong \text{zip} (q_1 \times \text{zip} (q_2 \times q_3))$$



10.2.8 Deriving the laws for `zip`: identity laws

- Identity laws seem to be complicated, e.g. the left identity:

$$g^\dagger (\text{zip} (\text{pure } a \times q)) = (b \Rightarrow g (a \times b))^\dagger q$$

- Replace `pure` by an equivalent “wrapped unit” method `wu: F[Unit]`

$$\text{wu}^{F^1} \equiv \text{pure} (1); \quad \text{pure}(a^A) = (1 \Rightarrow a)^\dagger \text{wu}$$

Then the left identity law can be simplified using left naturality:

$$g^\dagger (\text{zip} ((1 \Rightarrow a)^\dagger \text{wu} \times q)) = g^\dagger ((1 \Rightarrow a) \otimes \text{id})^\dagger \text{zip} (\text{wu} \times q)$$

- Denote $\phi^{B \Rightarrow 1 \times B} \equiv b \Rightarrow 1 \times b$ and $\beta_a^{1 \times B \Rightarrow A \times B} \equiv (1 \Rightarrow a) \otimes \text{id}$; then the function $b \Rightarrow g (a \times b)$ can be expressed more simply as $\phi \circ \beta_a \circ g$, and the identity law becomes

$$\phi^\dagger (\beta_a^\dagger \text{zip} (\text{wu} \times q)) = (\beta_a \circ g)^\dagger (\text{zip} (\text{wu} \times q)) = (\phi \circ \beta_a \circ g)^\dagger q = (\beta_a \circ g)^\dagger (\phi^\dagger q)$$

Omitting the common prefix $(\beta_a \circ g)^\dagger$, we obtain the **left identity law**:

$$\text{zip} (\text{wu} \times q) = \phi^\dagger q$$

- * Note that ϕ^\dagger is an isomorphism between F^B and $F^{1 \times B}$

- Assume that this isomorphism is applied as needed, then we may write

$$\text{zip} (\text{wu} \times q) \cong q$$

- * Similarly, the **right identity law** can be written as $\text{zip} (q \times \text{wu}) \cong q$

10.2.9 Similarity between applicative laws and monoid laws

- Define infix syntax for `zip` and write $\text{zip}(p \times q) \equiv p \bowtie q$
 - Then the associativity and identity laws may be written as

$$\begin{aligned} q_1 \bowtie (q_2 \bowtie q_3) &\cong (q_1 \bowtie q_2) \bowtie q_3 \\ (\text{wu} \bowtie q) &\cong q \\ (q \bowtie \text{wu}) &\cong q \end{aligned}$$

These are the laws of a monoid (with some assumed transformations)

- Naturality law for `zip` written in the infix syntax:

$$f_1^\dagger q_1 \bowtie f_2^\dagger q_2 = (f_1 \otimes f_2)^\dagger (q_1 \bowtie q_2)$$

- `wu` has no laws; the naturality for `pure` follows automatically
- The laws are simplest when formulated in terms of `zip` and `wu`
 - * Naturality for `zip` will usually follow from parametricity
 - A third naturality law for `map2` follows from defining `map2` through `zip!`
- “Zippable” functors have only the associativity and naturality laws
- Applicative functors are a strict superset of monadic functors
 - * There are applicative functors that *cannot* be monads
 - * Applicative functor implementation may disagree with the monad

10.2.10 A third naturality law for `map2`

- There must be one more naturality law for `map2`
 - Transform the result of a `map2`:

<pre>(for { x ← cont1 y ← cont2 } yield g(x, y)).map(f)</pre>	<pre>for { x ← cont1 y ← cont2 } yield f(g(x, y))</pre>
---	---

- Write this in terms of `map2`, obtain a third naturality law:

$$\begin{aligned} \text{map2}(\text{cont1}, \text{cont2})(g).map(f) &= \text{map2}(\text{cont1}, \text{cont2})(g \text{ andThen } f) \\ \text{ fmap2}(g) \circ f^\dagger &= \text{fmap2}(g \circ f) \\ f^\dagger(\text{fmap2}(g)(p \times q)) &= \text{fmap2}(g \circ f)(p \times q) \end{aligned}$$

- This law automatically follows if we define `map2` through `zip`:

$$\text{fmap2}(g) \circ f^\dagger = \text{zip} \circ g^\dagger \circ f^\dagger = \text{zip} \circ (g \circ f)^\dagger$$

- Note: We always have one naturality law per type parameter

10.2.11 Applicative operation `ap` as a “lifting”

- Consider `ap` as a “lifting” since it has type $F^{A \Rightarrow B} \Rightarrow (F^A \Rightarrow F^B)$
 - A “lifting” should obey the identity and the composition laws
 - * An “identity” value of type $F^{A \Rightarrow A}$, mapped to $\text{id}^{F^A \Rightarrow F^A}$ by `ap`
 - A good candidate for that value is $\text{id}_\odot \equiv \text{pure}(\text{id}^{A \Rightarrow A})$
 - * A “composition” of an $F^{A \Rightarrow B}$ and an $F^{B \Rightarrow C}$, yielding an $F^{A \Rightarrow C}$

- We can use `map2` to implement this composition, denoted $g \odot h$:

$$g^{F \Rightarrow B} \odot h^{F \Rightarrow C} \equiv \text{fmap2}(p^{A \Rightarrow B} \times q^{B \Rightarrow C} \Rightarrow p \circ q)(g, h)$$

- What are the laws that follow for $g \odot h$ from the `map2` laws?

$$\begin{aligned} \text{id}_\odot \odot h &= h; \quad g \odot \text{id}_\odot = g \\ g^{F \Rightarrow B} \odot (h^{F \Rightarrow C} \odot k^{F \Rightarrow D}) &= (g \odot h) \odot k \\ ((x^{B \Rightarrow C} \Rightarrow f^{A \Rightarrow B} \circ x)^\dagger g^{F \Rightarrow C}) \odot h^{F \Rightarrow D} &= (x^{B \Rightarrow D} \Rightarrow f^{A \Rightarrow B} \circ x)^\dagger (g \odot h) \\ g^{F \Rightarrow B} \odot ((x^{B \Rightarrow C} \Rightarrow x \circ f^{C \Rightarrow D})^\dagger h^{F \Rightarrow C}) &= (x^{A \Rightarrow C} \Rightarrow x \circ f^{C \Rightarrow D})^\dagger (g \odot h) \end{aligned}$$

- * The first 3 laws are the identity & associativity laws of a *category*
 - The morphism type is $A \rightsquigarrow B \equiv F^{A \Rightarrow B}$, the composition is \odot
- * The last 2 laws are naturality laws, connecting `fmap` and \odot
- Therefore `ap` is a functor's “lifting” of morphisms from two categories

10.2.12 Deriving the category laws for (id_\odot, \odot)

The five laws for id_\odot and \odot follow from the five `map2` laws

- Consider $\text{id}_\odot \odot h$ and substitute the definition of \odot via `map2`, cf. slide 7: $\text{id}_\odot \odot h = \text{fmap2}(p \times q \Rightarrow p \circ q)(\text{id} \circ b \Rightarrow \text{id} \circ b)^\dagger h = h$
 - The law $g \odot \text{id}_\odot = g$ is derived similarly
 - Associativity law: $g \odot (h \odot k) = \text{fmap2}(\text{id})(g \times \text{fmap2}(\text{id})(h \times k))$ The 3rd naturality law gives: $\text{fmap2}(\text{id})(h \times k) = (\text{id})^\dagger (\text{fmap2}(\text{id})(h \times k))$, and then:

$$\begin{aligned} g \odot (h \odot k) &= \text{fmap2}(x \times (y \times z) \Rightarrow x \circ y \circ z)(g \times \text{fmap2}(\text{id})(h \times k)) \\ (g \odot h) \odot k &= \text{fmap2}((x \times y) \times z \Rightarrow x \circ y \circ z)(\text{fmap2}(\text{id})(g \times h) \times k) \end{aligned}$$

Now the associativity law for `fmap2` yields $g \odot (h \odot k) = (g \odot h) \odot k$

- Derive naturality laws for \odot from the three `map2` naturality laws: $((x \Rightarrow f \circ x)^\dagger g) \odot h = \text{fmap2}(\text{id})(x \Rightarrow f \circ x)^\dagger h$
 $\text{fmap2}(x \times y \Rightarrow f \circ x \circ y)(g \times h) = (x \Rightarrow f \circ x)^\dagger (\text{fmap2}(\text{id})(g \times h)) = (x \Rightarrow f \circ x)^\dagger (g \odot h)$
- The law is $g \odot (x \Rightarrow f)^\dagger h = (x \Rightarrow f)^\dagger (g \odot h)$ is derived similarly

10.2.13 Deriving the functor laws for `ap`

Now that we established the laws for \odot , we have `ap` laws:

$$\text{ap}^{[B, Z]} : F^{B \Rightarrow Z} \Rightarrow F^B \Rightarrow F^Z = \text{fmap}_2(\text{id}^{(B \Rightarrow Z) \Rightarrow (B \Rightarrow Z)})$$

Identity law: $\text{ap}(\text{id}_\odot) = \text{id}^{F^A \Rightarrow F^A}$

- Derivation: $\text{ap}(\text{id}_\odot^{F^A \Rightarrow A})(q^{F^A}) = \text{fmap}_2(\text{id}^{(A \Rightarrow A) \Rightarrow A \Rightarrow A})(\text{pure}(\text{id}^{A \Rightarrow A}))(q^{F^A}) = \text{fmap2}(f \times x \Rightarrow f(x))(\text{pure}(\text{id})) \times q = (x \Rightarrow \text{id}(x))^\dagger q = \text{id}^\dagger q = q$
 - Easier derivation: first, express `ap` via \odot using the isomorphisms

$$A \cong 1 \Rightarrow A; \quad F^A \cong F^{1 \Rightarrow A}$$

Then $\text{ap}(p^{F^{B \Rightarrow Z}})(q^{F^B}) \cong q^{F^{1 \Rightarrow B}} \odot p^{F^{B \Rightarrow Z}}$ and so $\text{ap}(\text{id}_\odot)(q) \cong q \odot \text{id}_\odot = q$

Composition law: $\text{ap}(g \odot h) = \text{ap}(g) \circ \text{ap}(h)$

- Derivation: use $\text{ap}(p \circ q) \cong q \odot p$ to get $\text{ap}(g \odot h)(q) \cong q \odot (g \odot h)$ while $(\text{ap}(g) \circ \text{ap}(h))q = \text{ap}(h)(\text{ap}(g)(q)) \cong \text{ap}(h)(q \odot g) \cong (q \odot g) \odot h$

10.2.14 Constructions of applicative functors

- All monadic constructions still hold for applicative functors
 - Additionally, there are some non-monadic constructions
 - $F^A \equiv 1$ (constant functor) and $F^A \equiv A$ (identity functor)
 - $F^A \equiv G^A \times H^A$ for any applicative G^A and H^A
 - but $G^A + H^A$ is in general *not* applicative
 - $F^A \equiv A + G^A$ for any applicative G^A (**free pointed** over G)
 - $F^A \equiv A + G^{F^A}$ (recursive) for *any* functor G^A (**free monad** over G)
 - $F^A \equiv H^A \Rightarrow A$ for *any* contrafunctor H^A

Constructions that do not correspond to monadic ones:

 - $F^A \equiv Z$ (constant functor, Z a monoid)
 - $F^A \equiv Z + G^A$ for any applicative G^A and monoid Z
 - $F^A \equiv G^{H^A}$ when both G and H are applicative
 - Applicative that disagrees with its monad: $F^A \equiv 1 + (1 \Rightarrow A \times F^A)$
 - Examples of non-applicative functors: $F^A \equiv (P \Rightarrow A) + (Q \Rightarrow A)$, $F^A \equiv (A \Rightarrow P) \Rightarrow Q$, $F^A \equiv (A \Rightarrow P) \Rightarrow 1 + A$

10.2.15 All non-parameterized exp-poly types are monoids

- Using known monoid constructions (Chapter 7), we can implement $X + Y$, $X \times Y$, $X \Rightarrow Y$ as monoids when X and Y are monoids
 - All primitive types have at least one monoid instance:
 - * `Int`, `Float`, `Double`, `Char`, `Boolean` are “numeric” monoids
 - * `Seq[A]`, `Set[A]`, `Map[K,V]` are set-like monoids
 - * `String` is equivalent to a sequence of integers; `Unit` is a trivial monoid
 - Therefore, all exponential-polynomial types without type parameters are monoids in at least one way
 - Example of an exponential-polynomial type without type parameters: $\text{Int} + \text{String} \times \text{String} \times (\text{Int} \Rightarrow \text{Bool}) + (\text{Bool} \times \text{String} \Rightarrow 1 + \text{String})$
 - Example of a non-monoid type with type parameters: $A \Rightarrow B$

By constructions 1, 2, 6, 7, *all* polynomial F^A with monoidal coefficients are applicative: write $F^A = Z_1 + A \times (Z_2 + A \times \dots)$ with some monoids Z_i

- Examples: $F^A = 1 + A \times A$ (this F^A cannot be a monad!)
- $F^A = A + A \times A \times Z$ where Z is a monoid (this F^A is a monad)

Previous examples of non-applicative functors are all *non-polynomial*

10.2.16 Definition and constructions of applicative contrafunctors

- The applicative functor laws, if formulated via `zip` and `wu`, do not use `map` and therefore can be formulated for contrafunctors
 - Define an **applicative contrafunctor** C^A as having `zip` and `wu`:

$$\text{zip} : C^A \times C^B \Rightarrow C^{A \times B}; \quad \text{wu} : C^1$$

- Identity and associativity laws must hold for `zip` and `wu`

- * Note: applying `contramap` to the function $a \times b \Rightarrow a$ will yield some $C^A \Rightarrow C^{A \times B}$, but this will *not* give a valid implementation of `zip`!
- Naturality must hold for `zip`, but with `contramap` instead of `map`
 - * There are no corresponding `pure` or `contraap!` But have $\forall A : C^A$

Applicative contrafunctor constructions:

1. $C^A \equiv Z$ (constant functor, Z a monoid)
 2. $C^A \equiv G^A \times H^A$ for any applicative contrafunctors G^A and H^A
 3. $C^A \equiv G^A + H^A$ for any applicative contrafunctors G^A and H^A
 4. $C^A \equiv H^A \Rightarrow G^A$ for *any functor* H^A and applicative contrafunctor G^A
 5. $C^A \equiv G^{H^A}$ if a functor G^A and contrafunctor H^A are both applicative
- All exponential-polynomial contrafunctors with monoidal coefficients are applicative! (These constructions cover all exp-poly cases.)

10.2.17 Definition and laws of profunctors

- Profunctors have the type parameter in both contravariant and covariant positions; they can have neither `map` nor `contramap`
 - Examples of profunctors: $P^A \equiv 1 + \text{Int} \times A \Rightarrow A$; $P^A \equiv A + (A \Rightarrow \text{String})$
 - Example of non-profunctor: a GADT, $F^A \equiv \text{String}^{F^{\text{Int}}} + \text{Int}^{F^1}$

```
sealed trait F[A]
final case class F1(s: String) extends F[Int]
final case class F2(i: Int) extends F[Unit]
```
 - Rigorously: P^A is a profunctor if a type function $Q^{X,Y}$ exists which is a contrafunctor in X and a functor in Y , and such that $P^A \equiv Q^{A,A}$
 - Profunctors have `xmap` of type $(A \Rightarrow B) \times (B \Rightarrow A) \Rightarrow (P^A \Rightarrow P^B)$
 - Identity law: $\text{xmap}(\text{id}, \text{id}) = \text{id}$
 - Composition law: $\text{xmap}(f_1, g_1) \circ \text{xmap}(f_2, g_2) = \text{xmap}(f_1 \circ f_2, g_2 \circ g_1)$
 - * both `xmap` and the laws follow from the functor and contrafunctor laws
 - All exp-poly type constructors are profunctors since the type parameter is always in either a covariant or a contravariant position

10.2.18 Definition and constructions of applicative profunctors

- Definition of applicative profunctor: has `zip` and `wu` with the laws
 - There is no corresponding `ap!` But have `pure : A \Rightarrow P^A`

Applicative profunctors admit all previous constructions, and in addition:

1. $P^A \equiv G^A \times H^A$ for any applicative profunctors G^A and H^A
2. $P^A \equiv Z + G^A$ for any applicative profunctor G^A and monoid Z
3. $P^A \equiv A + G^A$ for any applicative profunctor G^A
4. $P^A \equiv F^A \Rightarrow Q^A$ for *any functor* F^A and applicative profunctor Q^A
 - Non-working construction: $P^A \equiv H^A \Rightarrow A$ for a profunctor H^A
5. $P^A \equiv G^{H^A}$ for a functor G^A and a profunctor H^A , both applicative

10.2.19 Commutative applicative functors

- The monoidal operation \oplus can be **commutative** w.r.t. its arguments:

$$x \oplus y = y \oplus x$$

- Applicative operation `zip` can be **commutative** w.r.t. its arguments:

$$(a \times b \Rightarrow b \times a)^\dagger (fa \bowtie fb) = fb \bowtie fa$$

or $fa \bowtie fb \cong fb \bowtie fa$, implicitly using the isomorphism $a \times b \Rightarrow b \times a$

- Applicative functor is commutative if the second effect is independent of the first effect (not only of the first value)
- Examples:
 - List is commutative; applicative parsers are not
 - If defined through the monad instance, `zip` is usually not commutative
 - All polynomial functors with *commutative* monoidal coefficients are commutative applicative functors
- Most applicative constructions preserve commutativity
- The same applies to applicative contrafunctors and profunctors
- Commutativity makes proving associativity easier:

$$(fa \bowtie fb) \bowtie fc \cong fc \bowtie (fb \bowtie fa)$$

so it's sufficient to swap fa and fc and show equivalence

10.2.20 Categorical overview of “regular” functor classes

The “liftings” show the types of category’s morphisms

class name	lifting’s name and type signature	category’s morphism
functor	fmap : $(A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$	$A \Rightarrow B$
filterable	fmapOpt : $(A \Rightarrow 1 + B) \Rightarrow F^A \Rightarrow F^B$	$A \Rightarrow 1 + B$
monad	flm : $(A \Rightarrow F^B) \Rightarrow F^A \Rightarrow F^B$	$A \Rightarrow F^B$
applicative	ap : $F^{A \Rightarrow B} \Rightarrow F^A \Rightarrow F^B$	$F^{A \Rightarrow B}$
contrafunctor	contramap : $(B \Rightarrow A) \Rightarrow F^A \Rightarrow F^B$	$B \Rightarrow A$
profunctor	xmap : $(A \Rightarrow B) \times (B \Rightarrow A) \Rightarrow F^A \Rightarrow F^B$	$(A \Rightarrow B) \times (B \Rightarrow A)$
contra-filterable	contramapOpt : $(B \Rightarrow 1 + A) \Rightarrow F^A \Rightarrow F^B$	$B \Rightarrow 1 + A$
Not yet considered:		
comonad	coflm : $(F^A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$	$F^A \Rightarrow B$

Need to define each category’s composition and identity morphism

Then impose the category laws, the naturality laws, and the functor laws

- Obtained a systematic picture of the “regular” type classes
 - Some classes (e.g. contra-applicative) aren’t covered by this scheme
 - Some of the possibilities (e.g. “contramonomad”) don’t actually work out

10.2.21 Exercises

1. Show that pure will be automatically a natural transformation when it is defined using wu as shown in the slides.
 - a) Use naturality of pure to show that $\text{pure } f \circ \text{pure } g = \text{pure } (f \circ g)$
 - b) Show that $F^A \equiv (A \Rightarrow Z) \Rightarrow (1 + A)$ is a functor but not applicative.
 - c) Show that P^S is a monoid if S is a monoid and P is any applicative functor, contrafunctor, or profunctor.
 - d) Implement an applicative instance for $F^A = 1 + \text{Int} \times A + A \times A \times A$.
 - e) Using applicative constructions, show without lengthy proofs that $F^A = G^A + H^{G^A}$ is applicative if G and H are applicative functors.
 - f) Explicitly implement contrafunctor construction 2 and prove the laws.
 - g) For any contrafunctor H^A , construction 5 says that $F^A \equiv H^A \Rightarrow A$ is applicative. Implement the code of $\text{zip}(fa, fb)$ for this construction.
 - h) Show that the recursive functor $F^A \equiv 1 + G^{A \times F^A}$ is applicative if G^A is applicative and wu_F is defined recursively as $0 + \text{pure}_G(1 \times \text{wu}_F)$.
 - i) Explicitly implement profunctor construction 5 and prove the laws.
 - j) Prove rigorously that all exponential-polynomial type constructors are profunctors.
 - k) Implement profunctor and applicative instances for $P^A \equiv A + Z \times G^A$ where G^A is a given applicative profunctor and Z is a monoid.
 - l) Show that, for any profunctor P^A , one can implement a function of type $A \Rightarrow P^B \Rightarrow P^{A \times B}$ but not of type $A \Rightarrow P^B \Rightarrow P^A$.

10.3 Practical use

10.3.1 Discussion

10.4 Laws and structure

11 Traversable functors and profunctors

11.1 Slides

11.1.1 Motivation for the `traverse` operation

Consider data of type List^A and processing $f : A \Rightarrow \text{Future}^B$

Typically, we want to wait until the entire data set is processed

What we need is $\text{List}^A \Rightarrow (A \Rightarrow \text{Future}^B) \Rightarrow \text{Future}^{\text{List}^B}$

Generalize: $L^A \Rightarrow (A \Rightarrow F^B) \Rightarrow F^{L^B}$ for some type constructors F, L

This operation is called `traverse`

How to implement it: for example, a 3-element list is $A \times A \times A$

Consider $L^A \equiv A \times A \times A$, apply map f and get $F^B \times F^B \times F^B$

We will get $F^{L^B} \equiv F^{B \times B \times B}$ if we can apply `zip` as $F^B \times F^B \Rightarrow F^{B \times B}$

So we need to assume that F is applicative

In Scala, we have `Future.traverse()` that assumes L to be a sequence

This is the easy-to-remember example that fixes the requirements

Questions:

Which functors L can have this operation?

Can we express `traverse` through a simpler operation?

What are the required laws for `traverse`?

What about contrafunctors or profunctors?

11.1.2 Deriving the `sequence` operation

The type signature of `traverse` is a complicated “lifting”

A “lifting” is often equivalent to a simpler natural transformation

To derive it, ask: what is missing from `fmap` to do the job of `traverse`?

$$\text{fmap}_L : (A \Rightarrow F^B) \Rightarrow L^A \Rightarrow L^{F^B}$$

We need F^{L^B} , but the `traverse` operation gives us L^{F^B} instead

What's missing is a natural transformation `sequence` : $L^{F^B} \Rightarrow F^{L^B}$

The functions `traverse` and `sequence` are computationally equivalent:

$$\text{trav } f \xrightarrow{A \Rightarrow F^B} = \text{fmap}_L f \circ \text{seq}$$

$$\begin{array}{ccc} & L^{F^B} & \\ \text{fmap}_L f \xrightarrow{A \Rightarrow F^B} & \nearrow \text{seq} & \\ L^A & \xrightarrow{\text{trav } f \xrightarrow{A \Rightarrow F^B}} & F^{L^B} \end{array}$$

Here F is an *arbitrary* applicative functor

Keep in mind the example `Future.sequence` : $\text{List}^{\text{Future}^X} \Rightarrow \text{Future}^{\text{List}^X}$

Examples: $L^A \equiv A \times A \times A$; $L^A = \text{List}^A$; finite trees

Non-traversable: $L^A \equiv R \Rightarrow A$; lazy list (“infinite product”)

Note: We *cannot have* the opposite transformation $F^{L^B} \Rightarrow L^{F^B}$

11.1.3 Polynomial functors are traversable

Generalize from the example $L^A \equiv A \times A \times A$ to other polynomials

Polynomial functors have the form

$$L^A \equiv Z \times A \times \dots \times A + Y \times A \times \dots \times A + \dots + Q \times A + P$$

To implement $\text{seq} : L^{F^B} \Rightarrow F^{L^B}$, consider monomial $L^A \equiv Z \times A \times \dots \times A$

We have $L^{F^B} = Z \times F^B \times \dots \times F^B$; apply `zip` and get $Z \times F^{B \times \dots \times B}$

Lift Z into the functor F using $Z \Rightarrow F^A \Rightarrow F^{Z \times A}$ (or with $F.\text{pure}$)

The result is $F^{Z \times B \times \dots \times B} \equiv F^{L^B}$

For a polynomial L^A , do this to each monomial, then lift to F^{L^B}

Note that we could apply `zip` in various different orders

The traversal order is arbitrary, may be application-specific

Non-polynomial functors are not traversable (see Bird et al., 2013)

Example: $L^A \equiv E \Rightarrow A; F^A \equiv 1 + A$; can't have $\text{seq} : L^{F^B} \Rightarrow F^{L^B}$

All polynomial functors are traversable, and usually in several ways

It is still useful to have a type class for traversable functors

11.1.4 Motivation for the laws of the `traverse` operation

The “law of traversals”{} paper (2012) argues that `traverse` should “visit each element” of the container L^A exactly once, and evaluate each corresponding “effect” F^B exactly once; then they formulate the laws

To derive the laws, use the “lifting” intuition for `traverse`,

$$\text{trav} : (A \Rightarrow F^B) \Rightarrow L^A \Rightarrow F^{L^B}$$

Look for “identity” and “composition” laws:

“Identity” as `pure` : $A \Rightarrow F^A$ must be lifted to `pure` : $L^A \Rightarrow F^{L^A}$

“Identity” as $\text{id} \xrightarrow{A \Rightarrow A}$ with $F^A \equiv A$ (identity functor) lifted to $\text{id} \xrightarrow{L^A \Rightarrow L^A}$

“Compose” $f : A \Rightarrow F^B$ and $g : B \Rightarrow G^C$ to get $h : A \Rightarrow F^{G^C}$, where F, G are applicative; a traversal with h maps L^A to $F^{G^{L^C}}$ and must be equal to the composition of traversals with f and then with g^{F^\uparrow}

Questions:

Are the laws for the `sequence` operation simpler?

Are all these laws independent?

What functors L satisfy these laws for all applicative functors F ?

11.1.5 Formulation of the laws for `traverse`

Identity law: For any applicative functor F ,

$$\text{trav}(\text{pure}) = \text{pure}$$

$$L^A \xrightarrow[\text{trav}(\text{pure} \xrightarrow{A \Rightarrow F^A})]{\text{pure} \xrightarrow{L^A \Rightarrow F^{L^A}}} F^{L^A}$$

Second identity law: $\text{trav}^{\text{Id}}(\text{id}^A) = \text{id}^{L^A}$ is a consequence with $F = \text{Id}$

So, we need only one identity law

Composition law: For any $f \xrightarrow{A \Rightarrow F^B}$ and $g \xrightarrow{B \Rightarrow G^C}$, & applicative F and G ,

$$\text{trav } f ; (\text{trav } g)^{F^\uparrow} = \text{trav } (f ; g^{F^\uparrow})$$

$$\begin{array}{ccc}
 & F^{L^B} & \\
 \text{trav}^F f \xrightarrow{A \Rightarrow F^B} & \nearrow \text{fmap}_F (\text{trav}^G g) \xrightarrow{L^B \Rightarrow G^{L^C}} & \\
 L^A & \xrightarrow{\quad \text{trav}^{FG} h \xrightarrow{A \Rightarrow FG^C} \quad} & F^{G^{L^C}}
 \end{array}$$

where $h \xrightarrow{A \Rightarrow FG^C} \equiv f \circ g^F$. (Note: $H^A \equiv F^{GA}$ is applicative!)

11.1.6 Derivation of the laws for sequence

Express $\text{trav } f = f^{L^\dagger} ; \text{seq}$ and substitute into the laws for trav :

Identity law: $\text{trav}(\text{pure}) = \text{pure}^{L^\dagger} ; \text{seq} = \text{pure}$

$$\begin{array}{ccc}
 & L^{F^A} & \\
 \text{fmap}_L \text{pure}^A \xrightarrow{} & \nearrow \text{seq} & \\
 L^A & \xrightarrow{\text{pure}^{L^A}} & F^{L^A}
 \end{array}$$

Naturality law: $\text{seq} ; g^{F^\dagger L^\dagger} = g^{L^\dagger F^\dagger} ; \text{seq}$ with $g \xrightarrow{A \Rightarrow B}$, mapping $L^{F^A} \Rightarrow F^{L^B}$

Composition law:

$$\begin{aligned}
 \text{trav } f ; (\text{trav } g)^{F^\dagger} &= f^{L^\dagger} ; \text{seq} ; (g^{L^\dagger} ; \text{seq})^{F^\dagger} \\
 &= f^{L^\dagger} ; \text{seq} ; g^{L^\dagger F^\dagger} ; \text{seq}^{F^\dagger} = f^{L^\dagger} ; g^{F^\dagger L^\dagger} ; \text{seq} ; \text{seq}^{F^\dagger} \\
 \text{trav } (f ; g^{F^\dagger})^{L^\dagger} ; \text{seq} &= f^{L^\dagger} ; g^{F^\dagger L^\dagger} ; \text{seq}
 \end{aligned}$$

Now omit the common prefix $f \cdots ; g \cdots$ and obtain: $\text{seq} ; \text{seq}^{F^\dagger} = \text{seq}$

$$\begin{array}{ccc}
 & F^{L^G A} & \\
 \text{seq}^F \xrightarrow{} & \nearrow (\text{seq}^G)^{F^\dagger} & \\
 L^{F^G A} & \xrightarrow{\text{seq}^{F^G ?}} & F^{G^{L^A}}
 \end{array}$$

11.1.7 Constructions of traversable and bitraversable functors

Constructions of traversable functors:

$L^A \equiv Z$ (constant functor) and $L^A \equiv A$ (identity functor)

$L^A \equiv G^A \times H^A$ for any traversable G^A and H^A

$L^A \equiv G^A + H^A$ for any traversable G^A and H^A

$L^A \equiv S^{A,L^A}$ (recursive) for a bitraversable bifunctor $S^{A,B}$

If L^A is infinite, laws will appear to hold but `seq` will not terminate

A bifunctor $S^{A,B}$ is **bitraversable** if `biseq` exists such that

$$\text{biseq} : S^{F^A, F^B} \Rightarrow F^{S^{A,B}}$$

for any applicative functor F ; the analogous laws must hold

Constructions of bitraversable bifunctors:

$S^{A,B} \equiv Z$, $S^{A,B} \equiv A$, and $S^{A,B} = B$

$S^{A,B} \equiv G^{A,B} \times H^{A,B}$ for any bitraversable G and H

$S^{A,B} \equiv G^{A,B} + H^{A,B}$ for any bitraversable G and H

All polynomial bifunctors are bitraversable

All polynomial functors, including recursive functors, are traversable

11.1.8 Foldable functors: traversing with respect to a monoid

Take $F^A \equiv Z$ where Z is a monoid

The `zip` operation is the monoid operation \oplus

The type signature of `traverse` becomes $(A \Rightarrow Z) \Rightarrow L^A \Rightarrow Z$

This method is called `foldMap`

The type signature of `seq` becomes $L^Z \Rightarrow Z$

This is called `mconcat` – combines all values in L^Z with Z 's \oplus

It is convenient to define the `Foldable` type class

But it has no laws any more

All traversable functors are also foldable

The `foldLeft` method can be defined via `foldMap` with $Z \equiv (B \Rightarrow B)$:

$$\text{foldl} : (A \Rightarrow B \Rightarrow B) \Rightarrow L^A \Rightarrow B \Rightarrow B$$

11.1.9 Traversable contrafunctors and profunctors are not useful

Traversing profunctors with respect to functors F : effects of F are ignored

All contrafunctors C^A are traversable w.r.t. applicative profunctors F^A ,

$$\text{seq} : C^{F^A} \Rightarrow F^{C^A} \equiv \text{pure}^{C \downarrow} ; \text{pure}$$

$$C^{F^A} \xrightarrow{\text{cmap}_C \text{pure}_F^A} C^A \xrightarrow{\text{pure}_F^{C^A}} F^{C^A}$$

But not profunctors that are neither functors nor contrafunctors

Counterexample: $P^A \equiv A \Rightarrow A$; need $\text{seq} : (F^A \Rightarrow F^A) \Rightarrow F^{A \Rightarrow A}$; we can't get an $A \Rightarrow A$, so the only implementation is to return $\text{pure}_F(\text{id})$, which ignores its argument and so will fail the identity law

Traversing profunctors L with respect to profunctors F : effects are ignored

Counterexample 1: contrafunctor $L^A \equiv A \Rightarrow R$ and contrafunctor $F^A \equiv A \Rightarrow S$, a `seq` of type $L^{F^A} \Rightarrow F^{L^A}$ must return $1 + 0$

Counterexample 2: contrafunctor $F^A \equiv (R \Rightarrow A) \Rightarrow S$ and functor $L^A \equiv 1 + A$; `seq` must return $1 + 0$
So, the result is trivial and probably not useful

Laws of traversables allow ignoring the effects of F

11.1.10 Examples of usage

Convert foldable data structures to list

Fold a tree to aggregate data

Decorate a tree with Depth-First traversal order labels

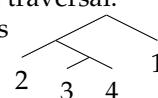
Implement `scanMap` and `scanLeft` as `traverse` with a state monad

Traversal for a non-monadic "rigid" tree $T^A \equiv A + T^{A \times A}$

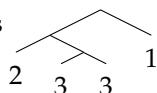
The corresponding construction is $L^A \equiv S^{A, L^{R^A}}$ where R is applicative and traversable and S is bitraversable

What *cannot* be implemented as a traversal:

Breadth-first traversal for a tree as



Depth labeling of a tree as



11.1.11 Naturality with respect to applicative functor as parameter

The `traverse` method must be “generic in the functor F ”:

$$\text{trav}^{F,A,B} : (A \Rightarrow F^B) \Rightarrow L^A \Rightarrow F^{L^B}$$

Which means: The code of `traverse` can only use `pure` and `zip` from F

A functor F^A is “generic in A ”: have $\text{fmap} : (A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$

“Generic in F ” means mapping $(F \Rightarrow G) \Rightarrow \text{trav}^F \Rightarrow \text{trav}^G$ in some way

Mathematical formulation:

For any natural transformation $F^A \Rightarrow G^A$ between applicative functors F and G such that $F.\text{pure}$ and $F.\text{zip}$ are mapped into $G.\text{pure}$ and $G.\text{zip}$, the result of transforming trav^F is trav^G

Such a natural transformation is a morphism of applicative functors

Category theory can describe $(F \Rightarrow G) \Rightarrow \text{trav}^F \Rightarrow \text{trav}^G$ as a “lifting”

Use a more general definition of category than what we had so far (morphisms between type constructors)

11.1.12 Exercises

Show that any traversable functor L admits a method

$$\text{consume} : (L^A \Rightarrow B) \Rightarrow L^{F^A} \Rightarrow F^B$$

for any applicative functor F . Show that `traverse` and `consume` are equivalent.

Show that $\text{seq} : L^{F^A} \Rightarrow F^{L^A} = \text{id}$ if we choose $F^A \equiv A$ as the identity functor.

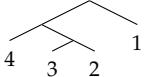
Show that the identity law is not satisfied by an implementation of $\text{seq} : L^{F^A} \Rightarrow F^{L^A}$ for $F^A \equiv 1 + A$ when seq always returns an empty option.

Show that $K^A \equiv G^{H^A}$ is traversable if G and H are traversable.

Show that all the bitraversable laws hold for the bifunctor $S^{A,B} \equiv A \times B$.

For the tree-like type defined as $T^A \equiv 1 + A \times T^A \times T^A$, define a traversable instance. Verify that the laws hold, using a suitable bifunctor $S^{X,Y}$.

For a tree type T^A of your choice, implement a traversable instance for right-to-left traversal order, and test that the tree is decorated with labels as



Implement a traversable instance for a “rose tree” $T^A \equiv A + \text{List}^{T^A}$. Represent T^A via a suitable bifunctor $S^{X,Y}$ and show that S is bitraversable (use constructions).

Is the recursive type constructor $L^A \equiv A + L^{\text{List}^A}$ traversable? Explain what sort of data container it is.

11.2 Discussion

Part III

Advanced level

12 “Free” type constructions

12.1 Slides

12.1.1 The interpreter pattern I. Expression trees

Main idea: Represent a program as a data structure, run it later

Example: a simple DSL for complex numbers

```
val a = "1+2*i".toComplex
val b = a * "3-4*i".toComplex
b.conj
```

```
Conj(
    Mul(
        Str("1+2*i"), Str("3-4*i")
    )
)
```

Unevaluated operations `Str`, `Mul`, `Conj` are defined as case classes:

```
sealed trait Prg
case class Str(s: String) extends Prg
case class Mul(p1: Prg, p2: Prg) extends Prg
case class Conj(p: Prg) extends Prg
```

An *interpreter* will “run” the program and return a complex number

```
def run(prg: Prg): (Double, Double) = ...
```

Benefits: programs are data, can compose & transform before running

Shortcomings: this DSL works only with simple expressions

Cannot represent variable binding and conditional computations

Cannot use any non-DSL code (e.g. a numerical algorithms library)

12.1.2 The interpreter pattern II. Variable binding

A DSL with variable binding and conditional computations

Example: imperative API for reading and writing files

Need to bind a *non-DSL variable* to a value computed by DSL

Later, need to use that non-DSL variable in DSL expressions

The rest of the DSL program is a (Scala) function of that variable

```
val p = path("/file")
val str: String = read(p)
if (str.nonEmpty)
    read(path(str))
else "Error: empty path"
```

```
Bind(
    Read(Path(Literal("/file"))),
    { str => // read value 'str'
        if (str.nonEmpty)
            Read(Path(Literal(str)))
        else Literal("Error: empty path")
    }
)
```

Unevaluated operations are implemented via case classes:

```
sealed trait Prg
case class Bind(p: Prg, f: String ⇒ Prg) extends Prg
case class Literal(s: String) extends Prg
case class Path(s: Prg) extends Prg
case class Read(p: Prg) extends Prg
```

Interpreter: `def run(prg: Prg): String = ...`

12.1.3 The interpreter pattern III. Type safety

So far, the DSL has no type safety: every value is a `Prg`

We want to avoid errors, e.g. `Read(Read(...))` should not compile

Let `Prg[A]` denote a DSL program returning value of type `A` when run:

```
sealed trait Prg[A]
```

```

case class Bind(p: Prg[String], f: String ⇒ Prg[String])
extends Prg[String]
case class Literal(s: String) extends Prg[String]
case class Path(s: Prg[String]) extends Prg[nio.file.Path]
case class Read(p: Prg[nio.file.Path]) extends Prg[String]
Interpreter: def run(prg: Prg[String]): String = ...

```

Our example DSL program is type-safe now:

```

val prg: Prg[String] = Bind(
  Read(Path(Literal("/file"))),
  { str: String ⇒
    if (str.nonEmpty)
      Read(Path(Literal(str)))
    else Literal("Error: empty path")
  })

```

12.1.4 The interpreter pattern IV. Cleaning up the DSL

Our DSL so far:

```

sealed trait Prg[A]
case class Bind(p: Prg[String], f: String ⇒ Prg[String])
extends Prg[String]
case class Literal(s: String) extends Prg[String]
case class Path(s: Prg[String]) extends Prg[nio.file.Path]
case class Read(p: Prg[nio.file.Path]) extends Prg[String]

```

Problems with this DSL:

Cannot use `Read(p: nio.file.Path)`, only `Read(p: Prg[nio.file.Path])`

Cannot bind variables or return values other than `String`

To fix these problems, make `Literal` a fully parameterized operation and replace `Prg[A]` by `A` in case class arguments

```

sealed trait Prg[A]
case class Bind[A, B](p: Prg[A], f: A ⇒ Prg[B]) extends Prg[B]
case class Literal[A](a: A) extends Prg[A]
case class Path(s: String) extends Prg[nio.file.Path]
case class Read(p: nio.file.Path) extends Prg[String]

```

The type signatures of `Bind` and `Literal` are like `flatMap` and `pure`

12.1.5 The interpreter pattern V. Define Monad-like methods

We can actually define the methods `map`, `flatMap`, `pure`:

```

sealed trait Prg[A] {
  def flatMap[B](f: A ⇒ Prg[B]): Prg[B] = Bind(this, f)
  def map[B](f: A ⇒ B): Prg[B] = flatMap(this, f andThen Prg.pure)
}
object Prg { def pure[A](a: A): Prg[A] = Literal(a) }

```

These methods don't run anything, only create unevaluated structures

DSL programs can now be written as functor blocks and composed:

```

def readPath(p: String): Prg[String] = for {
  path ← Path(p)
  str ← Read(path)
} yield str

```

```

val prg: Prg[String] = for {
  str ← readPath("/file")
  result ← if (str.nonEmpty)
    readPath(str)
  else Prg.pure("Error: empty path")
} yield result

```

Interpreter: `def run[A](prg: Prg[A]): A = ...`

12.1.6 The interpreter pattern VI. Refactoring to an abstract DSL

Write a DSL for complex numbers in a similar way:

```
sealed trait Prg[A] { def flatMap ... } // no code changes
case class Bind[A, B](p: Prg[A], f: A⇒Prg[B]) extends Prg[B]
case class Literal[A](a: A) extends Prg[A]
type Complex = (Double, Double) // custom code starts here
case class Str(s: String) extends Prg[Complex]
case class Mul(c1: Complex, c2: Complex) extends Prg[Complex]
case class Conj(c: Complex) extends Prg[Complex]
```

Refactor this DSL to separate common code from custom code:

```
sealed trait DSL[F[_], A] { def flatMap ... } // no code changes
type Prg[A] = DSL[F, A] // just for convenience
case class Bind[A, B](p: Prg[A], f: A⇒Prg[B]) extends Prg[B]
case class Literal[A](a: A) extends Prg[A]
case class Ops[A](f: F[A]) extends Prg[A] // custom operations here
```

Interpreter is parameterized by a “value extractor” $\text{Ex}^F \equiv \forall A. (F^A \Rightarrow A)$

```
def run[F[_], A](ex: Ex[F])(prg: DSL[F, A]): A = ...
```

The constructor `DSL[F[_], A]` is called a **free monad** over `F`

12.1.7 The interpreter pattern VII. Handling errors

To handle errors, we want to evaluate `DSL[F[_], A]` to `Either[Err, A]`

Suppose we have a value extractor of type $\text{Ex}^F \equiv \forall A. (F^A \Rightarrow \text{Err} + A)$

The code of the interpreter is almost unchanged:

```
def run[F[_], A](extract: Ex[F])(prg: DSL[F, A]): Either[Err, A] =
  prg match {
    case b: Bind[F, _, A] ⇒ b match { case Bind(p, f) ⇒
      run(extract)(p).flatMap(f andThen run(extract))
    } // Here, the .flatMap is from Either.
    case Literal(a) ⇒ Right(a) // pure: A ⇒ Err + A
    case Ops(f) ⇒ extract(f)
  }
```

The code of `run` only uses `flatMap` and `pure` from `Either`

We can generalize to any other monad M^A instead of `Either[Err, A]`

The resulting construction:

Start with an “operations type constructor” F^A (often not a functor)

Use $\text{DSL}^{F,A}$ and interpreter $\text{run}^{M,A} : (\forall X. F^X \Rightarrow M^X) \Rightarrow \text{DSL}^{F,A} \Rightarrow M^A$

Create a DSL program `prg : DSLF,A` and an extractor `exX : FX ⇒ MX`

Run the program with the extractor: `run(ex)(prg)`; get a value M^A

12.1.8 The interpreter pattern VIII. Monadic DSLs: summary

Begin with a number of operations, which are typically functions of fixed known types such as $A_1 \Rightarrow B_1, A_2 \Rightarrow B_2$ etc.

Define a type constructor (typically not a functor) encapsulating all the operations as case classes, with or without type parameters

```
sealed trait F[A]
case class Op1(a1: A1) extends F[B1]
case class Op2(a2: A2) extends F[B2]
```

Use `DSL[F, A]` with this `F` to write monadic DSL programs `prg: DSL[F, A]`

Choose a target monad `M[A]` and implement an extractor `ex: F[A] ⇒ M[A]`

Run the program with the extractor, `val res: M[A] = run(ex)(prg)`

Further directions (out of scope for this chapter):

May choose another monad `N[A]` and use interpreter `M[A] ⇒ N[A]`

E.g. transform into another monadic DSL to optimize, test, etc.

Since `DSL[F, A]` has a monad API, we can use monad transformers on it

Can combine two or more DSLs in a disjunction: `DSLF+G+H,A`

12.1.9 Monad laws for DSL programs

Monad laws hold for DSL programs only after evaluating them

Consider the law $\text{flm}(\text{pure}) = \text{id}$; both functions $\text{DSL}^{F,A} \Rightarrow \text{DSL}^{F,A}$

Apply both sides to some $\text{prg} : \text{DSL}^{F,A}$ and get the new value

`prg.flatMap(pure) == Bind(prg, a => Literal(a))`

This new value is *not equal* to `prg`, so this monad law fails!

Other laws fail as well because operations never reduce anything

After interpreting this program into a target monad M^A , the law holds:

```
run(ex)(prg).flatMap((a => Literal(a)) andThen run(ex))
== run(ex)(prg).flatMap(a => run(ex)(Literal(a)))
== run(ex)(prg).flatMap(a => pure(a))
== run(ex)(prg)
```

Here we have assumed that the laws hold for M^A

All other laws also hold after interpreting into a lawful monad M^A

The monad law violations are “not observable”

12.1.10 Free constructions in mathematics: Example I

Consider the Russian letter Π (tsè) and the Chinese word 水 (shui)

We want to *multiply* Π by 水. Multiply how?

Say, we want an associative (but noncommutative) product of them
So we want to define a *semigroup* that *contains* Π and 水 as elements
while we still know nothing about Π and 水

Consider the set of all *unevaluated expressions* such as $\Pi \cdot \text{水} \cdot \text{水} \cdot \Pi \cdot \text{水}$

Here $\Pi \cdot \text{水}$ is different from $\text{水} \cdot \Pi$ but $(a \cdot b) \cdot c = a \cdot (b \cdot c)$

All these expressions form a **free semigroup** generated by Π and 水

This is the most unrestricted semigroup that contains Π and 水

Example calculation: $(\text{水} \cdot \text{水}) \cdot (\Pi \cdot \text{水}) \cdot \Pi = \text{水} \cdot \text{水} \cdot \Pi \cdot \text{水} \cdot \Pi$

How to represent this as a data type:

Tree encoding: the full expression tree: $((\text{水}, \text{水}), (\Pi, \text{水})), \Pi$

Implement the operation $a \cdot b$ as pair constructor (easy)

Reduced encoding, as a “smart” structure: `List(水,水,Π,水,Π)`

Implement $a \cdot b$ by concatenating the lists (more expensive)

12.1.11 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):

$$\mathbf{u}_1 \otimes \mathbf{v}_1 + (\mathbf{u}_2 \otimes \mathbf{v}_2 + \mathbf{u}_3 \otimes \mathbf{v}_3) = (\mathbf{u}_1 \otimes \mathbf{v}_1 + \mathbf{u}_2 \otimes \mathbf{v}_2) + \mathbf{u}_3 \otimes \mathbf{v}_3$$

$$\mathbf{u} \otimes (a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2) = a_1 (\mathbf{u} \otimes \mathbf{v}_1) + a_2 (\mathbf{u} \otimes \mathbf{v}_2)$$

$$(a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2) \otimes \mathbf{u} = a_1 (\mathbf{v}_1 \otimes \mathbf{u}) + a_2 (\mathbf{v}_2 \otimes \mathbf{u})$$

We have such a product for 3-dimensional vectors only; ignore that

Consider *unevaluated expressions* of the form $\mathbf{u}_1 \otimes \mathbf{v}_1 + \mathbf{u}_2 \otimes \mathbf{v}_2 + \dots$

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

12.1.12 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: $\text{FSIS} \equiv \text{Int} + \text{String} + \text{FSIS} \times \text{FSIS}$

For a semigroup S and given $\text{Int} \Rightarrow S$ and $\text{String} \Rightarrow S$, map $\text{FSIS} \Rightarrow S$

Simplify and generalize this construction by setting $Z = \text{Int} + \text{String}$

The tree encoding is $\text{FS}^Z \equiv Z + \text{FS}^Z \times \text{FS}^Z$

```
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 $\text{FSR}^Z \equiv Z \times \text{List}^Z$ (non-empty list of Z 's)

`x |+| y` requires concatenating the lists, but `run()` is faster

12.1.13 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 object 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: $\text{FM}^Z \equiv 1 + Z + \text{FM}^Z \times \text{FM}^Z$

For a monoid M and given $Z \Rightarrow M$, map $\text{FM}^Z \Rightarrow M$

```
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 $\text{FMR}^Z \equiv \text{List}^Z$ (list of Z 's)

Implementing `|+|` requires concatenating the lists

Reduced encoding and tree encoding give identical results after `run()`

12.1.14 Mapping a free semigroup to different targets

What if we interpret FS^X into another free semigroup?

Given $Y \Rightarrow Z$, can we map $\text{FS}^Y \Rightarrow \text{FS}^Z$?

Need to map $\text{FS}^Y \equiv Y + \text{FS}^Y \times \text{FS}^Y \Rightarrow Z + \text{FS}^Z \times \text{FS}^Z$

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 $\text{FS}^X \Rightarrow \text{FS}^Y \Rightarrow \text{FS}^Z \Rightarrow S$, etc.

Functor laws hold for FS^X , so `fmap` is composable as usual

The “interpreter” commutes with `fmap` as well (naturality law):

$$\begin{array}{ccc} & \text{fmap } f: X \Rightarrow Y & \text{FS}^Y \\ \text{FS}^X & \swarrow & \searrow \text{run}^S g: Y \Rightarrow S \\ & \text{run}^S(f \circ g): X \Rightarrow S & \end{array}$$

Combine two free semigroups: FS^{X+Y} ; inject parts: $\text{FS}^X \Rightarrow \text{FS}^{X+Y}$

12.1.15 Church encoding I: Motivation

Multiple target semigroups S_i require many “extractors” $\text{ex}_i : Z \Rightarrow S_i$

Refactor extractors ex_i into evidence of a typeclass constraint on S_i

// Typeclass `ExZ[S]` has a single method, extract: $Z \Rightarrow S$.

```
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 $\forall S. (Z \Rightarrow S) \times (S \times S \Rightarrow S) \Rightarrow S$; an equivalent type is

$$\forall S. ((Z + S \times S) \Rightarrow S) \Rightarrow S$$

This is the “Church encoding” (of the free semigroup over Z)

The Church encoding is based on the theorem $A \cong \forall X. (A \Rightarrow X) \Rightarrow X$

this *resembles* the type of the continuation monad, $(A \Rightarrow R) \Rightarrow R$

but $\forall X$ makes the function fully generic, like a natural transformation

12.1.16 Church encoding II: Disjunction types

Consider the Church encoding for the disjunction type $P + Q$

The encoding is $\forall X. (P + Q \Rightarrow X) \Rightarrow X \cong \forall X. (P \Rightarrow X) \Rightarrow (Q \Rightarrow X) \Rightarrow X$

```
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

12.1.17 Church encoding III: How it works

Why is the type $\text{Ch}^A \equiv \forall X. (A \Rightarrow X) \Rightarrow X$ equivalent to the type A ?

```
trait Ch[A] { def run[X](cont: A ⇒ X): X }
```

12 “Free” type constructions

- 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)
}
```

$$\begin{array}{ccc} \text{id} : (A \Rightarrow A) & \xrightarrow{\text{ch.run}^A} & A \\ \downarrow \text{fmapReader}_A(f) & & \downarrow f \\ f : (A \Rightarrow X) & \xrightarrow{\text{ch.run}^X} & X \end{array}$$

- 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 ch^{Ch^A} , we must compute an $x:X$ given $f:A \Rightarrow X$, for any X , which requires having a value $a:A$ available

To show that $\text{ch} = \text{a2c}(\text{c2a}(\text{ch}))$, apply both sides to an $f: A \Rightarrow X$ and get $\text{ch.run}(f) = \text{a2c}(\text{c2a}(\text{ch})).\text{run}(f) = f(\text{c2a}(\text{ch})) = f(\text{ch.run}(a=>a))$

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 $\text{c2a}(\text{a2c}(a)) = \text{identity}(a) = a$

Church encoding satisfies laws: it is built up from parts of `run` method

12.1.18 Worked example III: Free functor I

The `Functor` type class has one method, `fmap`: $(Z \Rightarrow A) \Rightarrow F^Z \Rightarrow F^A$

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: $Q^A \equiv \exists Z. A \times Z$

The existential quantifier applies to the “hidden” type parameter

The constructor `qz` has type $\exists Z. (A \times Z \Rightarrow Q^A)$

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

12.1.19 Encoding with an existential type: How it works

Show that $P^A \equiv \exists Z. Z \times (Z \Rightarrow A) \cong A$

```
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 $Z \Rightarrow A$ and a Z , construct $Z \times (Z \Rightarrow A)$

Particular case: $Z \equiv A$, have $a : A$ and build $a \times \text{id} : A \Rightarrow A$

```
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 $Q^A \equiv \exists Z. F^Z \times (Z \Rightarrow A) \cong F^A$

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

12.1.20 Worked example III: Free functor II

Tree encoding of `FF` has type $\text{FF}^{F^\bullet, A} \equiv F^A + \exists Z. \text{FF}^{F^\bullet, Z} \times (Z \Rightarrow A)$

Derivation of the reduced encoding:

A value of type $\text{FF}^{F^\bullet, A}$ must be of the form

$$\exists Z_1. \exists Z_2. \dots \exists Z_n. F^{Z_n} \times (Z_n \Rightarrow Z_{n-1}) \times \dots \times (Z_2 \Rightarrow Z_1) \times (Z_1 \Rightarrow A)$$

The functions $Z_1 \Rightarrow A, Z_2 \Rightarrow Z_1$, etc., must be composed associatively

The equivalent type is $\exists Z_n. F^{Z_n} \times (Z_n \Rightarrow A)$

Reduced encoding: $\text{FreeF}^{F^\bullet, A} \equiv \exists Z. F^Z \times (Z \Rightarrow A)$

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 $F^A \cong \exists Z. F^Z \times (Z \Rightarrow A)$

Church encoding (starting from the tree encoding): $\text{FreeF}^{F^\bullet, A} \equiv \forall P^\bullet. (\forall C. (F^C + \exists Z. P^Z \times (Z \Rightarrow C)) \rightsquigarrow P^C) = P^A$

The structure of the type expression: $\forall P^\bullet. (\forall C. (\dots)^C \rightsquigarrow P^C) \Rightarrow P^A$

Cannot move $\forall C$ or $\exists Z$ to the outside of the type expression!

12.1.21 Church encoding IV: Recursive types and type constructors

Consider the recursive type $P \equiv Z + P \times P$ (tree with Z -valued leaves)

The Church encoding is $\forall X. ((Z + X \times X) \Rightarrow X) \Rightarrow X$

This is *non-recursive*: the inductive use of P is replaced by X

Generalize to recursive type $P \equiv S^P$ where S^\bullet is a “induction functor”:

The Church encoding of P is $\forall X. (S^X \Rightarrow X) \Rightarrow X$

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 $\text{Ch}^{P^\bullet, A} = \forall F^\bullet. (\forall X. P^X \Rightarrow F^X) \Rightarrow F^A$

Note: $\forall X. P^X \Rightarrow F^X$ 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: $P^A \equiv S^{P^\bullet, A}$ where $S^{P^\bullet, A}$ describes the “induction principle”

Notation: S^\bullet, A is a higher-order type function; Scala syntax: `S_[_, A]`

Example: $\text{List}^A \equiv 1 + A \times \text{List}^A \equiv S^{\text{List}^\bullet, A}$ where $S^{P^\bullet, A} \equiv 1 + A \times P^A$

The Church encoding of P^A is $\text{Ch}^{P^\bullet, A} = \forall F^\bullet. (S^\bullet \rightsquigarrow F^\bullet) \Rightarrow F^A$

The Church encoding of `List[_]` is non-recursive

12.1.22 Church encoding V: Type classes

Look at the Church encoding of the free semigroup:

$$\text{ChFS}^Z \equiv \forall X. (Z \Rightarrow X) \times (X \times X \Rightarrow X) \Rightarrow X$$

If X is constrained to the `Semigroup` typeclass, we will already have a value $X \times X \Rightarrow X$, so we can omit it: $\text{ChFS}^Z = \forall X^{\text{Semigroup}}. (Z \Rightarrow X) \Rightarrow X$

The “induction functor” for “semigroup over Z ” is $\text{SemiG}^X \equiv Z + X \times X$

So the Church encoding is $\forall X. (\text{SemiG}^X \Rightarrow X) \Rightarrow X$

Generalize to arbitrary type classes:

Type class C is defined by its operations $C^X \Rightarrow X$ (with a suitable C^\bullet)
 call C^\bullet the **method functor** of the inductive typeclass C

Tree encoding of “free C over Z ” is recursive, $\text{Free}C^Z \equiv Z + C^{\text{Free}C^Z}$

Church encoding is $\text{Free}C^Z \equiv \forall X. (Z + C^X \Rightarrow X) \Rightarrow X$

Equivalently, $\text{Free}C^Z \equiv \forall X^\bullet. (Z \Rightarrow X) \Rightarrow X$

Laws of the typeclass are satisfied automatically after “running”

Works similarly for type constructors: operations $C^{P^\bullet, A} \Rightarrow P^A$

Free typeclass C over F^\bullet is $\text{Free}C^{F^\bullet, A} \equiv \forall P^\bullet : C. (F^\bullet \rightsquigarrow P^\bullet) \Rightarrow P^A$

12.1.23 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

$\text{Free}F^\bullet, A$ 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 $\text{Free}C^{F,A}$ – “free instance of C over F ”?

For all F^\bullet , must have `wrap[A] : F^A → FreeC^{F,A}` or $F^\bullet \rightsquigarrow \text{Free}C^{F^\bullet}$

For all $M^\bullet : C$, must have `run : (F^\bullet ∼ M^\bullet) ⇒ FreeC^{F^\bullet} ∼ M^\bullet`

The laws of typeclass C must hold after interpreting into an $M^\bullet : C$

Given any `t : F^\bullet ∼ G^\bullet`, must have `fmap(t) : FreeC^{F^\bullet} ∼ FreeC^{G^\bullet}`

12.1.24 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, \dots , with type signatures $m_1 : Q_1^{P^\bullet, A} \Rightarrow P^A$, $m_2 : Q_2^{P^\bullet, A} \Rightarrow P^A$, 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

$$\text{FC}^A \equiv F^A + Q_1^{\text{FC}\bullet, A} + Q_2^{\text{FC}\bullet, A} + \dots$$

```
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

12.1.25 Properties of inductive typeclasses

If a typeclass C is inductive with methods $C^X \Rightarrow X$ then:

A free instance of C over Z can be tree-encoded as $\text{Free}C^Z \equiv Z + C^{\text{Free}C^Z}$

All inductive typeclasses have free instances, $\text{Free}C^Z$

If $P:C$ and $Q:C$ then $P \times Q$ and $Z \Rightarrow P$ also belong to typeclass C
but not necessarily $P + Q$ or $Z \times P$

Proof: can implement $(C^P \Rightarrow P) \times (C^Q \Rightarrow Q) \Rightarrow C^{P \times Q} \Rightarrow P \times Q$ and $(C^P \Rightarrow P) \Rightarrow C^{Z \Rightarrow P} \Rightarrow Z \Rightarrow P$,
but cannot implement $(\dots) \Rightarrow P + Q$

Analogous properties hold for type constructor typeclasses

Methods described as $C^{F^\bullet,A} \Rightarrow F^A$ 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 $\text{pt} : A \Rightarrow P^A$ and $\text{ex} : P^A \Rightarrow A$

Such typeclasses are not inductive

Typeclasses with methods of the form $P^A \Rightarrow \dots$ are **co-inductive**

12.1.26 Worked example IV: Free contrafunctor

Method contramap : $C^A \times (B \Rightarrow A) \Rightarrow C^B$

Tree encoding: $\text{Free}CF^{F^\bullet,B} \equiv F^B + \exists A. \text{Free}CF^{F^\bullet,A} \times (B \Rightarrow A)$

Reduced encoding: $\text{Free}CF^{F^\bullet,B} \equiv \exists A. F^A \times (B \Rightarrow A)$

A value of type $\text{Free}CF^{F^\bullet,B}$ must be of the form

$$\exists Z_1. \exists Z_2. \dots \exists Z_n. F^{Z_1} \times (B \Rightarrow Z_n) \times (Z_n \Rightarrow Z_{n-1}) \times \dots \times (Z_2 \Rightarrow Z_1)$$

The functions $B \Rightarrow Z_n$, $Z_n \Rightarrow Z_{n-1}$, etc., are composed associatively

The equivalent type is $\exists Z_1. F^{Z_1} \times (B \Rightarrow Z_1)$

The reduced encoding is non-recursive

Example: $F^A \equiv A$, “interpret” into the contrafunctor $C^A \equiv A \Rightarrow \text{String}$

`def prefixLog[A](p: A) = a => p.toString + a.toString`

If F^\bullet is already a contrafunctor then $\text{Free}CF^{F^\bullet,A} \cong F^A$

12.1.27 Worked example V: Free pointed functor

Over an arbitrary type constructor F^\bullet :

Pointed functor methods $\text{pt} : A \Rightarrow P^A$ and $\text{map} : P^A \times (A \Rightarrow B) \Rightarrow P^B$

Tree encoding: $\text{Free}PF^{F^\bullet,A} \equiv A + F^A + \exists Z. \text{Free}PF^{F^\bullet,Z} \times (Z \Rightarrow A)$

Derivation of the reduced encoding:

The tree encoding of a value $\text{Free}PF^{F^\bullet,A}$ is either

$$\exists Z_1. \exists Z_2. \dots \exists Z_n. F^{Z_n} \times (Z_n \Rightarrow Z_{n-1}) \times \dots \times (Z_2 \Rightarrow Z_1) \times (Z_1 \Rightarrow A)$$

or

$$\exists Z_1. \exists Z_2. \dots \exists Z_n. Z_n \times (Z_n \Rightarrow Z_{n-1}) \times \dots \times (Z_2 \Rightarrow Z_1) \times (Z_1 \Rightarrow A)$$

Compose all functions by associativity; one function $Z_n \Rightarrow A$ remains

The case $\exists Z_n. Z_n \times (Z_n \Rightarrow A)$ is equivalent to just A

Reduced encoding: $\text{Free}PF^{F^\bullet,A} \equiv A + \exists Z. F^Z \times (Z \Rightarrow A)$, non-recursive

This reuses the free functor as $\text{Free}PF^{F^\bullet,A} = A + \text{Free}F^{F^\bullet,A}$

If the type constructor F^\bullet is *already* a functor, $\text{Free}F^{F^\bullet,A} \cong F^A$ 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 $\text{Free}PF^{F^\bullet,A} \not\cong F^A$

only functors and contrafunctors do not change under “free”

12.1.28 Worked example VI: Free filterable functor

(See Chapter 6.) Methods:

$$\begin{aligned}\text{map} &: F^A \Rightarrow (A \Rightarrow B) \Rightarrow F^B \\ \text{mapOpt} &: F^A \Rightarrow (A \Rightarrow 1 + B) \Rightarrow F^B\end{aligned}$$

We can recover `map` from `mapOpt`, so we keep only `mapOpt`

Tree encoding: $\text{FreeFi}^{F^\bullet, A} \equiv F^A + \exists Z. \text{FreeFi}^{F^\bullet, Z} \times (Z \Rightarrow 1 + A)$

If F^\bullet is already a functor, can simplify the tree encoding using the identity $\exists Z. P^Z \times (Z \Rightarrow 1 + A) \cong P^A$ and obtain $\text{FreeFi}^{F^\bullet, A} \equiv F^A + \text{FreeFi}^{F^\bullet, 1+A}$, which is equivalent to $\text{FreeFi}^{F^\bullet, A} = F^A + F^{1+A} + F^{1+1+A} + \dots$

Reduced encoding: $\text{FreeFi}^{F^\bullet, A} \equiv \exists Z. F^Z \times (Z \Rightarrow 1 + A)$, non-recursive

Derivation: $\exists Z_1 \dots \exists Z_n. F^{Z_n} \times (Z_n \Rightarrow 1 + Z_{n-1}) \times \dots \times (Z_1 \Rightarrow 1 + A)$ is simplified using the laws of `mapOpt` and Kleisli composition, and yields $\exists Z_n. F^{Z_n} \times (Z_n \Rightarrow 1 + A)$. Encode F^A as $\exists Z. F^Z \times (Z \Rightarrow 0 + Z)$.

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

12.1.29 Worked example VII: Free monad

Methods:

$$\begin{aligned}\text{pure} &: A \Rightarrow F^A \\ \text{flatMap} &: F^A \Rightarrow (A \Rightarrow F^B) \Rightarrow F^B\end{aligned}$$

Can recover `map` from `flatMap` and `pure`, so we keep only `flatMap`

Tree encoding: $\text{FreeM}^{F^\bullet, A} \equiv F^A + A + \exists Z. \text{FreeM}^{F^\bullet, Z} \times (Z \Rightarrow \text{FreeM}^{F^\bullet, A})$

Derivation of reduced encoding:

can simplify $A \times (A \Rightarrow \text{FreeM}^{F^\bullet, B}) \cong \text{FreeM}^{F^\bullet, B}$

use associativity to replace $\text{FreeM}^A \times (A \Rightarrow \text{FreeM}^B) \times (B \Rightarrow \text{FreeM}^C)$ by $\text{FreeM}^A \times (A \Rightarrow \text{FreeM}^B \times (B \Rightarrow \text{FreeM}^C))$

therefore we can replace $\exists Z. \text{FreeM}^{F^\bullet, Z} \times \dots$ by $\exists Z. F^Z \times \dots$

Reduced encoding: $\text{FreeM}^{F^\bullet, A} \equiv A + \exists Z. F^Z \times (Z \Rightarrow \text{FreeM}^{F^\bullet, A})$

“Final Tagless style” means “Church encoding of free monad over F^\bullet ”

Free monad over a functor F^\bullet is $\text{FreeM}^{F^\bullet, A} \equiv A + F^{\text{FreeM}^{F^\bullet, A}}$

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 $\text{FreeM}^{F^\bullet, A} \equiv F^A + F^{\text{FreeM}^{F^\bullet, A}}$

start from half-reduced encoding $F^A + \exists Z. F^Z \times (Z \Rightarrow \text{FreeM}^{F^\bullet, A})$

replace the existential type by an equivalent type $F^{\text{FreeM}^{F^\bullet, A}}$

12.1.30 Worked example VIII: Free applicative functor

Methods:

$$\begin{aligned}\text{pure} &: A \Rightarrow F^A \\ \text{ap} &: F^A \Rightarrow F^{A \Rightarrow B} \Rightarrow F^B\end{aligned}$$

We can recover `map` from `ap` and `pure`, so we omit `map`

Tree encoding: $\text{FreeAp}^{F^\bullet, A} \equiv F^A + A + \exists Z. \text{FreeAp}^{F^\bullet, Z} \times \text{FreeAp}^{F^\bullet, Z \Rightarrow A}$

Reduced encoding: $\text{FreeAp}^{F^\bullet, A} \equiv A + \exists Z. F^Z \times \text{FreeAp}^{F^\bullet, Z \Rightarrow A}$

Derivation: a FreeAp^A is either $\exists Z_1 \dots \exists Z_n. Z_1 \times \text{FreeAp}^{Z_1 \Rightarrow Z_2} \times \dots$ or $\exists Z_1 \dots \exists Z_n. F^{Z_1} \times \text{FreeAp}^{Z_1 \Rightarrow Z_2} \times \dots$; encode $Z_1 \times \text{FreeAp}^{Z_1 \Rightarrow Z_2}$ equivalently as $\text{FreeAp}^{Z_1 \Rightarrow Z_2} \times ((Z_1 \Rightarrow Z_2) \Rightarrow Z_2)$ 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 :

$$\begin{aligned}\text{FreeAp}^{F^\bullet, A} &\equiv A + \text{FreeZ}^{F^\bullet, A} \\ \text{FreeZ}^{F^\bullet, A} &\equiv F^A + \exists Z. F^Z \times \text{FreeZ}^{F^\bullet, Z \Rightarrow A}\end{aligned}$$

$\text{FreeZ}^{F^\bullet, \bullet}$ 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

12.1.31 Laws for free typeclass constructions

Consider an inductive typeclass C with methods $C^A \Rightarrow A$

Define a free instance of C over Z recursively, $\text{FreeC}^Z \equiv Z + C^{\text{FreeC}^Z}$

FreeC^Z has an instance of C , i.e. we can implement $C^{\text{FreeC}^Z} \Rightarrow \text{FreeC}^Z$

~~FreeC^Z is a functor in Z and implements the functions $C^Y \Rightarrow \text{FreeC}^Z$~~

$$\begin{aligned}\text{run}^P : (Z \Rightarrow P) &\Rightarrow \text{FreeC}^Z \Rightarrow P \\ \text{wrap} : Z &\Rightarrow \text{FreeC}^Z\end{aligned}$$

$$\begin{array}{ccc} \text{FreeC}^Y & & \\ \downarrow \text{fmap } f : Y \Rightarrow Z & & \\ \text{FreeC}^Z & \xrightarrow{\text{run}(f \circ g)} & P \\ & \text{run}(g : Z \Rightarrow P) & \end{array}$$

Law 1: $\text{run}(\text{wrap}) = \text{id}$; law 2: $\text{fmap } f \circ \text{run } g = \text{run}(f \circ g)$ (naturality of `run`)

For any $P:C, Q:C, g:Z \Rightarrow P$, and a typeclass-preserving $f:P \Rightarrow Q$, we have

$$\text{run}^P(g \circ f) = \text{run}^Q(g \circ f) \quad \text{— “universal property” of run}$$

$$\begin{array}{ccc} \text{FreeC}^Z & & C^P \xrightarrow{\text{ops}_P} P \\ \text{run}^P(g : Z \Rightarrow P) \downarrow & \searrow \text{run}^Q(g \circ f) & \downarrow \text{fmap}_S f \\ P \xrightarrow{f : P \Rightarrow Q} Q & & C^Q \xrightarrow{\text{ops}_Q} Q \end{array}$$

$f:P \Rightarrow Q$ preserves typeclass C if the diagram on the right commutes

12.1.32 Combining the generating constructors in a free typeclass

Consider FreeC^Z for an inductive typeclass C with methods $C^X \Rightarrow X$

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:

$\text{FreeC}^Z \equiv \forall X. (Z \Rightarrow X) \times (C^X \Rightarrow X) \Rightarrow X$ and then

$$\text{FreeC}^{Z_1+Z_2} \equiv \forall X. (Z_1 \Rightarrow X) \times (Z_2 \Rightarrow X) \times (C^X \Rightarrow X) \Rightarrow X$$

Encode the functions $Z_i \Rightarrow X$ via typeclasses `ExZ1`, `ExZ2`, etc., where typeclass `ExZ1` has method $Z_1 \Rightarrow X$, etc.

Then

$$\text{FreeC}^{Z_1+Z_2} = \forall X^{:E_{Z_1}:E_{Z_2}}. (C^X \Rightarrow X) \Rightarrow X$$

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

12.1.33 Combining different free typeclasses

To combine free instances of different typeclasses C_1 and C_2 :

Option 1: use functor composition, $\text{FreeC}_{12}^Z \equiv \text{FreeC}_1^{\text{FreeC}_2^Z}$

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, $C^X \equiv C_1^X + C_2^X \Rightarrow X$, and build the free typeclass instance using C^X

Church encoding: $\text{FreeC}_{12}^Z \equiv \forall X. (Z \Rightarrow X) \times (C_1^X + C_2^X \Rightarrow X) \Rightarrow X$

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

12.1.34 Exercises

Implement a free semigroup generated by 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 before interpreting into a lawful semigroup S .

Type P is of typeclass Mod_L (called “ L -module”) if a fixed monoid L “acts” on P via act: $L \Rightarrow P \Rightarrow P$, with laws $\text{act } x ; \text{act } y = \text{act } (x ; y)$ and $\text{act } (1 : L) = \text{id}$. Show that Mod_L is an inductive typeclass. Implement a free L -module over a type Z .

Implement a monadic DSL with operations put: $A \Rightarrow 1$ and get: A ; run examples.

Implement the Church encoding of the type constructor $P^A \equiv \text{Int} + A \times A$. For the resulting type constructor, implement a Functor instance.

Describe the monoid type class via a method functor C^\bullet (such that the monoid’s operations are combined into the type $S^M \Rightarrow M$). Using S^\bullet , implement the free monoid over a type Z in the Church encoding.

Assuming that F^\bullet is a functor, define $Q^A \equiv \exists Z. F^Z \times (Z \Rightarrow A)$ and implement f2q: $F^A \Rightarrow Q^A$ and q2f: $Q^A \Rightarrow F^A$. 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 .

Show: $\forall X. X = 0 ; \exists Z. Z \cong 1 ; \exists Z. Z \times A \cong A ; \forall A. (A \times A \times A \Rightarrow A) \cong 1 + 1 + 1$.

Derive a reduced encoding for a free applicative functor over a pointed functor.

Implement a “free pointed filterable” typeclass (combining pointed and filterable) over a type constructor F^\bullet in the tree encoding. Derive a reduced encoding. Simplify these encodings when F^\bullet is already a functor.

12.1.35 Corrections

The slides say that the “universal property” of the runner is $\text{run}^P g ; f = \text{run}^Q (g ; f)$, however, this is not true; it is the right naturality property of $\text{run}^P : (Z \Rightarrow P) \Rightarrow \text{FreeC}^Z \Rightarrow P$ with respect to the type parameter P . The universal property is $f = \text{wrap} ; \text{run}^P f$ for any $f : Z \Rightarrow P$ and any type P that belongs to the typeclass C .

12.2 Discussion

13 Computations in functor blocks. III. Monad transformers

13.1 Slides

13.1.1 Computations within a functor context: Combining monads

Programs often need to combine monadic effects (see code)

“Effect” \triangleq what else happens in $A \Rightarrow M^B$ besides computing B from A

Examples of effects for some standard monads:

`Option` – computation will have no result or a single result

`List` – computation will have zero, one, or multiple results

`Either` – computation may fail to obtain its result, reports error

`Reader` – computation needs to read an external context value

`Writer` – some value will be appended to a (monoidal) accumulator

`Future` – computation will be scheduled to run later

How to combine several effects in the same functor block (`This/yield`? valid Scala!)

```
val result = for { i <- 1 to n
                  j <- Future { q(i) }
                  k <- maybeError(j) : Try[Int]
                } yield f(k)
    // What should be the type of result??
```

`(1 to n).flatMap { i => Future(q(i)).flatMap { j => maybeError(j).map { k => f(k) } }}`

The code will work if we “unify” all effects in a new, larger monad

Need to compute the type of new monad that contains all given effects

13.1.2 Combining monadic effects I. Trial and error

There are several ways of combining two monads into a new monad:

If M_1^A and M_2^A are monads then $M_1^A \times M_2^A$ is also a monad

But $M_1^A \times M_2^A$ describes two separate values with two separate effects

If M_1^A and M_2^A are monads then $M_1^A + M_2^A$ is usually not a monad

If it worked, it would be a choice between two different values / effects

If M_1^A and M_2^A are monads then one of $M_1^{M_2^A}$ or $M_2^{M_1^A}$ is often a monad

Examples and counterexamples for functor composition:

Combine $Z \Rightarrow A$ and List^A as $Z \Rightarrow \text{List}^A$

Combine `Future[A]` and `Option[A]` as `Future[Option[A]]`

But `Either[Z, Future[A]]` and `Option[Z => A]` are not monads

Neither `Future[State[A]]` nor `State[Future[A]]` are monads

The order of effects matters when composition works both ways:

Combine `Either` ($M_1^A = Z + A$) and `Writer` ($M_2^A = W \times A$)

as $Z + W \times A$ – either compute result and write a message, or all fails

as $(Z + A) \times W$ – message is always written, but computation may fail

Find a general way of defining a new monad with combined effects

Derive properties required for the new monad

13.1.3 Combining monadic effects II. Lifting into a larger monad

If a “big monad” $\text{BigM}[A]$ somehow combines all the needed effects:

```
// This could be valid Scala...
val result: BigM[Int] = for {
    i ← lift1(1 to n)
    j ← lift2(Future{ q(i) })
    k ← lift3(maybeError(j))
} yield f(k)
```

// If we define the various
// required “lifting” functions:
def lift1[A]: Seq[A] ⇒ BigM[A] = ???
def lift2[A]: Future[A] ⇒ BigM[A] = ???
def lift3[A]: Try[A] ⇒ BigM[A] = ???

Example 1: combining as $\text{BigM}[A]$ $\Rightarrow \text{Future}[Option[A]]$ with liftings:

```
def lift1[A]: Future[Option[A]] ⇒ BigM[A] = _.map(x ⇒ Some(x))
```

Example 2: combining as $\text{BigM}[A] = \text{List}[\text{Try}[A]]$ with liftings:

```
def lift1[A]: Try[A] ⇒ List[Try[A]] = x ⇒ List(x)
```

```
def lift2[A]: List[A] ⇒ List[Try[A]] = _.map(x ⇒ Success(x))
```

Remains to be understood:

Finding suitable laws for the liftings; checking that the laws hold

Building a “big monad” out of “smaller” ones, with lawful liftings

Is this always possible? Unique? Are there alternative solutions?

Ways of reducing the complexity of code; make liftings automatic

13.1.4 Laws for monad liftings I. Identity laws

Whatever identities we expect to hold for monadic programs must continue to hold after lifting M_1 or M_2 values into the “big monad” BigM

We assume that M_1 , M_2 , and BigM already satisfy all the monad laws

Consider the various functor block constructions containing the liftings:

Left identity law after lift_1 :

```
// Anywhere inside a for/yield:  
i ← lift1(M1.pure(x)) // Any BigM value.  
j ← bigM(i) // Any BigM value.
```

// Must be equivalent to...
i = x
j ← bigM(x)

$\text{lift}_1(M_1.\text{pure}(x)).\text{flatMap}(b) = b(x)$ — in terms of Kleisli composition (\diamond):

$(\text{pure}_{M_1} \circ \text{lift}_1)^{:X \Rightarrow \text{BigM}^X} \diamond b^{:X \Rightarrow \text{BigM}^Y} = b$ with $f^{:X \Rightarrow M^Y} \diamond g^{:Y \Rightarrow M^Z} \triangleq x \Rightarrow f(x).\text{flatMap}(g)$

Right identity law after lift_1 :

```
// Anywhere inside a for/yield:  
x ← bigM // Any BigM value.  
i ← lift1(M1.pure(x))
```

// Must be equivalent to...
x ← bigM
i = x

$b.\text{flatMap}(M_1.\text{pure andThen lift}_1) = b$ — in terms of Kleisli composition:

$b^{:X \Rightarrow \text{BigM}^Y} \diamond (\text{pure}_{M_1} \circ \text{lift}_1)^{:Y \Rightarrow \text{BigM}^Y} = b$

The same identity laws must hold for M_2 and lift_2 as well

13.1.5 Laws for monad liftings II. Simplifying the laws

$(\text{pure}_{M_1} \circ \text{lift}_1)$ is a unit for the Kleisli composition \diamond in the monad BigM

But the monad BigM already has a unit element, namely $\text{pure}_{\text{BigM}}$

The two-sided unit element is always unique: $u = u \diamond u' = u'$

So the two identity laws for $(\text{pure}_{M_1} \circ \text{lift}_1)$ can be reduced to one law:

$\text{pure}_{M_1} \circ \text{lift}_1 = \text{pure}_{\text{BigM}}$

Refactoring a portion of a monadic program under lift_1 gives another law:

```
// Anywhere inside a for/yield, this: i ← lift1(p) // Any M1 value.  
j ← lift1(q(i)) // Any M1 value.
```

// must be equivalent to...
 $pq = p.\text{flatMap}(q)$ // In M_1 .
 $j ← lift1(pq)$ // Now lift it.

$\text{lift}_1(p).\text{flatMap}(q \text{ andThen lift}_1) = \text{lift}_1(p.\text{flatMap} q)$

Rewritten equivalently through $\text{flm}_M : (A \Rightarrow M^B) \Rightarrow M^A \Rightarrow M^B$ as

$\text{lift}_1 \circ \text{flm}_{\text{BigM}}(q \circ \text{lift}_1) = \text{flm}_{M_1}(q \circ \text{lift}_1)$ — both sides are functions $M_1^A \Rightarrow \text{BigM}^B$

Rewritten equivalently through $\text{ftn}_M : M^{M^A} \Rightarrow M^A$, the law is

$\text{lift}_1 \circ \text{fmap}_{\text{BigM}} \text{lift}_1 \circ \text{ftn}_{\text{BigM}} = \text{ftn}_{M_1} \circ \text{lift}_1$ — both sides are functions $M_1^{M^A} \Rightarrow \text{BigM}^A$

In terms of Kleisli composition \diamond_M it becomes the **composition law**:

$(b^{:X \Rightarrow M_1^Y} \circ \text{lift}_1) \diamond_{\text{BigM}} (c^{:Y \Rightarrow M_1^Z} \circ \text{lift}_1) = (b \diamond_{M_1} c) \circ \text{lift}_1$

Liftings lift_1 and lift_2 must obey an identity law and a composition law

The laws say that the liftings **commute** with the monads’ operations

13.1.6 Laws for monad liftings III. The naturality law

Show that $\text{lift}_1 : M_1^A \Rightarrow \text{BigM}^A$ is a natural transformation

It maps pure_{M_1} to $\text{pure}_{\text{BigM}}$ and flm_{M_1} to flm_{BigM}

lift_1 is a **monadic morphism** between monads M_1^\bullet and BigM^\bullet

example: monad “interpreters” $M^A \Rightarrow N^A$ are monadic morphisms

The (functor) naturality law: for any $f : X \Rightarrow Y$,

$$\begin{array}{c} \text{lift}_1 \circ \text{fmap}_{\text{BigM}} f = \text{fmap}_{M_1} f \circ \text{lift}_1 \\ M_1^X \xrightarrow{\text{lift}_1} \text{BigM}^X \\ \downarrow \text{fmap}_{M_1} f : X \Rightarrow Y \qquad \qquad \qquad \downarrow \text{fmap}_{\text{BigM}} f : X \Rightarrow Y \\ M_1^Y \xrightarrow{\text{lift}_1} \text{BigM}^Y \end{array}$$

Derivation of the functor naturality law for lift_1 :

Express fmap as $\text{fmap}_M f \triangleq f \uparrow M = \text{flm}_M (f \circ \text{pure}_M)$ for both monads

Given $f : X \Rightarrow Y$, use the law $\text{flm}_{M_1} q \circ \text{lift}_1 = \text{lift}_1 \circ \text{flm}_{\text{BigM}} (q \circ \text{lift}_1)$ to compute $\text{flm}_{M_1} (f \circ \text{pure}_{M_1}) \circ \text{lift}_1 = \text{lift}_1 \circ \text{flm}_{\text{BigM}} (f \circ \text{pure}_{\text{BigM}})$

$\text{lift}_1 \circ \text{flm}_{\text{BigM}} (f \circ \text{pure}_{\text{BigM}}) = \text{lift}_1 \circ \text{fmap}_{\text{BigM}} f$

A monadic morphism is always also a natural transformation of the functors

13.1.7 Monad transformers I: Motivation

Combine $Z \Rightarrow A$ and $1 + A$: only $Z \Rightarrow 1 + A$ works, not $1 + (Z \Rightarrow A)$

It is not possible to combine monads via a natural bifunctor B^{M_1, M_2}

It is not possible to combine arbitrary monads as $M_1^{M_2^\bullet}$ or $M_2^{M_1^\bullet}$

Example: state monad $\text{St}_S^A \triangleq S \Rightarrow A \times S$ does not compose

The trick: for a fixed **base** monad L^\bullet , let M^\bullet (**foreign** monad) vary

Call the desired result $T_L^{M^\bullet}$ the **monad transformer for L**

In Scala: `MyMonadT[M[_]:Monad, A]` – e.g. `ReaderT`, `StateT`, etc.

$T_L^{M^\bullet}$ is generic in M but not in L

No general formula for monad transformers seems to exist

For each base monad L , a different construction is needed

Some transformers are compositions L^{M^\bullet} or M^{L^\bullet} , others are not

Do all monads L have a transformer? (Unknown.)

To combine 3 or more monads, “stack up” the transformers as $T_{L_1}^{T_{L_2}^{T_{L_3}^{M^\bullet}}}$.

Example in Scala: `StateT[S, ListT[Reader[R, ?], ?], A]`

Substitute nested transformers into the monad argument, not as A

This is called a **monad stack** – but may not be *functor composition*

because e.g. `State[S, List[Reader[R, A]]]` is not a monad

13.1.8 Monad transformers II: The requirements

A **monad transformer** for a **base** monad L^\bullet is a type constructor $T_L^{M^\bullet}$ parameterized by a monad M^\bullet , such that for all monads M^\bullet :

$T_L^{M^\bullet}$ is a monad (the monad M **transformed with** T_L)

“Lifting” – a monadic morphism $\text{lift}_L^M : M^A \rightsquigarrow T_L^{M,A}$

“Base lifting” – a monadic morphism $\text{blift} : L^A \rightsquigarrow T_L^{M,A}$

The “base lifting” could not possibly be natural in L^\bullet

Transformed identity monad (Id) must become L , i.e. $T_L^{\text{Id}, \bullet} \cong L^\bullet$

$T_L^{M^\bullet}$ is **monadically natural** in M^\bullet (but not in L^\bullet)

$T_L^{M^\bullet}$ is natural w.r.t. a monadic functor M^\bullet as a type parameter

For any monad N^\bullet and a monadic morphism $f : M^\bullet \rightsquigarrow N^\bullet$ we need to have a monadic morphism $T_L^{M,\bullet} \rightsquigarrow T_L^{N,\bullet}$ for the transformed monads: $\text{mrun}_L^M : (M^\bullet \rightsquigarrow N^\bullet) \Rightarrow T_L^{M,\bullet} \rightsquigarrow T_L^{N,\bullet}$ with the “lifting” laws

If we implement $T_L^{M,\bullet}$ only via M ’s monad methods, naturality will hold

Cf. `traverse`: $L^A \Rightarrow (A \Rightarrow F^B) \Rightarrow F^{L^B}$ – natural w.r.t. applicative F^\bullet

This can be used for lifting a “runner” $M^A \rightsquigarrow A$ to $T_L^{M,\bullet} \rightsquigarrow T_L^{\text{Id},\bullet} = L^\bullet$

“Base runner”: lifts $L^A \rightsquigarrow A$ into a monadic morphism $T_L^{M,\bullet} \rightsquigarrow M^\bullet$; so $\text{brun}_L^M : (L^\bullet \rightsquigarrow \bullet) \Rightarrow T_L^{M,\bullet} \rightsquigarrow M^\bullet$, must commute with `lift` and `blift`

13.1.9 Monad transformers III: First examples

Recall these monad constructions:

If M^A is a monad then $R \Rightarrow M^A$ is also a monad (for a fixed type R)

If M^A is a monad then $M^{Z+A\times W}$ is also a monad (for fixed W, Z)

This gives the monad transformers for base monads `Reader`, `Writer`, `Either`:

```
type ReaderT[R, M[_], A] = R ⇒ M[A]
type EitherT[Z, M[_], A] = M[Either[Z, A]]
type WriterT[W, M[_], A] = M[(W, A)]
```

`ReaderT` composes with the foreign monad from the *outside*

`EitherT` and `WriterT` must be composed *inside* the foreign monad

Remaining questions:

What are transformers for other standard monads (`List`, `State`, `Cont`)?

These monads do not compose (neither “inside” nor “outside” works)!

How to derive a monad transformer for an arbitrary given monad?

For monads obtained via known monad constructions?

For monads constructed via other monad transformers? (Stack them.)

Is it always possible? (No known counterexamples.)

Is a given monad’s transformer unique? (No.)

How to avoid the boilerplate around `lift`? (`mtl`-style transformers)

13.1.10 Monad transformers IV: The zoology of *ad hoc* methods

Need to choose the correct monad transformer construction, per monad:

“Composed-inside”, base monad is inside foreign monad: $T_L^{M,A} = M^{L^A}$

Examples: the **linear** monads `OptionT`, `WriterT`, `EitherT`

“Composed-outside” – the base monad is outside: $T_L^{M,A} = L^{M^A}$

Examples: `ReaderT`; `SearchT` for search monad $S[A] = (A \Rightarrow Z) \Rightarrow A$

More generally: all **rigid** monads have “outside” transformers

“Recursive”: interleaves the base monad and the foreign monad

Examples: `ListT`, `NonEmptyListT`, `FreeMonadT`

Monad constructions: defining a transformer for new monads

Product monads $L_1^A \times L_2^A$ – product transformer $T_{L_1}^{M,A} \times T_{L_2}^{M,A}$

“Contrafunctor-choice” $H^A \Rightarrow A$ – composed-outside transformer

Free pointed monads $A + L^A$ – transformer $M^{A+T_L^{M,A}}$

“Irregular”: none of the above constructions work, need something else

$T_{\text{State}}^{M,A} = S \Rightarrow M^{S \times A}$; $T_{\text{Cont}}^{M,A} = (A \Rightarrow M^R) \Rightarrow M^R$; “selector” $F^{A \Rightarrow P^Q} \Rightarrow P^A$ – transformer $F^{A \Rightarrow T_P^{M,Q}} \Rightarrow T_P^{M,A}$; codensity $\forall R. (A \Rightarrow M^R) \Rightarrow M^R$

Examples and monads like K^A for which no transformers exist? (not known)

13.2 Practical use

13.3 Laws and structure

13.3.1 Laws of monad transformers

A monad transformer $T_L^{M,A}$ is a type constructor with a type parameter A and a monad parameter M , such that the following laws hold:

- Monad construction law:** $T_L^{M,\bullet}$ is a lawful monad for any monad M . In other words, the transformed monad $T_L^{M,\bullet}$ has methods pu_T and ftn_T that satisfy the monad laws.
- Identity law:** $T_L^{\text{Id},\bullet} \cong L^\bullet$ via a monadic isomorphism, where Id is the identity monad, $\text{Id}^A \triangleq A$.
- Lifting law:** For any monad M , the function $\text{lift} : M^A \Rightarrow T_L^{M,A}$ is a monadic morphism. (In a shorter notation, $\text{lift} : M \rightsquigarrow T_L^M$.)
- Runner laws:** For any monads M, N and any monadic morphism $\phi : M \rightsquigarrow N$, the runner $\text{mrun}(\phi) : T_L^M \rightsquigarrow T_L^N$ is a monadic morphism. Moreover, the function mrun lifts monadic morphisms from $M \rightsquigarrow N$ to $T_L^M \rightsquigarrow T_L^N$ and must satisfy the corresponding **lifting laws**:

$$\text{mrun}(\text{id}) = \text{id} , \quad \text{mrun}(\phi) ; \text{mrun}(\chi) = \text{mrun}(\phi ; \chi) .$$

It follows from the identity law $T_L^{\text{Id}} \cong L$ that the base monad L can be lifted into T_L^M : Setting $\phi = \text{pu}_M : \text{Id} \rightsquigarrow M$, we obtain

$$\text{mrun}(\text{pu}_M) : T_L^{\text{Id}} \rightsquigarrow T_L^M = L \rightsquigarrow T_L^M .$$

This function is called the **base lifting**, $\text{mrun}(\text{pu}_M) \triangleq \text{blift} : L^A \Rightarrow T_L^{M,A}$. The base lifting automatically satisfies the non-degeneracy law,

$$\text{blift} ; \text{mrun}(\phi^{M \rightsquigarrow \text{Id}}) = \text{id} ,$$

for any monadic morphism $\phi : M \rightsquigarrow \text{Id}$, because the left-hand side equals $\text{mrun}(\text{pu}_M ; \phi)$, and the composition law for monadic morphisms gives $\text{pu}_M ; \phi = \text{pu}_{\text{Id}} = \text{id}$.

- Base runner laws:** For any monadic morphism $\theta : L \rightsquigarrow \text{Id}$ and for any monad M , the base runner $\text{brun}(\theta) : T_L^M \rightsquigarrow M$ is a monadic morphism. The base runner must also satisfy the **non-degeneracy law**,

$$\text{lift} ; \text{brun}(\theta) = \text{id} .$$

Since the monad transformer is specific to the base monad L and does not support an arbitrary other base monad, there are no lifting laws for brun , unlike mrun . So the non-degeneracy law for brun is not an automatic consequence of other laws.

In total, monad transformers must satisfy 15 laws. Are all these laws necessary?

13.3.2 Examples of incorrect 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 may obtain a type constructor that does not work correctly even though it has the methods with the required type signatures.

The simplest example of an incorrect monad transformer is obtained by defining the transformed monad to be the unit monad, $T_L^{M,A} \triangleq \mathbb{1}$, for any monads L and M . It is clear that this “transformer”

is completely wrong: it cannot possibly keep the information about the monads L and M , because the methods of the unit monad discard *all* information and return 1. However, the type constructor $T_L^{M,A}$ still has all the required methods pu_T , ftnt_T , lift_T , mrun_T , and brun_T with the required type signatures (they are constant functions returning 1). All these functions are automatically monadic morphisms, since a function from any monad to the unit monad is always a monadic morphism. So, the fake “transformer” satisfies almost all of the monad transformer laws! However, the identity law $T_L^{\text{Id}} \cong L$ and the non-degeneracy law $\text{lift} ; \text{brun}(\theta) = \text{id}$ are violated since $T_L^{\text{Id}} = \mathbb{1} \not\cong L$ and $\text{lift} ; \text{brun}(\theta) = (_ \Rightarrow 1) \neq \text{id}$. For this reason, the unit monad is not a lawful monad transformer.

This simple example demonstrates the importance of the monad transformer laws. A malicious programmer could give us a fake implementation of a “transformer” that appears to have all the methods with the correct type signatures but, instead of a bigger monad, constructs a unit monad dressed up as a type constructor $T_L^{M,A}$. The only way for us to detect the fraud is to find that the identity law and the non-degeneracy law are violated.

Other examples of fake “transformers” violating some of the laws are $T_L^M = L$ (no lifting law) and $T_L^M = M$ (no identity law).

In these cases, it is intuitively clear that the fake transformer definitions are incorrect because the information about either L or M is missing in T_L^M . A potentially working definition of T_L^M must be a type constructor whose definition somehow combines both L and M . Many such definitions are possible, but few will satisfy the monad transformer laws.

13.3.3 Examples of failure to define a generic monad transformer

It appears to be impossible to define T_L^M as a generic construction that works in the same way for all monads L and M . We will now consider a few ways of combining the type constructors L and M in a way that is independent of their structure. In all these cases, we will find that some of the monad transformer laws are violated.

General ways of combining two type constructors L^\bullet and M^\bullet are functor composition L^{M^\bullet} or M^{L^\bullet} , disjunction $L^\bullet + M^\bullet$, and product $L^\bullet \times M^\bullet$.

Functor composition A general way of combining two type constructors L^\bullet and M^\bullet is the functor composition L^{M^\bullet} or M^{L^\bullet} . However, the functor composition works only for certain monads and only in a certain order; so it cannot work as a generic monad transformer. A simple counter-example is $L^A \triangleq \mathbb{1} + A$ and $M^A \triangleq A \times A$ where M^{L^A} is a monad but L^{M^A} is not (see Section ???). Another counter-example is the `State` monad, $\text{State}_S^A \triangleq S \Rightarrow S \times A$, for which we have already shown that $\mathbb{1} + \text{State}_S^A$ is not a monad and $\text{State}_S^{Z \Rightarrow A}$ is not a monad (see Section ???). In other words, the `State` monad does not compose with arbitrary monads M in either order.

Functor disjunction The functor disjunction $L^\bullet + M^\bullet$ is in general not a monad when L and M are arbitrary monads. An immediate counter-example is found by using two `Reader` monads, $L^A \triangleq R \Rightarrow A$ and $M^A \triangleq S \Rightarrow A$. The disjunction $(R \Rightarrow A) + (S \Rightarrow A)$ is a functor that is not a monad (and not even applicative, see Section ???).

Functor product The functor product $L^\bullet \times M^\bullet$ is a monad for arbitrary monads L and M . However, there is no naturally defined $\text{lift} : M^\bullet \rightsquigarrow L^\bullet \times M^\bullet$ because we cannot create values of type L^A out of values of type M^A for arbitrary monads L and M .

Using the free monad The functor composition L^{M^\bullet} and the disjunction $L^\bullet + M^\bullet$ may not always be monads, but they are always functors. So we can make monads out of them, by using the free monad construction. We get $\text{Free}^{L^{M^\bullet}}$, 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 transformer are satisfied by these constructions. However, the identity laws fail because

$$\text{Free}^{L^{\text{Id}}^\bullet} \cong \text{Free}^{L^\bullet} \not\cong L \quad , \quad \text{Free}^{L^\bullet + \text{Id}^\bullet} \not\cong L \quad ,$$

and the lifting laws are also violated because $\text{lift} : M^A \Rightarrow \text{Free}^{L^\bullet + M^\bullet, A}$ is not a monad morphism because 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.

“Monoidal convolution” The construction called “**monoidal convolution**” defines a new functor $L \star M$ via

$$(L \star M)^A \triangleq \exists P \exists Q. (P \times Q \Rightarrow A) \times L^P \times M^Q . \quad (13.1)$$

This formula can be seen as a combination of the co-Yoneda identities

$$L^A \cong \exists P. L^P \times (P \Rightarrow A) , \quad M^A \cong \exists Q. M^Q \times (Q \Rightarrow A) .$$

The functor product $L \times M$ is equivalent to

$$\begin{aligned} & L^A \times M^A \\ \text{co-Yoneda identities for } L^A \text{ and } M^A : & \cong \exists P. L^P \times \underline{(P \Rightarrow A)} \times \exists Q. M^Q \times \underline{(Q \Rightarrow A)} \\ \text{equivalence in Eq. (13.3) :} & \cong \exists P. \exists Q. L^P \times M^Q \times (P + Q \Rightarrow A) \end{aligned} \quad (13.2)$$

where we used the type equivalence

$$(P \Rightarrow A) \times (Q \Rightarrow A) \cong P + Q \Rightarrow A . \quad (13.3)$$

If we (arbitrarily) replace $P + Q \Rightarrow A$ by $P \times Q \Rightarrow A$ in Eq. (13.2), we will obtain Eq. (13.1).

The monoidal convolution $L \star M$ always produces a functor since Eq. (13.1) is covariant in A . An example where the monoidal convolution fails to produce a monad transformer is $L^A \triangleq 1 + A$ and $M^A \triangleq R \Rightarrow A$. We compute the functor $L \star M$ and establish that it is not a monad:

$$\begin{aligned} & (L \star M)^A \\ \text{definitions of } L, M, \star : & \cong \exists P \exists Q. \underline{(P \times Q \Rightarrow A)} \times (1 + P) \times (R \Rightarrow Q) \\ \text{curry the arguments, move quantifier :} & \cong \exists P. (1 + P) \times \underline{(Q \Rightarrow P \Rightarrow A)} \times (R \Rightarrow Q) \\ \text{co-Yoneda identity with } \exists Q : & \cong \exists P. (1 + P) \times \underline{(R \Rightarrow P \Rightarrow A)} \\ \text{swap curried arguments :} & \cong \exists P. (1 + P) \times (P \Rightarrow R \Rightarrow A) \\ \text{co-Yoneda identity with } \exists P : & \cong 1 + (R \Rightarrow A) . \end{aligned}$$

This functor is not a monad (see Section ???).

Codensity tricks***

- codensity monad over L^{M^\bullet} : $F^A \triangleq \forall B. (A \Rightarrow L^{M^B}) \Rightarrow L^{M^B}$ – no lift
- Codensity- L transformer: $\text{Cod}_L^{M,A} \triangleq \forall B. (A \Rightarrow L^B) \Rightarrow L^{M^B}$ – no lift
 - applies the continuation transformer to $M^A \cong \forall B. (A \Rightarrow B) \Rightarrow M^B$
- Codensity composition: $F^A \triangleq \forall B. (M^A \Rightarrow L^B) \Rightarrow L^B$ – not a monad
 - Counterexample: $M^A \triangleq R \Rightarrow A$ and $L^A \triangleq S \Rightarrow A$

13.3.4 Properties of monadic morphisms

A natural transformation $\phi^A : M^A \Rightarrow N^A$, equivalently written as $\phi : M^\bullet \rightsquigarrow N^\bullet$, is called a **monadic morphism** between monads M and N if the following two laws hold:

$$\begin{aligned} \text{identity law for } \phi : & \text{pu}_M \circ \phi = \text{pu}_N , \\ \text{composition law for } \phi : & \text{ftn}_M \circ \phi = \phi^{\uparrow M} \circ \phi \circ \text{ftn}_N . \end{aligned}$$

Statement 13.3.4.1 For any monad M , the method $\text{pu}_M : A \Rightarrow M^A$ is a monadic morphism $\text{pu}_M : \text{Id} \rightsquigarrow M$ between the identity monad and M .

Proof The identity law requires $\text{pu}_{\text{Id}} ; \text{pu}_M = \text{pu}_M$. This holds because $\text{pu}_{\text{Id}} = \text{id}$. The composition law requires $\text{ftn}_{\text{Id}} ; \text{pu}_M = \text{pu}_M^{\uparrow \text{Id}} ; \text{pu}_M ; \text{ftn}_M$. Since $\text{ftn}_{\text{Id}} = \text{id}$, the left-hand side of the composition law simplifies to pu_M . Transform the right-hand side:

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

Exercise 13.3.4.2 Suppose M is a given monad, Z is a fixed type, and a fixed value $m : M^Z$ is given.

(a) Consider the function f defined as

$$\begin{aligned} f : (Z \Rightarrow A) &\Rightarrow M^A , \\ f(q^{Z \Rightarrow A}) &\triangleq q^{\uparrow M} m . \end{aligned}$$

Prove that f is *not* a monadic morphism from the reader monad $R^A \triangleq Z \Rightarrow A$ to the monad M , despite having the correct type signature.

(b) Under the same assumptions, consider the function ϕ defined as

$$\begin{aligned} \phi : (Z \Rightarrow M^A) &\Rightarrow M^A , \\ \phi(q^{Z \Rightarrow M^A}) &\triangleq \text{flm}_M(q)(m) . \end{aligned}$$

Show that ϕ is *not* a monadic morphism from the monad $Q^A \triangleq Z \Rightarrow M^A$ to M .

Statement 13.3.4.3 If L, M, N are monads and $\phi : L^\bullet \rightsquigarrow M^\bullet$ and $\chi : M^\bullet \rightsquigarrow N^\bullet$ are monadic morphisms then the composition $\phi ; \chi : L^\bullet \rightsquigarrow N^\bullet$ is also a monadic morphism.

Proof The identity law for $\phi ; \chi$ is

$$\begin{aligned} \text{expect to equal } \text{pu}_N : & \quad \text{pu}_L ; (\phi ; \chi) \\ \text{identity law for } \phi : & \quad = \text{pu}_M ; \chi \\ \text{identity law for } \chi : & \quad = \text{pu}_N . \end{aligned}$$

The composition law for $\phi ; \chi$ is

$$\begin{aligned} \text{expect to equal } \text{ftn}_L ; \phi ; \chi : & \quad (\phi ; \chi)^{\uparrow L} ; (\phi ; \chi) ; \text{ftn}_N \\ \text{naturality of } \phi : & \quad = \phi^{\uparrow L} ; \phi ; \chi^{\uparrow M} ; \chi ; \text{ftn}_N \\ \text{composition law for } \chi : & \quad = \phi^{\uparrow L} ; \phi ; \text{ftn}_M ; \chi \\ \text{composition law for } \phi : & \quad = \text{ftn}_L ; \phi ; \chi . \end{aligned}$$

Statement 13.3.4.4 For any monad M , the function $\Delta : M^A \Rightarrow M^A \times M^A$ is a monadic morphism between monads M and $M \times M$.

Proof We use the definition of the product monad. The method $\text{pu}_{M \times M}$ is defined by

$$x \triangleright \text{pu}_{M \times M} = \text{pu}_M(x) \times \text{pu}_M(x) = x \triangleright \text{pu}_M ; \Delta ,$$

which is the identity law for Δ . To verify the composition law for Δ ,

$$\text{ftn}_M ; \Delta = \Delta^{\uparrow M} ; \Delta ; \text{ftn}_{M \times M} ,$$

we use the definition of $\text{ftn}_{M \times M}$,

$$\text{ftn}_{M \times M} : M^{M^\bullet \times M^\bullet} \times M^{M^\bullet \times M^\bullet} \Rightarrow M^\bullet \times M^\bullet = (\nabla_1^{\uparrow M} ; \text{ftn}_M) \boxtimes (\nabla_2^{\uparrow M} ; \text{ftn}_M) ,$$

and compute

$$\begin{aligned} \text{expect to equal } \text{ftn}_M ; \Delta &: \underline{\Delta^{\uparrow M}} ; \Delta ; \text{ftn}_{M \times M} \\ \text{naturality of } \Delta &: = \Delta ; (\Delta^{\uparrow M} \boxtimes \Delta^{\uparrow M}) ; \underline{\text{ftn}_{M \times M}} \\ \text{definition of } \text{ftn}_{M \times M} &: = \Delta ; (\Delta^{\uparrow M} \boxtimes \Delta^{\uparrow M}) ; (\nabla_1^{\uparrow M} ; \text{ftn}_M) \boxtimes (\nabla_2^{\uparrow M} ; \text{ftn}_M) \\ \text{composition law of } \boxtimes &: = \Delta ; (\underline{\Delta^{\uparrow M}} ; \nabla_1^{\uparrow M} ; \text{ftn}_M) \boxtimes (\underline{\Delta^{\uparrow M}} ; \nabla_2^{\uparrow M} ; \text{ftn}_M) \\ \text{simplify } \Delta ; \nabla_i = \text{id} &: = \Delta ; \text{ftn}_M \boxtimes \text{ftn}_M \\ \text{duplication law of } \Delta &: = \text{ftn}_M ; \Delta . \end{aligned}$$

Statement 13.3.4.5 For any monads K, L, M, N and monadic morphisms $\phi : K^\bullet \rightsquigarrow M^\bullet$ and $\chi : L^\bullet \rightsquigarrow N^\bullet$, the pair product $\phi \boxtimes \chi : K^\bullet \times L^\bullet \rightsquigarrow M^\bullet \times N^\bullet$ is a monadic morphism between the product monads $K \times L$ and $M \times N$.

Proof The definitions of $\text{pu}_{K \times L}$ and $\text{pu}_{M \times N}$ are

$$\text{pu}_{K \times L} = \text{pu}_K \boxtimes \text{pu}_L , \quad \text{pu}_{M \times N} = \text{pu}_M \boxtimes \text{pu}_N .$$

The identity law for $\phi \boxtimes \chi$ is verified by

$$\begin{aligned} \text{expect to equal } \text{pu}_{M \times N} &: \underline{\text{pu}_{K \times L}} ; \phi \boxtimes \chi \\ \text{definition of } \text{pu}_{K \times L} &: = \text{pu}_K \boxtimes \text{pu}_L ; \phi \boxtimes \chi \\ \text{composition law of } \boxtimes &: = (\underline{\text{pu}_K ; \phi}) \boxtimes (\underline{\text{pu}_L ; \chi}) \\ \text{identity laws for } \phi, \chi &: = \text{pu}_M \boxtimes \text{pu}_N = \text{pu}_{M \times N} . \end{aligned}$$

To verify the composition law for $\phi \boxtimes \chi$, we use the definitions

$$\begin{aligned} \text{ftn}_{K \times L} &= (\nabla_1^{\uparrow K} ; \text{ftn}_K)^{:K^{K^\bullet \times L^\bullet} \Rightarrow K^\bullet} \boxtimes (\nabla_2^{\uparrow L} ; \text{ftn}_L)^{:L^{K^\bullet \times L^\bullet} \Rightarrow L^\bullet} , \\ \text{ftn}_{M \times N} &= (\nabla_1^{\uparrow M} ; \text{ftn}_M) \boxtimes (\nabla_2^{\uparrow N} ; \text{ftn}_N) . \end{aligned}$$

Denote $\psi \triangleq \phi \boxtimes \chi$ for brevity. The required law is

$$\text{ftn}_{K \times L} ; \psi = \psi^{\uparrow(K \times L)} ; \psi ; \text{ftn}_{M \times N} .$$

The left-hand side of this law is

$$\begin{aligned} \text{ftn}_{K \times L} ; \psi & \\ \text{definition of } \text{ftn}_{M \times N} &: = (\nabla_1^{\uparrow K} ; \text{ftn}_K) \boxtimes (\nabla_2^{\uparrow L} ; \text{ftn}_L) ; \phi \boxtimes \chi \\ \text{composition law of } \boxtimes &: = (\nabla_1^{\uparrow K} ; \text{ftn}_K ; \phi) \boxtimes (\nabla_2^{\uparrow L} ; \text{ftn}_L ; \chi) \end{aligned}$$

The right-hand side is

$$\begin{aligned} \psi^{\uparrow(K \times L)} ; \psi ; \text{ftn}_{M \times N} & \\ \text{lifting to } K \times L &: = (\psi^{\uparrow K} \boxtimes \psi^{\uparrow L}) ; (\phi \boxtimes \chi) ; \underline{\text{ftn}_{M \times N}} \\ \text{definition of } \text{ftn}_{M \times N} &: = \psi^{\uparrow K} \boxtimes \psi^{\uparrow L} ; \phi \boxtimes \chi ; (\nabla_1^{\uparrow M} ; \text{ftn}_M) \boxtimes (\nabla_2^{\uparrow N} ; \text{ftn}_N) \\ \text{composition law of } \boxtimes &: = (\psi^{\uparrow K} ; \phi ; \nabla_1^{\uparrow M} ; \text{ftn}_M) \boxtimes (\psi^{\uparrow L} ; \chi ; \nabla_2^{\uparrow N} ; \text{ftn}_N) . \end{aligned}$$

Consider now the first part of the pair product in the last line:

$$\begin{aligned}
 & \underline{\psi^{\uparrow K} ; \phi ; \nabla_1^{\uparrow M}} ; \text{ftn}_M \\
 \text{naturality of } \phi : &= \phi ; \underline{\psi^{\uparrow M} ; \nabla_1^{\uparrow M}} ; \text{ftn}_M \\
 \text{projection law of } \nabla_1 : &= \phi ; \underline{\nabla_1^{\uparrow M}} ; \phi^{\uparrow M} ; \text{ftn}_M \\
 \text{naturality of } \phi : &= \nabla_1^{\uparrow K} ; \underline{\phi ; \phi^{\uparrow M}} ; \text{ftn}_M \\
 \text{composition law of } \phi : &= \nabla_1^{\uparrow K} ; \text{ftn}_K ; \phi .
 \end{aligned}$$

In the same way, we find that the second part of the pair product is $\nabla_2^{\uparrow L} ; \text{ftn}_L ; \chi$, and so the composition law holds.

Statement 13.3.4.6 For any monads M and N , the projection function $\nabla_1 : M^\bullet \times N^\bullet \rightsquigarrow M^\bullet$ is a monadic morphism. Same for $\nabla_2 : M^\bullet \times N^\bullet \rightsquigarrow N^\bullet$.

Proof It is sufficient to verify the laws for ∇_1 ; the proof for ∇_2 will be analogous. The identity law:

$$\begin{aligned}
 \text{expect to equal } \text{pu}_M : & \text{pu}_{M \times N} ; \nabla_1 \\
 \text{definition of } \text{pu}_{M \times N} : &= (\text{pu}_M \boxtimes \text{pu}_N) ; \nabla_1 \\
 \text{projection law of } \nabla_1 : &= \text{pu}_M .
 \end{aligned}$$

The composition law:

$$\begin{aligned}
 \text{expect to equal } \nabla_1 ; \nabla_1^{\uparrow M} ; \text{ftn}_M : & \underline{\text{ftn}_{M \times N} ; \nabla_1} \\
 \text{definition of } \text{ftn}_{M \times N} : &= (\nabla_1^{\uparrow M} ; \text{ftn}_M) \boxtimes (\nabla_2^{\uparrow N} ; \text{ftn}_N) ; \underline{\nabla_1} \\
 \text{projection law of } \nabla_1 : &= \nabla_1 ; \nabla_1^{\uparrow M} ; \text{ftn}_M .
 \end{aligned}$$

Statement 13.3.4.7 For any monads M and N , the component-swapping function $\sigma : M^\bullet \times N^\bullet \rightsquigarrow N^\bullet \times M^\bullet$ is a monadic morphism.

Proof The code for σ can be written as a combination of other functions as $\sigma = \Delta ; (\nabla_2 \boxtimes \nabla_1)$. The functions Δ , ∇_1 , and ∇_2 are monadic morphisms by Statements 13.3.4.4 and 13.3.4.6. The function product $\nabla_1 \boxtimes \nabla_2$ is a monadic morphism by Statement 13.3.4.5. So σ is a composition of monadic morphisms; by Statement 13.3.4.3, σ is a monadic morphism.

13.3.5 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 .

A useful property of monad transformers 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

$$T^{M^T L^M L^A} .$$

The properties of the transformer allow us to convert this type to a single layer of the transformed monad T . In this example, we will have a natural transformation

$$T^{M^T L^M L^A} \Rightarrow T^A .$$

To achieve this, we first use the methods `blift` and `lift` 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 `ftnT` as many times as necessary to flatten all nested layers of T into a single layer. The result is a value of type T^A .

13.3.6 Stacking two monads

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 $R^A \triangleq T_P^{Q,A}$. What would be the monad transformer T_R for the monad R ?

A simple solution is to first transform the foreign monad M with T_Q , obtaining a new monad $T_Q^{M,\bullet}$, and then to transform that new monad with T_P . So the formula for the transformer T_R is

$$T_R^{M,A} = T_P^{T_Q^{M,\bullet},A} .$$

Here the monad $T_Q^{M,\bullet}$ was 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” composition 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 since, in general, there will be no monadic isomorphism between monads stacked in a different order.

We will now show that the transformer T_R is lawful (satisfies all five laws shown in Section 13.3.1), as long as both T_P and T_Q satisfy the same five 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 $G^A \triangleq T_P^{M,A}$.

Monad construction law We need to show that $T_P^{T_Q^M}$ is a monad for any monad M . The monad construction law for T_Q says that T_Q^M is a monad. The monad construction law for T_P says 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, as required.

Identity law We need to show that $T_P^{T_Q^{\text{Id}}} \cong T_P^Q$ via a monadic isomorphism. The identity law for T_Q says that $T_Q^{\text{Id}} \cong Q$ via a monadic isomorphism. So, we already have a monadic morphism $\phi : Q \rightsquigarrow T_Q^{\text{Id}}$ and its inverse, $\chi : T_Q^{\text{Id}} \rightsquigarrow Q$. The runner mrun_P for T_P can be applied to both ϕ and χ since they are monadic morphisms. So we obtain two new monadic morphisms,

$$\text{mrun}_P(\phi) : T_P^Q \rightsquigarrow T_P^{T_Q^{\text{Id}}} ; \quad \text{mrun}_P(\chi) : T_P^{T_Q^{\text{Id}}} \rightsquigarrow T_P^Q .$$

Are these two monadic morphisms inverses of each other? To show this, we need to verify that

$$\text{mrun}_P(\phi) ; \text{mrun}_P(\chi) = \text{id} , \quad \text{mrun}_P(\chi) ; \text{mrun}_P(\phi) = \text{id} .$$

By the runner law for T_P , we have $\text{mrun}_P(f) ; \text{mrun}_P(g) = \text{mrun}_P(f ; g)$ for any two monadic morphisms f and g . We also have $\text{mrun}_P(\text{id}) = \text{id}$ by the same law. So,

$$\begin{aligned} \text{mrun}_P(\phi) ; \text{mrun}_P(\chi) &= \text{mrun}_P(\phi ; \chi) = \text{mrun}_P(\text{id}) = \text{id} , \\ \text{mrun}_P(\chi) ; \text{mrun}_P(\phi) &= \text{mrun}_P(\chi ; \phi) = \text{mrun}_P(\text{id}) = \text{id} . \end{aligned}$$

We have indeed obtained a monadic isomorphism between T_P^Q and $T_P^{T_Q^{\text{Id}}}$.

Lifting law We need to show that there exists a monadic morphism $M \rightsquigarrow T_P^M$ for any monad M . The lifting law for T_Q gives a monadic morphism $\text{lift}_Q : M \rightsquigarrow T_Q^M$. The lifting law for T_P can be applied to the monad T_Q^M , which gives a monadic morphism

$$\text{lift}_P : T_Q^M \rightsquigarrow T_P^{T_Q^M} .$$

The composition of this with lift_Q is a monadic morphism of the required type $M \rightsquigarrow T_P^M$. (A composition of monadic morphisms is again a monadic morphism by Statement 13.3.4.3.)

Runner law We need to show that there exists a lawful lifting

$$\text{mrun}_R : (M \rightsquigarrow N) \Rightarrow T_P^{T_Q^M} \rightsquigarrow T_P^{T_Q^N} .$$

First, we have to define $\text{mrun}_R \phi$ for any given $\phi : M \rightsquigarrow N$. We use the lifting law for T_Q to get a monadic morphism

$$\text{lift}_Q \phi : T_Q^M \rightsquigarrow T_Q^N .$$

Now we can apply the lifting law for T_P to this monadic morphism and obtain

$$\text{lift}_P (\text{lift}_Q \phi) : T_P^{T_Q^M} \rightsquigarrow T_P^{T_Q^N} .$$

This function has the correct type signature. So we can define

$$\text{lift}_R \triangleq \text{lift}_Q ; \text{lift}_P = \text{lift}_P \circ \text{lift}_Q .$$

It remains to prove that lift_R is a lawful lifting. We use the fact that both lift_P and lift_Q are lawful liftings; we need to show that their composition is also a lawful lifting. To verify the identity law of lifting, apply lift_R to an identity function $\text{id} : M \rightsquigarrow M$,

$$\begin{aligned} \text{lift}_R (\text{id}^{M \rightsquigarrow M}) &= \text{lift}_P (\text{lift}_Q \text{id}^{M \rightsquigarrow M}) \\ \text{identity law for lift}_Q : &= \text{lift}_P (\text{id}^{T_Q^M \rightsquigarrow T_Q^M}) \\ \text{identity law for lift}_P : &= \text{id} . \end{aligned}$$

To verify the composition law of lifting, apply lift_R to a composition of two monadic morphisms $\phi : L \rightsquigarrow M$ and $\chi : M \rightsquigarrow N$,

$$\begin{aligned} \text{lift}_R (\phi ; \chi) &= \text{lift}_P (\text{lift}_Q (\phi ; \chi)) \\ \text{composition law for lift}_Q : &= \text{lift}_P (\text{lift}_Q \phi ; \text{lift}_Q \chi) \\ \text{composition law for lift}_P : &= \text{lift}_P (\text{lift}_Q \phi) ; \text{lift}_P (\text{lift}_Q \chi) \\ \text{definition of lift}_R : &= \text{lift}_R \phi ; \text{lift}_R \chi . \end{aligned}$$

Base runner law We need to show that for any monadic morphism $\theta : T_P^Q \rightsquigarrow \text{Id}$ and for any monad M , there exists a monadic morphism $\text{brun}_R \theta : T_P^{T_Q^M} \rightsquigarrow M$. To define this morphism for a given θ , we clearly need to use the base runners for T_P and T_Q . The base runner for T_Q has the type signature

$$\text{brun}_Q : (Q \rightsquigarrow \text{Id}) \Rightarrow T_Q^M \rightsquigarrow M .$$

We can apply the base runner for T_P to T_Q^M as the foreign monad,

$$\text{brun}_P : (P \rightsquigarrow \text{Id}) \Rightarrow T_P^{T_Q^M} \rightsquigarrow T_Q^M .$$

It is now clear that we could obtain a monadic morphism $T_P^{T_Q^M} \rightsquigarrow M$ if we had some monadic morphisms $\phi : P \rightsquigarrow \text{Id}$ and $\chi : Q \rightsquigarrow \text{Id}$,

$$\text{brun}_P \phi ; \text{brun}_Q \chi : T_P^{T_Q^M} \rightsquigarrow M .$$

However, we are only given a single monadic 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 $\text{lift}_P : Q \rightsquigarrow T_P^Q$, which are both monadic morphisms, and compose them with θ :

$$(\text{blift}_P ; \theta) : P \rightsquigarrow \text{Id} ; \quad (\text{lift}_P ; \theta) : Q \rightsquigarrow \text{Id} .$$

So we can define the monadic morphism $\text{brun}_R\theta$ as

$$\begin{aligned}\text{brun}_R\theta : T_P^{T_Q^M} &\rightsquigarrow M \quad , \\ \text{brun}_R\theta &\triangleq \text{brun}_P(\text{blift}_P ; \theta) ; \text{brun}_Q(\text{lift}_P ; \theta) \quad .\end{aligned}$$

Since we have defined $\text{brun}_R\theta$ as a composition of monadic morphisms, $\text{brun}_R\theta$ is a monadic morphism by Statement 13.3.4.3.

To verify the non-degeneracy law of the base runner, $\text{lift}_R ; \text{brun}_R\theta = \text{id}$, we need to use the non-degeneracy laws for the base runners of T_P and T_Q , which are

$$\text{lift}_P ; \text{brun}_P \chi^{P \rightsquigarrow \text{id}} = \text{id} \quad , \quad \text{lift}_Q ; \text{brun}_Q \psi^{Q \rightsquigarrow \text{id}} = \text{id} \quad .$$

Then we can write

$$\begin{aligned}&\underline{\text{lift}_R ; \text{brun}_R\theta} \\ \text{expand definitions : } &= \underline{\text{lift}_Q ; \underline{\text{lift}_P ; \text{brun}_P(\text{blift}_P ; \theta) ; \text{brun}_Q(\text{lift}_P ; \theta)}} \\ \text{non-degeneracy for } \text{brun}_P : &= \underline{\text{lift}_Q ; \underline{\text{brun}_Q(\text{lift}_P ; \theta)}} \\ \text{non-degeneracy for } \text{brun}_Q : &= \text{id} \quad .\end{aligned}$$

13.3.7 Stacking any number of monads

The monad transformer for T_P^Q can be applied to another monad K ; the result is the transformed monad

$$S^A \triangleq T_P^{T_Q^K A}.$$

What is the monad transformer for the monad S ? Assuming that we know the monad transformer T_K , we could stack the transformers one level higher:

$$T_S^{M,A} \triangleq T_P^{T_Q^{T_K^M A}}.$$

This looks like a stack of four monads P , Q , K , and M . Note that the type parameter A is used as $T_P^{(\dots),A}$, that is, it belongs to the *outer* transformer T_P .

We can now define a transformer stack for any number of monads P, Q, \dots, Z in a similar way,

$$T_S^{M,A} \triangleq T_P^{T_Q^{T_Z^{(\dots)^M A}}} \quad . \tag{13.4}$$

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 derived the transformer laws for any *three* stacked monads (two monads P, Q within the transformer and one foreign monad M). Now we need to derive the same laws for a general transformer stack, such as that in Eq. (13.4). Let us temporarily denote by J the monad

$$J \triangleq T_Q^{T_Z^{(\dots)^{T_Z^{\text{Id}}}}},$$

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

$$T_J^M \triangleq T_Q^{T_Z^{(\dots)^{T_Z^M}}}.$$

Since both T_P and T_J are lawful transformers, their stacking composition $T_P^{T_J^M}$ is also a lawful transformer (this was shown in the Section 13.3.6). In our notation, $T_S^{M,A} = T_P^{T_J^M,A}$, and so we have shown that $T_S^{M,A}$ is a lawful transformer.

13.4 Monad transformers via functor composition: General properties

We have seen examples of monad transformers that work via functor composition, either as composed-inside or as composed-outside. The simplest examples are the `OptionT` transformer,

$$L^A \triangleq \mathbb{1} + A, \quad T_L^{M,A} \triangleq M^{L^A} = M^{\mathbb{1}+A} ,$$

which puts the base monad L *inside* the monad M , and the `ReaderT` transformer,

$$L^A \triangleq R \Rightarrow A, \quad T_L^{M,A} \triangleq L^{M^A} = R \Rightarrow M^A ,$$

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 $T^\bullet \triangleq L^{M^\bullet}$ 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 generic in the foreign monad. This is expressed via the monad transformer laws for the runners `mrun` and `brun`, which we will need to derive separately for each of the two kinds of transformers.

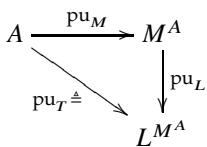
13.4.1 Motivation for the swap function

The first task is to show that the composed monad $T^\bullet \triangleq L^{M^\bullet}$ obeys the monad laws. For this, we need to define the methods for the monad T , namely `pure` (short notation “`puT`”) and `flatten` (short notation “`ftnT`”), with the type signatures

$$\text{pu}_T : A \Rightarrow L^{M^A} , \quad \text{ftn}_T : L^{M^{L^M}} \Rightarrow L^{M^A} .$$

How can we implement these methods? All we know about L and M is that they are monads with their own methods `puL`, `ftnL`, `puM`, and `ftnM`. We can easily implement

$$\text{pu}_T \triangleq \text{pu}_M \circ \text{pu}_L . \tag{13.5}$$



It remains to implement `ftnT`. In the type $L^{M^{L^M}}$, we have two layers of the functor L and two layers of the functor M . We could use the available method `ftnL` to flatten the two layers of L if we could

somewhat bring these nested layers together. However, these layers are separated by a layer of the functor M . To show this layered structure in a more visual way, let us employ another notation for the functor composition,

$$L \circ M \triangleq L^{M^*} .$$

In this notation, the type signature for `flatten` is written as

$$\text{ftn}_T : L \circ M \circ L \circ M \rightsquigarrow L \circ M .$$

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 achieve this? The trick is to *assume* that we have a function called `swap` (short notation “sw”), which can interchange the order of the layers. The type signature of `swap` is

$$\text{sw} : M \circ L \rightsquigarrow L \circ M ,$$

which is equivalently written in a more verbose notation as

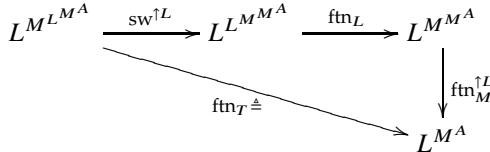
$$\text{sw} : M^{L^A} \Rightarrow L^{M^A} .$$

If this operation were *somewhat* 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

$$L \circ M \circ L \circ M \rightsquigarrow L \circ L \circ M \circ M$$

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

$$\text{ftn}_T \triangleq \text{sw}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} . \quad (13.6)$$



It turns out that in *both* cases (the composed-inside and the composed-outside transformers), the new monad's `flatten` method can be defined through the `swap` operation. For the two kinds of transformers, the type signatures of these functions are

$$\text{composed-inside} : \text{ftn}_T : M^{L^{M^A}} \Rightarrow M^{L^A} , \quad \text{sw} : L^{M^A} \Rightarrow M^{L^A} ,$$

$$\text{composed-outside} : \text{ftn}_T : M^{L^{M^A}} \Rightarrow L^{M^A} , \quad \text{sw} : M^{L^A} \Rightarrow L^{M^A} .$$

The difference between the operations `swap` and `sequence` There is a certain similarity between the `swap` operation introduced here and the `sequence` operation introduced in Chapter 11 for traversable functors. Indeed, the type signature of the sequence operation is

$$\text{seq} : L^{F^A} \Rightarrow F^{L^A} ,$$

where F is an arbitrary applicative functor (which could be M , since monads are applicative functors) and L is a traversable functor. However, the similarity stops here. The laws required for the `swap` operation to yield a monad T are different from the laws of traversable functors. In particular, if we wish M^{L^*} to be a monad, it is insufficient to require the monad L to be a traversable functor. A simple counterexample is found with $L^A \triangleq A \times A$ and $M^A \triangleq 1 + A$. Both L and M are traversable (since they are polynomial functors); but their composition $Q^A \triangleq 1 + A \times A$ is not a monad.

Another difference between `swap` and `sequence` is that the `swap` operation needs to be generic in the foreign monad, which may be either L or M according to the type of the monad transformer; whereas `sequence` is always generic in the applicative functor F .

To avoid confusion, I use the name “`swap`” rather than “`sequence`” for the function $\text{sw}_{L,M} : M^{L^\bullet} \rightsquigarrow L^{M^\bullet}$ in the context of monad transformers. Let us now find out what laws are required for the `swap` operation.¹

13.4.2 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 $f : A \Rightarrow B$,

$$f^{\uparrow L \uparrow M} ; \text{sw} = \text{sw} ; f^{\uparrow M \uparrow L} . \quad (13.7)$$

$$\begin{array}{ccc} M^{L^A} & \xrightarrow{f^{\uparrow L \uparrow M}} & M^{L^B} \\ \text{sw} \downarrow & & \downarrow \text{sw} \\ L^{M^A} & \xrightarrow{f^{\uparrow M \uparrow L}} & L^{M^B} \end{array}$$

To derive further laws for `swap`, consider the requirement that the transformed monad T should satisfy the monad laws:

$$\begin{aligned} \text{pu}_T ; \text{ftn}_T &= \text{id} , & \text{pu}_T^{\uparrow T} ; \text{ftn}_T &= \text{id} , \\ \text{ftn}_T^{\uparrow T} ; \text{ftn}_T &= \text{ftn}_T ; \text{ftn}_T . \end{aligned}$$

Additionally, T must satisfy the laws of a monad transformer. 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. (13.6).

We substitute Eq. (13.6) into the left identity law for ftn_T and simplify:

$$\begin{aligned} \text{id} &= \text{pu}_T ; \text{ftn}_T \\ \text{replace } \text{ftn}_T \text{ using Eq. (13.6)} : &= \text{pu}_T ; \text{sw}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} \\ \text{replace } \text{pu}_T \text{ using Eq. (13.5)} : &= \text{pu}_M ; \text{pu}_L ; \text{sw}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} \\ \text{naturality of } \text{pu}_L : &= \text{pu}_M ; \text{sw} ; \text{pu}_L ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} \\ \text{left identity law for } L : &= \text{pu}_M ; \text{sw} ; \text{ftn}_M^{\uparrow L} . \end{aligned} \quad (13.8)$$

How could the last expression in Eq. (13.8) be equal to `id`? We know nothing about the `pure` and `flatten` methods of the monads L and M , except that these methods satisfy their monad laws. We could satisfy the law in Eq. (13.8) if we somehow reduce that expression to

$$\text{pu}_M^{\uparrow L} ; \text{ftn}_M^{\uparrow L} = (\text{pu}_M ; \text{ftn}_M)^{\uparrow L} = \text{id} .$$

This will be possible only if we are able to interchange the order of function compositions with `sw` and eliminate `swap` from the expression. So, we must require the “outer-identity law” for `swap`,

$$\text{pu}_M ; \text{sw} = \text{pu}_M^{\uparrow L} . \quad (13.9)$$

¹The `swap` operation was used in a 1993 paper “Composing monads” by M. P. Jones and L. Duponcheel. They studied various ways of composing monads and also gave some arguments to show that no generic transformer could compose all monads L, M . The impossibility of a generic monad composition is demonstrated by the `State` monad that, as I show in this chapter, does not compose with arbitrary other monads M – either from inside or from outside.

$$\begin{array}{ccc} L^A & \xrightarrow{\text{pu}_M} & M^{L^A} \\ & \searrow \text{pu}_M^{\uparrow L} & \downarrow \text{sw} \\ & & L^{M^A} \end{array}$$

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. (13.8) as

$$\begin{aligned} \text{outer-identity law for } \text{sw} : & \quad = \text{pu}_M^{\uparrow L} ; \text{ftn}_M^{\uparrow L} \\ \text{functor composition law for } L : & \quad = (\text{pu}_M ; \text{ftn}_M)^{\uparrow L} \\ \text{left identity law for } M : & \quad = \text{id}^{\uparrow L} \\ \text{functor identity law for } L : & \quad = \text{id} . \end{aligned}$$

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`,

$$\text{pu}_L^{\uparrow M} ; \text{sw} = \text{pu}_L . \quad (13.10)$$

$$\begin{array}{ccc} M^A & \xrightarrow{\text{pu}_L^{\uparrow M}} & M^{L^A} \\ & \searrow \text{pu}_L & \downarrow \text{sw} \\ & & L^{M^A} \end{array}$$

This law expresses the idea that a pure layer of the functor L remains pure after swapping the order of layers.

Assuming this law, we can derive the right identity law for T :

$$\begin{aligned} & \text{pu}_T^{\uparrow T} ; \text{ftn}_T \\ (\text{by definition, } f^{\uparrow T} \triangleq f^{\uparrow M \uparrow L}) : & \quad = (\text{pu}_T)^{\uparrow M \uparrow L} ; \text{ftn}_T \\ \text{definitions of } \text{pu}_T \text{ and } \text{ftn}_T : & \quad = \text{pu}_M^{\uparrow M \uparrow L} ; \underline{\text{pu}_L^{\uparrow M \uparrow L} ; \text{sw}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L}} \\ \text{inner-identity law for } \text{sw, under } \uparrow L : & \quad = \text{pu}_M^{\uparrow M \uparrow L} ; \underline{\text{pu}_L^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L}} \\ \text{right identity law for } L : & \quad = \text{pu}_M^{\uparrow M \uparrow L} ; \text{ftn}_M^{\uparrow L} = \underline{(\text{pu}_M ; \text{ftn}_M)^{\uparrow L}} \\ \text{right identity law for } M : & \quad = \text{id}^{\uparrow L} = \text{id} . \end{aligned}$$

Deriving the monad associativity law for T ,

$$\text{ftn}_T^{\uparrow T} ; \text{ftn}_T = \text{ftn}_T ; \text{ftn}_T ,$$

turns out to require *two* further laws for `swap`. Let us see why.

Substituting the definition of ftn_T into the associativity law, we get

$$\begin{aligned} & (\text{sw}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L})^{\uparrow M \uparrow L} ; \underline{\text{sw}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L}} \\ & = \text{sw}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} ; \underline{\text{sw}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L}} . \end{aligned} \quad (13.11)$$

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 monads L and M .

Let us look for such interchange laws. One possibility is to have a law involving $\text{ftn}_M \circ \text{sw}$, which is a function of type $M^{M^{LA}} \Rightarrow L^{MA}$ or, in another 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

$$\text{ftn}_M \circ \text{sw} = \text{sw}^{\uparrow M} \circ \text{sw} \circ \text{ftn}_M^{\uparrow L} . \quad (13.12)$$

$$\begin{array}{ccccc} & & M^{M^{LA}} & \xrightarrow{\text{ftn}_M} & M^{LA} \\ & \swarrow \text{sw}^{\uparrow M} & & & \downarrow \text{sw} \\ M^{LA} & \xrightarrow{\text{sw}} & L^{M^{MA}} & \xrightarrow{\text{ftn}_M^{\uparrow L}} & L^{MA} \end{array}$$

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

$$\text{ftn}_L^{\uparrow M} \circ \text{sw} = \text{sw} \circ \text{sw}^{\uparrow L} \circ \text{ftn}_L . \quad (13.13)$$

$$\begin{array}{ccccc} & & M^{LA} & \xrightarrow{\text{ftn}_L^{\uparrow M}} & M^{LA} \\ & \swarrow \text{sw} & & & \downarrow \text{sw} \\ L^{LA} & \xrightarrow{\text{sw}^{\uparrow L}} & L^{L^{MA}} & \xrightarrow{\text{ftn}_L} & L^{MA} \end{array}$$

At this point, we have simply written down these two interchange 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. (13.11) 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 transform both sides of Eq. (13.11) into a similar form, hoping to prove that they are equal.

We begin with the right-hand side of Eq. (13.11) 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:

$$\begin{aligned} & \text{sw}^{\uparrow L} \circ \text{ftn}_L \circ \text{ftn}_M^{\uparrow L} \circ \text{sw}^{\uparrow L} \circ \text{ftn}_L \circ \text{ftn}_M^{\uparrow L} \\ \text{composition for } L : & = \text{sw}^{\uparrow L} \circ \text{ftn}_L \circ (\text{ftn}_M \circ \text{sw})^{\uparrow L} \circ \text{ftn}_L \circ \text{ftn}_M^{\uparrow L} \\ \text{outer-interchange for } \text{sw} : & = \text{sw}^{\uparrow L} \circ \text{ftn}_L \circ (\text{sw}^{\uparrow M} \circ \text{sw} \circ \text{ftn}_M^{\uparrow L})^{\uparrow L} \circ \text{ftn}_L \circ \text{ftn}_M^{\uparrow L} \\ \text{composition for } L : & = \text{sw}^{\uparrow L} \circ \text{ftn}_L \circ \text{sw}^{\uparrow M \uparrow L} \circ \text{sw}^{\uparrow L} \circ \text{ftn}_M^{\uparrow L \uparrow L} \circ \text{ftn}_L \circ \text{ftn}_M^{\uparrow L} \\ \text{naturality of } \text{ftn}_L : & = \text{sw}^{\uparrow L} \circ \text{ftn}_L \circ (\text{sw}^{\uparrow M} \circ \text{sw})^{\uparrow L} \circ \text{ftn}_L \circ \text{ftn}_M^{\uparrow L} \circ \text{ftn}_M^{\uparrow L} \\ \text{naturality of } \text{ftn}_L : & = \text{sw}^{\uparrow L} \circ (\text{sw}^{\uparrow M} \circ \text{sw})^{\uparrow L \uparrow L} \circ \text{ftn}_L \circ \text{ftn}_L \circ \text{ftn}_M^{\uparrow L} \circ \text{ftn}_M^{\uparrow L} . \end{aligned}$$

Now all `swaps` are on the left and all `flattens` on the right of the expression.

Transform the right-hand side of Eq. (13.11) in the same way as

$$\begin{aligned}
 & \left(\text{sw}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} \right)^{\uparrow M \uparrow L} ; \text{sw}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} \\
 \text{functor composition : } &= \left(\text{sw}^{\uparrow L} ; \text{ftn}_L \right)^{\uparrow M \uparrow L} ; \left(\text{ftn}_M^{\uparrow L \uparrow M} ; \text{sw} \right)^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} \\
 \text{naturality of sw : } &= \left(\text{sw}^{\uparrow L} ; \text{ftn}_L \right)^{\uparrow M \uparrow L} ; \left(\text{sw} ; \text{ftn}_M^{\uparrow M \uparrow L} \right)^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} \\
 \text{naturality of ftn}_L : &= \text{sw}^{\uparrow L \uparrow M \uparrow L} ; \text{ftn}_L^{\uparrow M \uparrow L} ; \text{sw}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow M \uparrow L} ; \text{ftn}_M^{\uparrow L} \\
 \text{associativity of ftn}_M : &= \text{sw}^{\uparrow L \uparrow M \uparrow L} ; \left(\text{ftn}_L^{\uparrow M} ; \text{sw} \right)^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} ; \text{ftn}_M^{\uparrow L} \\
 \text{inner-interchange for sw : } &= \text{sw}^{\uparrow L \uparrow M \uparrow L} ; \left(\text{sw} ; \text{sw}^{\uparrow L} ; \text{ftn}_L \right)^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} ; \text{ftn}_M^{\uparrow L} \\
 \text{associativity of ftn}_L : &= \left(\text{sw}^{\uparrow L \uparrow M} ; \text{sw} ; \text{sw}^{\uparrow L} \right)^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} ; \text{ftn}_M^{\uparrow L} .
 \end{aligned}$$

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

$$\text{sw}^{\uparrow L} ; \left(\text{sw}^{\uparrow M} ; \text{sw} \right)^{\uparrow L \uparrow L} = \left(\text{sw}^{\uparrow L \uparrow M} ; \text{sw} ; \text{sw}^{\uparrow L} \right)^{\uparrow L} .$$

or equivalently

$$\left(\text{sw} ; \text{sw}^{\uparrow M \uparrow L} ; \text{sw}^{\uparrow L} \right)^{\uparrow L} = \left(\text{sw}^{\uparrow L \uparrow M} ; \text{sw} ; \text{sw}^{\uparrow L} \right)^{\uparrow L} .$$

The two sides are equal due to the naturality law of `swap`,

$$\text{sw} ; \text{sw}^{\uparrow M \uparrow L} = \text{sw}^{\uparrow L \uparrow M} ; \text{sw}.$$

This completes the proof of the following theorem:

Theorem 13.4.2.1 If two monads L and M are such that there exists a function

$$\text{sw}_{L,M} : M^{L^A} \Rightarrow L^{M^A}$$

(called “`swap`”), which is a natural transformation satisfying four additional laws:

$$\begin{aligned}
 \text{outer-identity : } & \text{pu}_L^{\uparrow M} ; \text{sw}_{L,M} = \text{pu}_L , \\
 \text{inner-identity : } & \text{pu}_M ; \text{sw}_{L,M} = \text{pu}_M^{\uparrow L} , \\
 \text{outer-interchange : } & \text{ftn}_L^{\uparrow M} ; \text{sw}_{L,M} = \text{sw}_{L,M} ; \text{sw}_{L,M}^{\uparrow L} ; \text{ftn}_L , \\
 \text{inner-interchange : } & \text{ftn}_M ; \text{sw}_{L,M} = \text{sw}_{L,M}^{\uparrow M} ; \text{sw}_{L,M} ; \text{ftn}_M^{\uparrow L} ,
 \end{aligned}$$

then the functor composition

$$T^A \triangleq L^{M^A}$$

is a monad with the methods `pure` and `flatten` defined by

$$\text{pu}_T \triangleq \text{pu}_M ; \text{pu}_L , \tag{13.14}$$

$$\text{ftn}_T \triangleq \text{sw}_{L,M}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} . \tag{13.15}$$

13.4.3 Intuition behind the laws of swap

The interchange laws for `swap` guarantee that any functor composition built up from L and M , e.g. like this,

$$M \circ M \circ L \circ M \circ L \circ L \circ M \circ M \circ L ,$$

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 flattening or interchange 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 $L\text{-flatMaps}$ 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 “linear” and the “rigid” monads) are defined via `swap`.

13.4.4 Deriving swap from flatten

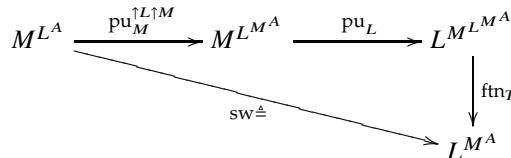
We have shown that the `flatten` method of the monad $T^\bullet = L^{M^\bullet}$ can be defined via the `swap` method. However, we have seen examples of some composable monads (such as `Reader` and `Option`) where we already know the definitions of the `flatten` method for the composed monad T . Does 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 establish whether a `swap` function exists such that the given `flatten` function is expressed via Eq. (13.6)?

To answer this question, let us look at the type signature of `flatten` for T :

$$\text{ftn}_T : L \circ M \circ L \circ M \rightsquigarrow L \circ M .$$

This type signature is different from $\text{sw} : M \circ L \rightsquigarrow L \circ M$ only because the argument of ftn_T has extra layers of the functors L and M that are placed outside the $M \circ L$ composition. We can use the `pure` methods of M and L to add these 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

$$\text{sw} = \text{pu}_M^{\uparrow L \uparrow M} ; \text{pu}_L ; \text{ftn}_T . \quad (13.16)$$



We have expressed ftn_T and sw through each other. Are these functions always equivalent? To decide this, we need to answer two questions:

1. If we first define ftn_T using Eq. (13.6) through a given implementation of `swap` and then substitute that ftn_T into Eq. (13.16), will we always recover the initially given function sw ? (Yes, assuming naturality for `swap`.)
2. If we first define sw using Eq. (13.16) through a given implementation of ftn_T and then substitute that sw into Eq. (13.6), will we always recover the initially given function ftn_T ? (No, not without additional laws for ftn_T .)

To answer the first question, substitute ftn_T from Eq. (13.6) into Eq. (13.16):

$$\begin{aligned}
 & \text{substitute } \text{ftn}_T : = \text{pu}_M^{\uparrow L \uparrow M} ; \underline{\text{pu}_L} ; \underline{\text{ftn}_T} \\
 & \text{naturality of } \text{pu}_L : = \text{pu}_M^{\uparrow L \uparrow M} ; \underline{\text{sw}} ; \underline{\text{pu}_L} ; \underline{\text{ftn}_L} ; \underline{\text{ftn}_M^{\uparrow L}} \\
 & \text{left identity law for } L : = \underline{\text{pu}_M^{\uparrow L \uparrow M}} ; \underline{\text{sw}} ; \underline{\text{ftn}_M^{\uparrow L}} \\
 & \text{naturality of } \text{sw} : = \underline{\text{sw}} ; \underline{\text{pu}_M^{\uparrow M \uparrow L}} ; \underline{\text{ftn}_M^{\uparrow L}} \\
 & \text{functor composition for } L : = \underline{\text{sw}} ; \underline{(\text{pu}_M^{\uparrow M} ; \text{ftn}_M)^{\uparrow L}} \\
 & \text{right identity law for } M : = \underline{\text{sw}} \quad .
 \end{aligned}$$

So, indeed, we always recover the initial `swap` function.

To answer the second question, substitute `sw` from Eq. (13.16) into Eq. (13.6):

$$\begin{aligned}
 & \text{substitute } \text{sw} : = \underline{(\text{pu}_M^{\uparrow L \uparrow M} ; \text{pu}_L ; \text{ftn}_T)^{\uparrow L}} ; \underline{\text{ftn}_L} ; \underline{\text{ftn}_M^{\uparrow L}} \\
 & \text{functor composition} : = \text{pu}_M^{\uparrow L \uparrow M \uparrow L} ; \underline{\text{pu}_L^{\uparrow L}} ; \underline{\text{ftn}_T^{\uparrow L}} ; \underline{\text{ftn}_L} ; \underline{\text{ftn}_M^{\uparrow L}} \quad . \tag{13.17}
 \end{aligned}$$

At this point, we are stuck: we can find no laws to transform the last expression. Without assuming additional laws, it *does not follow* that the right-hand side of Eq. (13.17) is equal to ftn_T . Let us now derive those additional laws.

The only sub-expression in Eq. (13.17) that we could possibly transform is the composition $\text{ftn}_T^{\uparrow L} ; \text{ftn}_L$. So, we need to assume a law involving the expression

$$(\text{ftn}_T^{\uparrow L} ; \text{ftn}_L) : L \circ L \circ M \circ L \circ M \rightsquigarrow L \circ M \quad .$$

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 $(L \circ M)$:

$$(\text{ftn}_L ; \text{ftn}_T) : L \circ L \circ M \circ L \circ M \rightsquigarrow L \circ M \quad .$$

So we conjecture that a possibly useful additional law for ftn_T is

$$\text{ftn}_L ; \text{ftn}_T = \text{ftn}_T^{\uparrow L} ; \text{ftn}_L \quad .$$

$$\begin{array}{ccc}
 L^{L^M L^{M^A}} & \xrightarrow{\text{ftn}_L} & L^{M^{L^{M^A}}} \\
 \downarrow \text{ftn}_T^{\uparrow L} & & \downarrow \text{ftn}_T \\
 L^{L^{M^A}} & \xrightarrow{\text{ftn}_L} & L^{M^A}
 \end{array}$$

This law expresses a kind of “compatibility” between the monads L and T .

With this law, the right-hand side of Eq. (13.17) becomes

$$\begin{aligned}
 & \text{right identity law of } L : = \text{pu}_M^{\uparrow L \uparrow M \uparrow L} ; \underline{\text{pu}_L^{\uparrow L}} ; \underline{\text{ftn}_L} ; \underline{\text{ftn}_T} ; \underline{\text{ftn}_M^{\uparrow L}} \\
 & \text{right identity law of } M : = \text{pu}_M^{\uparrow L \uparrow M \uparrow L} ; \underline{\text{ftn}_T} ; \underline{\text{ftn}_M^{\uparrow L}} \quad .
 \end{aligned}$$

Again, we cannot proceed unless we assume a law involving the expression

$$(ftn_T \circ ftn_M^{\uparrow L}) : L \circ M \circ L \circ M \circ M \rightsquigarrow L \circ M .$$

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 :

$$(ftn_M^{\uparrow L \uparrow M \uparrow L} \circ ftn_T) : L \circ M \circ L \circ M \circ M \rightsquigarrow L \circ M .$$

The second conjectured law is therefore

$$ftn_T \circ ftn_M^{\uparrow L} = ftn_M^{\uparrow L \uparrow M \uparrow L} \circ ftn_T .$$

$$\begin{array}{ccc} L^{M^{LM^{MA}}} & \xrightarrow{ftn_T} & L^{M^{MA}} \\ ftn_M^{\uparrow L \uparrow M \uparrow L} \downarrow & & \downarrow ftn_M^{\uparrow L} \\ L^{M^{LM^{MA}}} & \xrightarrow{ftn_T} & L^{M^A} \end{array}$$

This law expresses a kind of “compatibility” between the monads M and T .

Assuming this law, we can finally complete the derivation:

$$\begin{aligned} & pu_M^{\uparrow L \uparrow M \uparrow L} \circ ftn_T \circ ftn_M^{\uparrow L} \\ \text{substitute the second conjecture : } & = pu_M^{\uparrow L \uparrow M \uparrow L} \circ ftn_M^{\uparrow L \uparrow M \uparrow L} \circ ftn_T \\ \text{functor composition : } & = (pu_M \circ ftn_M)^{\uparrow L \uparrow M \uparrow L} \circ ftn_T \\ \text{left identity law of } M : & = ftn_T . \end{aligned}$$

We recovered the initial ftn_T by assuming two additional laws.

It turns out that these additional laws will always hold when ftn_T is defined via `swap` (see Exercise 13.4.4.1).

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 (in practice, this is always the case), it is simpler to verify the laws of `swap` and then obtain the monad laws of $L \circ M$ via Theorem 13.4.2.1.

Exercise 13.4.4.1 Assuming that

- L and M are monads,
- the method `swap` is a natural transformation $M \circ L \rightsquigarrow L \circ M$,
- the method ftn_T of the monad $T = L \circ M$ is *defined* via `swap` by Eq. (13.6),

show that the two interchange laws must hold for ftn_T :

$$\begin{aligned} \text{inner-interchange : } & ftn_L \circ ftn_T = ftn_T^{\uparrow L} \circ ftn_L , \\ \text{outer-interchange : } & ftn_T \circ ftn_M^{\uparrow L} = ftn_M^{\uparrow L \uparrow M \uparrow L} \circ ftn_T . \end{aligned}$$

Exercise 13.4.4.2 With the same assumptions as Exercise 13.4.4.1 and additionally assuming the inner and outer identity laws for `swap` (see Theorem 13.4.2.1), show that the monad $T^\bullet \triangleq L^{M^\bullet}$ satisfies two “pure compatibility” laws,

$$\text{inner-pure-compatibility : } ftn_L = pu_M^{\uparrow L} \circ ftn_T : L^{L^{M^\bullet}} \Rightarrow L^{M^\bullet} ,$$

$$\text{outer-pure-compatibility : } ftn_M^{\uparrow L} = pu_L^{\uparrow T} \circ ftn_T : L^{M^{M^\bullet}} \Rightarrow L^{M^\bullet} ,$$

or, expressed equivalently through the flm methods instead of ftn ,

$$\begin{aligned}\text{flm}_L f^{A \Rightarrow L^M B} &= \text{pure}_M^{\uparrow L} ; \text{flm}_T f^{A \Rightarrow L^M B} , \\ (\text{flm}_M f^{A \Rightarrow M^B})^{\uparrow L} &= \text{pure}_L^{\uparrow T} ; \text{flm}_T (f^{\uparrow L}) .\end{aligned}$$

13.4.5 Monad transformer identity law: Proofs

The identity law requires that $T_L^M \cong L$ if $M = \text{Id}$. We will now prove this law assuming that pu_T and ftn_T are defined by Eqs. (13.14)–(13.15), and that the two identity laws of swap hold (see Theorem 13.4.2.1),

$$\begin{aligned}\text{outer-identity} : \quad \text{pu}_L^{\uparrow M} ; \text{sw}_{L,M} &= \text{pu}_L , \\ \text{inner-identity} : \quad \text{pu}_M ; \text{sw}_{L,M} &= \text{pu}_M^{\uparrow L} .\end{aligned}$$

Note that M is the foreign monad for composed-outside transformers, $T_L^M = L \circ M$. Setting $M = \text{Id}$ in the inner-identity law, we obtain

$$\text{pu}_{\text{Id}} ; \text{sw}_{L,\text{Id}} = \text{pu}_{\text{Id}}^{\uparrow L} .$$

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,

$$\text{pu}_{\text{Id}}^{\uparrow M} ; \text{sw}_{\text{Id},M} = \text{pu}_{\text{Id}} .$$

We obtain $\text{sw}_{\text{Id},M} = \text{id}$.

Note that $\text{sw}_{L,\text{Id}} : L^A \Rightarrow L^A$ is a natural transformation for a monad L , so one may heuristically expect $\text{sw}_{L,\text{Id}}$ to be equal to the identity map (the only natural transformation $L^A \Rightarrow L^A$ that exists for all monads L). Similarly, one may expect that $\text{sw}_{\text{Id},M} : M^A \Rightarrow M^A = \text{id}$ since it is a natural transformation. But these are only heuristic expectations, while we have just shown that the properties $\text{sw}_{L,\text{Id}} = \text{id}$ and $\text{sw}_{\text{Id},M} = \text{id}$ follow from the previously established laws of swap without any new assumptions. These properties will be needed in the proofs below.

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, $\text{id} : T_L^{\text{Id},A} \Rightarrow L^A$ and $\text{id} : L^A \Rightarrow T_L^{\text{Id},A}$.

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, $\text{id} : T_L^{\text{Id},A} \Rightarrow L^A$ and $\text{id} : L^A \Rightarrow T_L^{\text{Id},A}$.

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 *monadic* isomorphism (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. (13.14)–(13.15) 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 $\text{id} : T^A \Rightarrow L^A$ will satisfy the laws of the monadic morphism,

$$\text{pu}_T ; \text{id} = \text{pu}_L , \quad \text{ftn}_T ; \text{id} = \text{id}^{\uparrow T} ; \text{id} ; \text{ftn}_L .$$

In the same way, the laws of the monadic morphism will hold for the identity map 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. (13.14), as appropriate for the composed-inside transformer $T_L^M = M \circ L$. Setting the foreign monad to identity, $L = \text{Id}$, in Eq. (13.14) gives

$$\text{pu}_T = \text{pu}_M ; \text{pu}_{\text{Id}} = \text{pu}_M.$$

To show that $\text{ftn}_M = \text{ftn}_T$, we use Eq. (13.15) with $L = \text{Id}$:

$$\begin{aligned} & \text{ftn}_T \\ \text{use Eq. (13.15)} : &= \text{sw}_{\text{Id}, M}^{\uparrow \text{Id}} ; \text{ftn}_{\text{Id}} ; \text{ftn}_M^{\uparrow \text{Id}} \\ \text{use } \text{ftn}_{\text{Id}} = \text{id} \text{ and } \text{sw}_{\text{Id}, M} = \text{id} : &= \text{ftn}_M . \end{aligned}$$

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. (13.14), as appropriate for the composed-outside transformer $T_L^M = L \circ M$. Setting the foreign monad to identity, $M = \text{Id}$, in Eq. (13.14) gives

$$\text{pu}_T = \text{pu}_{\text{Id}} ; \text{pu}_L = \text{pu}_L.$$

To show that $\text{ftn}_L = \text{ftn}_T$, use Eq. (13.15) with $M = \text{Id}$:

$$\begin{aligned} & \text{ftn}_T \\ \text{use Eq. (13.15)} : &= \text{sw}_{L, \text{Id}}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_{\text{Id}}^{\uparrow L} \\ \text{use } \text{ftn}_{\text{Id}} = \text{id} \text{ and } \text{sw}_{L, \text{Id}} = \text{id} : &= \text{ftn}_L . \end{aligned}$$

13.4.6 Monad transformer lifting laws: Proofs

We will now derive the laws of monad transformer liftings from the laws of `swap`, using Eqs. (13.14)–(13.15) as definitions of the methods of T .

To be specific, let us assume that L is the base monad of the transformer. Only the monads' names will need to change for the other choice of the base monad.

The lifting morphisms of a compositional monad transformer are defined by

$$\begin{aligned} \text{lift} &= \text{pu}_L : M^A \Rightarrow L^{M^A} , \\ \text{blift} &= \text{pu}_M^{\uparrow L} : L^A \Rightarrow L^{M^A} . \end{aligned}$$

Their laws of liftings (the identity and the composition laws) are

$$\begin{aligned} \text{pu}_M ; \text{lift} &= \text{pu}_T , & \text{pu}_L ; \text{blift} &= \text{pu}_T , \\ \text{ftn}_M ; \text{lift} &= \text{lift}^{\uparrow M} ; \text{lift} ; \text{ftn}_T , & \text{ftn}_L ; \text{blift} &= \text{blift}^{\uparrow L} ; \text{blift} ; \text{ftn}_T . \end{aligned}$$

The identity laws are verified quickly,

$$\begin{aligned} \text{expect to equal } \text{pu}_T : & \text{pu}_M ; \text{lift} = \text{pu}_M ; \text{pu}_L \\ \text{definition of } \text{pu}_T : &= \text{pu}_T , \\ \text{expect to equal } \text{pu}_T : & \text{pu}_L ; \text{blift} = \text{pu}_L ; \text{pu}_M^{\uparrow L} \\ \text{naturality of } \text{pu}_L : &= \text{pu}_M ; \text{pu}_L = \text{pu}_T . \end{aligned}$$

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 lift:

$$\begin{aligned}
 & \text{expect to equal } \text{ftn}_M ; \text{pu}_L : \quad \text{lift}^{\uparrow M} ; \text{lift} ; \text{ftn}_T \\
 & \text{definitions of lift and ftn}_T : = \text{pu}_L^{\uparrow M} ; \underline{\text{pu}_L ; \text{sw}^{\uparrow L}} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} \\
 & \quad \text{naturality of } \text{pu}_L : = \text{pu}_L^{\uparrow M} ; \text{sw} ; \underline{\text{pu}_L ; \text{ftn}_L} ; \text{ftn}_M^{\uparrow L} \\
 & \quad \text{left identity law of } L : = \underline{\text{pu}_L^{\uparrow M} ; \text{sw}} ; \text{ftn}_M^{\uparrow L} \\
 & \quad \text{inner-identity law of sw} : = \text{pu}_L ; \text{ftn}_M^{\uparrow L} \\
 & \quad \text{naturality of } \text{pu}_L : = \text{ftn}_M ; \text{pu}_L .
 \end{aligned}$$

The composition law for blift:

$$\begin{aligned}
 & \text{expect to equal } \text{ftn}_L ; \text{pu}_M^{\uparrow L} : \quad \text{blift}^{\uparrow L} ; \text{blift} ; \text{ftn}_T \\
 & \text{definitions of blift and ftn}_T : = \text{pu}_M^{\uparrow L \uparrow L} ; \underline{\text{pu}_M^{\uparrow L} ; \text{sw}^{\uparrow L}} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} \\
 & \quad \text{functor composition in } L : = \text{pu}_M^{\uparrow L \uparrow L} ; \underline{(\text{pu}_M ; \text{sw})^{\uparrow L}} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} \\
 & \quad \text{outer-identity law of sw} : = \underline{(\text{pu}_M^{\uparrow L \uparrow L} ; \text{pu}_M^{\uparrow L \uparrow L})} ; \underline{\text{ftn}_L ; \text{ftn}_M^{\uparrow L}} \\
 & \quad \text{naturality of } \text{ftn}_L : = \text{ftn}_L ; (\text{pu}_M^{\uparrow L} ; \underline{\text{pu}_M^{\uparrow L}}) ; \text{ftn}_M^{\uparrow L} \\
 & \quad \text{right identity law of } M : = \text{ftn}_L ; \text{pu}_M^{\uparrow L} .
 \end{aligned}$$

So, the lifting laws for T follow from the laws of `swap`.

13.4.7 Monad transformer runner laws: Proofs

The laws of runners are not symmetric with respect to the base monad and the foreign monad: the runner, `mrun`, is generic 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, this law needs to be written differently, depending on the choice of the base monad. Let us consider separately the situations when either L or M is the base monad.

If the base monad is L , the runners are

$$\begin{aligned}
 \text{mrun } \phi^{M^\bullet \rightsquigarrow N^\bullet} : L^{M^\bullet} \rightsquigarrow L^{N^\bullet} , \quad \text{mrun } \phi = \phi^{\uparrow L} ; \\
 \text{brun } \theta^{L^\bullet \rightsquigarrow \bullet} : L^{M^\bullet} \rightsquigarrow M^\bullet , \quad \text{brun } \theta = \theta .
 \end{aligned}$$

The laws of runners require that $\text{mrun } \phi$ and $\text{brun } \theta$ must be monadic morphisms, i.e. the identity and composition laws must hold for $\text{mrun } \phi$ and $\text{brun } \theta$:

$$\begin{aligned}
 \text{pu}_{L \circ M} ; \text{mrun } \phi &= \text{pu}_{L \circ N} , \\
 \text{ftn}_{L \circ M} ; \text{mrun } \phi &= (\text{mrun } \phi)^{\uparrow M \uparrow L} ; \text{mrun } \phi ; \text{ftn}_{L \circ N} , \\
 \text{pu}_{L \circ M} ; \text{brun } \theta &= \text{pu}_M , \\
 \text{ftn}_{L \circ M} ; \text{brun } \theta &= (\text{brun } \theta)^{\uparrow M \uparrow L} ; \text{brun } \theta ; \text{ftn}_M .
 \end{aligned}$$

To derive these laws, we may use the identity and composition laws of monadic morphisms for ϕ and θ . We also use Eqs. (13.14)–(13.15) as definitions of the monad T . Additionally, the **monadic naturality** of `swap` with respect to ϕ and θ are assumed to hold,

$$\text{sw}_{L,M} ; \phi^{\uparrow L} = \phi ; \text{sw}_{L,N} , \quad \text{sw}_{L,M} ; \theta = \theta^{\uparrow M} .$$

$$\begin{array}{ccc}
 M^{L^A} & \xrightarrow{\text{SW}_{L,M}} & L^{M^A} \\
 \phi \downarrow & & \downarrow \phi^{\uparrow L} \\
 N^{L^A} & \xrightarrow[\text{SW}_{L,N}]{} & L^{N^A}
 \end{array}
 \quad
 \begin{array}{ccc}
 M^{L^A} & \xrightarrow{\text{SW}_{L,M}} & L^{M^A} \\
 & \searrow \theta^{\uparrow M} & \downarrow \theta \\
 & & M^A
 \end{array}$$

The first law to be shown is the identity law for mrun ϕ :

$$\begin{aligned}
 &\text{expect this to equal } \text{pu}_{L \circ N} : \quad \text{pu}_{L \circ M} ; \text{mrun } \phi \\
 &\text{definitions of mrun and } \text{pu}_{L \circ M} : = \text{pu}_M ; \underline{\text{pu}_L ; \phi^{\uparrow L}} \\
 &\text{naturality of } \text{pu}_L : = \underline{\text{pu}_M ; \phi ; \text{pu}_L} \\
 &\text{identity law for } \phi : = \text{pu}_N ; \text{pu}_L \\
 &\text{definition of } \text{pu}_{L \circ N} : = \text{pu}_{L \circ N} .
 \end{aligned}$$

The next law to be shown is the composition law for mrun ϕ :

$$\begin{aligned}
 &\text{expect this to equal } \text{ftn}_T ; \phi^{\uparrow L} : \quad (\text{mrun } \phi)^{\uparrow M \uparrow L} ; \text{mrun } \phi ; \text{ftn}_{L \circ N} \\
 &\text{definitions of mrun and } \text{ftn}_{L \circ N} : = \phi^{\uparrow L \uparrow M \uparrow L} ; \phi^{\uparrow L} ; \underline{\text{sw}_{L,N}^{\uparrow L}} ; \text{ftn}_L ; \text{ftn}_N^{\uparrow L} \\
 &\text{monadic naturality of } \text{sw}_{L,M} : = \underline{\phi^{\uparrow L \uparrow M \uparrow L} ; \text{sw}_{L,M}^{\uparrow L}} ; \phi^{\uparrow L \uparrow L} ; \text{ftn}_L ; \text{ftn}_N^{\uparrow L} \\
 &\text{naturality of } \text{sw}_{L,M} : = \underline{\text{sw}_{L,M}^{\uparrow L} ; \phi^{\uparrow M \uparrow L \uparrow L}} ; \phi^{\uparrow L \uparrow L} ; \text{ftn}_L ; \text{ftn}_N^{\uparrow L} \\
 &\text{naturality of } \text{ftn}_L : = \underline{\text{sw}_{L,M}^{\uparrow L} ; \text{ftn}_L ; (\phi^{\uparrow M} ; \phi ; \text{ftn}_N)^{\uparrow L}} \\
 &\text{composition law for } \phi : = \underline{\text{sw}_{L,M}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L}} ; \phi^{\uparrow L} \\
 &\text{definition of } \text{ftn}_T : = \underline{\text{ftn}_T ; \phi^{\uparrow L}} .
 \end{aligned}$$

The next law is the identity law for brun:

$$\begin{aligned}
 &\text{expect this to equal } \text{pu}_M : \quad \text{pu}_{L \circ M} ; \text{brun } \theta \\
 &\text{definitions of brun and } \text{pu}_{L \circ M} : = \text{pu}_M ; \underline{\text{pu}_L ; \theta} \\
 &\text{identity law for } \theta : = \text{pu}_M .
 \end{aligned}$$

The last law to be shown is the composition law for brun θ . Begin with its right-hand side since it is simpler,

$$\begin{aligned}
 &(\text{brun } \theta)^{\uparrow M \uparrow L} ; \text{brun } \theta ; \text{ftn}_M \\
 &\text{definition of brun} : = \theta^{\uparrow M \uparrow L} ; \theta ; \text{ftn}_M .
 \end{aligned}$$

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:

$$\begin{aligned}
 &\text{ftn}_{L \circ M} ; \text{brun } \theta \\
 &\text{definitions of } \text{ftn}_{L \circ M} \text{ and } \text{brun} : = \text{sw}_{L,M}^{\uparrow L} ; \text{ftn}_L ; \underline{\text{ftn}_M^{\uparrow L} ; \theta} \\
 &\text{naturality of } \theta : = \underline{\text{sw}_{L,M}^{\uparrow L} ; \text{ftn}_L ; \theta ; \text{ftn}_M} \\
 &\text{composition law for } \theta : = \underline{\text{sw}_{L,M}^{\uparrow L} ; \theta^{\uparrow L} ; \theta ; \text{ftn}_M} \\
 &\text{functor composition} : = \underline{(\text{sw}_{L,M} ; \theta)^{\uparrow L} ; \theta ; \text{ftn}_M} \\
 &\text{monadic naturality of } \text{sw}_{L,M} : = \underline{\theta^{\uparrow M \uparrow L} ; \theta ; \text{ftn}_M} .
 \end{aligned}$$

We have transformed both sides of the law into the same expression.

The lifting laws for mrun are

$$\text{mrun}(\text{id}) = \text{id} , \quad \text{mrun}(\phi) ; \text{mrun}(\chi) = \text{mrun}(\phi ; \chi) .$$

Since $\text{mrun}(\phi) = \phi^{\uparrow L}$ in our case, these laws hold because they are the same as the functor laws of L .

Finally, we verify the non-degeneracy law for brun :

$$\begin{aligned} \text{expect to equal id : } & \text{lift} ; \text{brun}(\theta) \\ \text{definitions of lift and brun : } & = \text{pu}_L ; \theta \\ \text{identity law for } \theta : & = \text{id} . \end{aligned}$$

If the base monad is M , the runners have the type signatures

$$\begin{aligned} \text{mrun } \phi^{L \rightsquigarrow N} : L^M \rightsquigarrow N^M , \quad \text{mrun } \phi = \phi ; \\ \text{brun } \theta^{M \rightsquigarrow L} : L^M \rightsquigarrow L^L , \quad \text{brun } \theta = \theta^{\uparrow L} . \end{aligned}$$

The laws of runners require that $\text{mrun } \phi$ and $\text{brun } \theta$ must be monadic morphisms, i.e. the identity and composition laws must hold for $\text{mrun } \phi$ and $\text{brun } \theta$:

$$\begin{aligned} \text{pu}_{L \circ M} ; \text{mrun } \phi &= \text{pu}_{N \circ M} , \\ \text{ftn}_{L \circ M} ; \text{mrun } \phi &= (\text{mrun } \phi)^{\uparrow M \uparrow L} ; \text{mrun } \phi ; \text{ftn}_{N \circ M} , \\ \text{pu}_{L \circ M} ; \text{brun } \theta &= \text{pu}_L , \\ \text{ftn}_{L \circ M} ; \text{brun } \theta &= (\text{brun } \theta)^{\uparrow M \uparrow L} ; \text{brun } \theta ; \text{ftn}_L . \end{aligned}$$

The monadic naturality laws for swap with respect to ϕ and χ are

$$\begin{array}{ccc} \text{sw}_{L,M} ; \phi = \phi^{\uparrow M} ; \text{sw}_{N,M} & , & \text{sw}_{L,M} ; \theta^{\uparrow L} = \theta . \\ \begin{array}{c} M^{L^A} \xrightarrow{\text{sw}_{L,M}} L^{M^A} \\ \downarrow \phi^{\uparrow M} \qquad \downarrow \phi \\ M^{N^A} \xrightarrow{\text{sw}_{N,M}} N^{M^A} \end{array} & & \begin{array}{c} M^{L^A} \xrightarrow{\text{sw}_{L,M}} L^{M^A} \\ \searrow \theta \qquad \downarrow \theta^{\uparrow L} \\ L^A \end{array} \end{array}$$

The first law to be proved is

$$\begin{aligned} \text{expect to equal } \text{pu}_{N \circ M} : & \text{pu}_{L \circ M} ; \text{mrun } \phi \\ \text{definitions of mrun and } \text{pu}_{L \circ M} : & = \text{pu}_M ; \text{pu}_L ; \phi \\ \text{identity law for } \phi : & = \text{pu}_M ; \text{pu}_N \\ \text{definition of } \text{pu}_{N \circ M} : & = \text{pu}_{N \circ M} . \end{aligned}$$

The next law is the composition law for $\text{mrun } \phi$:

$$\begin{aligned} \text{expect this to equal } \text{ftn}_T ; \phi : & (\text{mrun } \phi)^{\uparrow M \uparrow L} ; \text{mrun } \phi ; \text{ftn}_{N \circ M} \\ \text{definitions of mrun and } \text{ftn}_{N \circ M} : & = \phi^{\uparrow M \uparrow L} ; \phi ; \text{sw}_{N,M}^{\uparrow N} ; \text{ftn}_N ; \text{ftn}_M^{\uparrow N} \\ \text{naturality of } \phi : & = \phi^{\uparrow M \uparrow L} ; \text{sw}_{N,M}^{\uparrow L} ; \phi ; \text{ftn}_N ; \text{ftn}_M^{\uparrow N} \\ \text{monadic naturality of } \text{sw}_{N,M} \text{ raised to } L : & = \text{sw}_{L,M}^{\uparrow L} ; \phi^{\uparrow L} ; \phi ; \text{ftn}_N ; \text{ftn}_M^{\uparrow N} \\ \text{composition law for } \phi : & = \text{sw}_{L,M}^{\uparrow L} ; \text{ftn}_L ; \phi ; \text{ftn}_M^{\uparrow N} \\ \text{naturality of } \phi : & = \text{sw}_{L,M}^{\uparrow L} ; \text{ftn}_L ; \text{ftn}_M^{\uparrow L} ; \phi \\ \text{definition of } \text{ftn}_T : & = \text{ftn}_T ; \phi . \end{aligned}$$

The next law is the identity law for $\text{brun}(\theta)$:

$$\begin{aligned} \text{expect this to equal } \text{pu}_L : & \quad \text{pu}_{L \circ M} ; \text{brun} \theta \\ \text{definitions of } \text{brun} \text{ and } \text{pu}_{L \circ M} : & = \text{pu}_M ; \underline{\text{pu}_L ; \theta^{\uparrow L}} \\ \text{naturality of } \text{pu}_L : & = \underline{\text{pu}_M ; \theta} ; \text{pu}_L \\ \text{identity law for } \theta : & = \text{pu}_L . \end{aligned}$$

The last law is the composition law for $\text{brun}(\theta)$. Begin with its right-hand side,

$$\begin{aligned} (\text{brun} \theta)^{\uparrow M \uparrow L} ; \text{brun} \theta ; \text{ftn}_L \\ \text{definition of } \text{brun} : & = \theta^{\uparrow L \uparrow M \uparrow L} ; \theta^{\uparrow L} ; \text{ftn}_L \\ \text{functor composition} : & = (\theta^{\uparrow L \uparrow M} ; \theta)^{\uparrow L} ; \text{ftn}_L \\ \text{naturality of } \theta : & = (\theta ; \theta^{\uparrow L})^{\uparrow L} ; \text{ftn}_L . \end{aligned}$$

We now transform the left-hand side, hoping to obtain the same expression. We need to move ftn_L to the right and θ to the left:

$$\begin{aligned} \text{expect to equal } \theta^{\uparrow L} ; \theta^{\uparrow L \uparrow L} ; \text{ftn}_L : & \quad \text{ftn}_{L \circ M} ; \text{brun} \theta \\ \text{definitions of } \text{ftn}_{L \circ M} \text{ and } \text{brun} : & = \text{sw}_{L,M}^{\uparrow L} ; \text{ftn}_L ; \underline{\text{ftn}_M^{\uparrow L} ; \theta^{\uparrow L}} \\ \text{composition law for } \theta : & = \text{sw}_{L,M}^{\uparrow L} ; \underline{\text{ftn}_L ; (\theta ; \theta)^{\uparrow L}} \\ \text{naturality of } \text{ftn}_L : & = \underline{\text{sw}_{L,M}^{\uparrow L} ; (\theta ; \theta)^{\uparrow L \uparrow L}} ; \text{ftn}_L \\ \text{functor composition} : & = (\text{sw}_{L,M} ; \theta^{\uparrow L})^{\uparrow L} ; \theta^{\uparrow L \uparrow L} ; \text{ftn}_L \\ \text{composition law for } \theta : & = \underline{\theta^{\uparrow L} ; \theta^{\uparrow L \uparrow L}} ; \text{ftn}_L . \end{aligned}$$

The lifting laws for mrun are

$$\text{mrun}(\text{id}) = \text{id} , \quad \text{mrun}(\phi) ; \text{mrun}(\chi) = \text{mrun}(\phi ; \chi) .$$

Since $\text{mrun}(\phi) = \phi$ in our case, these laws are trivially satisfied.

Finally, the non-degeneracy law for brun :

$$\begin{aligned} \text{expect to equal } \text{id} : & \quad \text{lift} ; \text{brun}(\theta) \\ \text{definitions of } \text{lift} \text{ and } \text{brun} : & = \text{pu}_M^{\uparrow L} ; \theta^{\uparrow L} \\ \text{identity law for } \theta : & = \text{id}^{\uparrow L} = \text{id} . \end{aligned}$$

13.4.8 Summary of results

The following two theorems summarize the derivations in Section 13.4:

Theorem 13.4.8.1 (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 Theorem 13.4.2.1 and the monadic naturality laws

$$\text{sw}_{L,M} ; \phi^{\uparrow L} = \phi ; \text{sw}_{L,N} , \quad \text{sw}_{L,M} ; \theta = \theta^{\uparrow M} ,$$

with respect to arbitrary monadic 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 Theorem 13.4.2.1.

Theorem 13.4.8.2 (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 Theorem 13.4.2.1 and the monadic naturality laws

$$\text{sw}_{L,M} ; \phi = \phi^{\uparrow M} ; \text{sw}_{N,M} , \quad \text{sw}_{L,M} ; \theta^{\uparrow L} = \theta ,$$

with respect to arbitrary monadic 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 Theorem 13.4.2.1.

Theorems 13.4.8.1–13.4.8.2 enable us to check more easily whether a given base monad has a monad transformer of a “composed” kind. It is easier to check the 6 laws for the `swap` function than to verify the 15 monad transformer laws directly. Also, the laws of `swap` use simpler type constructors than the monad transformer laws.

13.5 Composed-inside transformers: Linear monads

A monad M is **linear** if it is of the form $M^A \triangleq P + Q \times A$, where P and Q are fixed types, and Q is a monoid. (The polynomial $P + Q \times A$ is linear in its type parameter A .) Well-known examples of linear monads are `Option`, `Either`, and `Writer`. The general case $M^A \triangleq P + Q \times A$ represents a computation that can fail and at the same time produce a log message. So, M can be seen as a composition of `Either` and `Writer`.

A different (but also linear) monad is obtained from the composition of `Writer` and `Either`. The type constructor of this monad is $Q \times (P + A)$.

In general, composition of two linear monads $M_1^A \triangleq P_1 + Q_1 \times A$ and $M_2^A \triangleq P_2 + Q_2 \times A$ is again linear because

$$\begin{aligned} & P_1 + Q_1 \times (P_2 + Q_2 \times A) \\ \text{expand brackets : } &= \underline{P_1 + Q_1 \times P_2} + \underline{Q_1 \times Q_2 \times A} \\ \text{define new } P, Q : &= P + Q \times A . \end{aligned}$$

Note that we need to define $Q \triangleq Q_1 \times Q_2$, and so Q is a monoid since, by assumption, Q_1 and Q_2 are monoids.

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 defined as

```
type OptionT[L[_], A] = L[Option[A]]
```

The `Option` type constructor must be nested *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 composed inside the foreign monad.

13.5.1 Definitions of `swap` and `flatten`

To show that the monad transformer for the base monad $M^A \triangleq P + Q \times A$ is $T_M^{L,A} = L^{M^A}$, we will implement a suitable `swap` function having the type signature

$$\text{sw}_{N,M} : M^{L^A} \Rightarrow L^{M^A} ,$$

for the base monad $M^A \triangleq P + Q \times A$ and an arbitrary foreign monad L . We will then prove that `swap` satisfies all the required laws stated in Theorem 13.4.8.2. This will guarantee that $T_M^{L,A} = L^{M^A}$ is a lawful monad transformer.

Expanding the definition of the type constructor M^A , we can write the type signature of the `swap` function as

$$\text{sw}_{L,M} : P + Q \times L^A \Rightarrow L^{P+Q \times A} .$$

We can map P to L^P by applying pu_L . We can also map $Q \times L^A \Rightarrow L^{Q \times A}$ since L is a functor,

$$q \times l \Rightarrow (a \Rightarrow q \times a)^{\uparrow L} l \quad .$$

It remains to unite these two functions. In the matrix notation, we write

$$\text{sw}_{L,M} = \left| \begin{array}{c|cc} P & (x:P \Rightarrow x + 0:Q \times A) \circ \text{pu}_L \\ Q \times L^A & q \times l \Rightarrow (a:A \Rightarrow 0:P + q \times a)^{\uparrow L} l \end{array} \right| \quad . \quad (13.18)$$

In Scala, the code is

```
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, a)) => Right((q, a))
}
```

Given this `swap` function, we define the `flatten` method for the transformed monad T (short notation ftn_T) by the standard formula

$$\text{ftn}_T = \text{sw}^{\uparrow L} \circ \text{ftn}_L \circ \text{ftn}_M^{\uparrow L} \quad .$$

The `pure` method for T (short notation pu_T) is $\text{pu}_T = \text{pu}_M \circ \text{pu}_L$. In Scala:

```
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.
```

Now we will 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 :

$$\begin{aligned} \text{fmap}_M f:A \Rightarrow B = f^{\uparrow M} &= \left| \begin{array}{c|cc} & P & Q \times B \\ \hline P & \text{id} & 0 \\ Q \times A & 0 & q \times a \Rightarrow q \times f(a) \end{array} \right| \quad , \\ \text{pu}_M a:A &= 0:P + q_0 \times a \quad , \quad \text{pu}_M = \left| \begin{array}{c|cc} & P & Q \times A \\ \hline A & 0 & a \Rightarrow q_0 \times a \end{array} \right| \quad , \\ \text{ftn}_M^{M^M A \Rightarrow M^A} &= \left| \begin{array}{c|cc} & P & Q \times A \\ \hline P & \text{id} & 0 \\ Q \times P & q \times p \Rightarrow p & 0 \\ Q \times Q \times A & 0 & q_1 \times q_2 \times a \Rightarrow (q_1 \oplus q_2) \times a \end{array} \right| \quad . \end{aligned}$$

13.5.2 Laws of swap

We do not need to verify naturality since `swap` is defined as a fully parametric function.

The inner-identity law We need to show that $\text{pu}_L^{\uparrow M} ; \text{sw} = \text{pu}_L$:

$$\begin{aligned}
 \text{pu}_L^{\uparrow M} ; \text{sw} &= \left\| \begin{array}{c} \text{id} \quad \emptyset \\ \emptyset \quad q \times a \Rightarrow q \times \text{pu}_L a \end{array} \right\|_{\stackrel{\circ}{;}} \left\| \begin{array}{c} (x:P \Rightarrow x + \emptyset:Q \times A) ; \text{pu}_L \\ q \times l \Rightarrow (x:A \Rightarrow \emptyset:P + q \times x)^{\uparrow L} l \end{array} \right\| \\
 \text{composition :} &= \left\| \begin{array}{c} (x:P \Rightarrow x + \emptyset:Q \times A) ; \text{pu}_L \\ q \times a \Rightarrow (a:A \Rightarrow \emptyset:P + q \times a)^{\uparrow L} (\text{pu}_L a) \end{array} \right\| \\
 \text{pu}_L \text{'s naturality :} &= \left\| \begin{array}{c} P \\ Q \times A \end{array} \right\| \left\| \begin{array}{c} x:P \Rightarrow \text{pu}_L(x + \emptyset:Q \times A) \\ q \times a \Rightarrow \text{pu}_L(\emptyset:P + q \times a) \end{array} \right\| \\
 \text{matrix notation :} &= \text{pu}_L .
 \end{aligned}$$

The outer-identity law We need to show that $\text{pu}_M ; \text{sw} = \text{pu}_M^{\uparrow L}$:

$$\begin{aligned}
 \text{pu}_M ; \text{sw} &= \left\| \begin{array}{c} \emptyset \quad l^{\uparrow L} \Rightarrow q_0 \times l \end{array} \right\|_{\stackrel{\circ}{;}} \left\| \begin{array}{c} (x:P \Rightarrow x + \emptyset:Q \times A) ; \text{pu}_L \\ q \times l \Rightarrow (x:A \Rightarrow \emptyset:P + q \times x)^{\uparrow L} l \end{array} \right\| \\
 \text{composition :} &= l^{\uparrow L} \Rightarrow (x:A \Rightarrow \emptyset:P + q_0 \times x)^{\uparrow L} l \\
 \text{definition of } \text{pu}_M : &= l \Rightarrow \text{pu}_M^{\uparrow L} l = \text{pu}_M .
 \end{aligned}$$

The inner-interchange law Show that $\text{ftn}_L^{\uparrow M} ; \text{sw} = \text{sw} ; \text{sw}^{\uparrow L} ; \text{ftn}_L$:

$$\begin{aligned}
 \text{ftn}_L^{\uparrow M} ; \text{sw} &= \left\| \begin{array}{c} \text{id} \quad \emptyset \\ \emptyset \quad q \times l \Rightarrow q \times \text{ftn}_L l \end{array} \right\|_{\stackrel{\circ}{;}} \left\| \begin{array}{c} (x:P \Rightarrow x + \emptyset:Q \times A) ; \text{pu}_L \\ q \times l \Rightarrow (a \Rightarrow \emptyset:P + q \times a)^{\uparrow L} l \end{array} \right\| \\
 &= \left\| \begin{array}{c} (x:P \Rightarrow x + \emptyset:Q \times A) ; \text{pu}_L \\ q \times l \Rightarrow (a \Rightarrow \emptyset:P + q \times a)^{\uparrow L} (\text{ftn}_L l) \end{array} \right\| , \tag{13.19} \\
 \text{sw} ; \text{sw}^{\uparrow L} ; \text{ftn}_L &= \left\| \begin{array}{c} (x:P \Rightarrow x + \emptyset:Q \times A) ; \text{pu}_L \\ q \times l \Rightarrow (a \Rightarrow \emptyset:P + q \times a)^{\uparrow L} l \end{array} \right\| ; \text{sw}^{\uparrow L} ; \text{ftn}_L \\
 &= \left\| \begin{array}{c} (x:P \Rightarrow x + \emptyset:Q \times A) ; \text{pu}_L ; \text{sw}^{\uparrow L} ; \text{ftn}_L \\ (q \times l \Rightarrow (a \Rightarrow \emptyset:P + q \times a)^{\uparrow L} l) ; \text{sw}^{\uparrow L} ; \text{ftn}_L \end{array} \right\| .
 \end{aligned}$$

It is quicker to simplify each expression in the last column separately and then to compare with the column in Eq. (13.19). Simplify the upper expression:

$$\begin{aligned}
 &(x:P \Rightarrow x + \emptyset:Q \times A) ; \text{pu}_L ; \text{sw}^{\uparrow L} ; \text{ftn}_L \\
 \text{naturality of } \text{pu}_L : &= (x:P \Rightarrow x + \emptyset:Q \times A) ; \text{sw} ; \text{pu}_L ; \text{ftn}_L \\
 \text{identity law of } L : &= (x:P \Rightarrow x + \emptyset:Q \times A) ; \text{sw} \\
 \text{definition of sw :} &= (x:P \Rightarrow x + \emptyset:Q \times A) ; \text{pu}_L .
 \end{aligned}$$

This equals the upper expression in Eq. (13.19). Simplify the lower expression;

$$\begin{aligned}
 &(q \times l \Rightarrow (a \Rightarrow \emptyset:P + q \times a)^{\uparrow L} l) ; \text{sw}^{\uparrow L} ; \text{ftn}_L \\
 \text{definition of } \triangleright : &= q \times l \Rightarrow l \triangleright (a \Rightarrow \emptyset:P + q \times a)^{\uparrow L} ; \text{sw}^{\uparrow L} ; \text{ftn}_L . \tag{13.20}
 \end{aligned}$$

Simplify the expression $(a \Rightarrow 0^P + q \times a)^{\uparrow L} ; sw^{\uparrow L}$ separately:

$$\begin{aligned}
 & (a \Rightarrow 0^P + q \times a) ; sw \\
 \text{composition : } & = a \Rightarrow \underline{sw(0^P + q \times a)} \\
 \text{definition of } sw : & = \underline{a \Rightarrow a \triangleright (x \Rightarrow 0^P + q \times x)^{\uparrow L}} \\
 \text{omit argument : } & = (x \Rightarrow 0^P + q \times x)^{\uparrow L} .
 \end{aligned} \tag{13.21}$$

Then we continue simplifying Eq. (13.20):

$$\begin{aligned}
 q \times l & \Rightarrow l \triangleright (a \Rightarrow 0^P + q \times a)^{\uparrow L} ; sw^{\uparrow L} ; ftn_L \\
 \text{use Eq. (13.21) : } & = q \times l \Rightarrow l \triangleright (x \Rightarrow 0^P + q \times x)^{\uparrow L} ; sw^{\uparrow L} ; ftn_L \\
 \text{naturality of } ftn_L : & = q \times l \Rightarrow l \triangleright ftn_L ; (x \Rightarrow 0^P + q \times x)^{\uparrow L} \\
 \text{definition of } \triangleright : & = q \times l \Rightarrow (x \Rightarrow 0^P + q \times x)^{\uparrow L}(ftn_L l) .
 \end{aligned}$$

This equals the lower expression in Eq. (13.19) after renaming x to a .

The outer-interchange law Show that $ftn_M ; sw = sw^{\uparrow M} ; sw ; ftn_M^{\uparrow L}$. The left-hand side is written using the matrices for ftn_M and sw :

$$\begin{aligned}
 & ftn_M ; sw \\
 & = \left\| \begin{array}{cc} id & 0 \\ q \times p \Rightarrow p & 0 \\ 0 & q_1 \times q_2 \times a \Rightarrow (q_1 \oplus q_2) \times a \end{array} \right\| ; \left\| \begin{array}{c} (x^P \Rightarrow x + 0) ; pu_L \\ q \times l \Rightarrow (x \Rightarrow 0 + q \times x)^{\uparrow L} l \end{array} \right\| \\
 & = \left\| \begin{array}{c} (x^P \Rightarrow x + 0) ; pu_L \\ (q \times p \Rightarrow p + 0) ; pu_L \\ q_1 \times q_2 \times a \Rightarrow (x \Rightarrow 0 + (q_1 \oplus q_2) \times x)^{\uparrow L} a \end{array} \right\| . \tag{13.22}
 \end{aligned}$$

We cannot simplify this any more, so we hope to transform the right-hand side, $sw^{\uparrow M} ; sw ; ftn_M^{\uparrow L}$, to the same column expression. Begin by writing the matrix for $sw^{\uparrow M}$, expanding the rows for the input type $M^{M^{LA}}$:

$$\begin{aligned}
 sw^{\uparrow M} & = \left\| \begin{array}{c} P \\ Q \times P \\ Q \times Q \times L^A \end{array} \right\| \left\| \begin{array}{cc} id & 0 \\ 0 & q \times p \Rightarrow q \times sw(p + 0) \\ 0 & q_1 \times q_2 \times l \Rightarrow q_1 \times sw(0 + q_2 \times l) \end{array} \right\| \\
 & = \left\| \begin{array}{c} P \\ Q \times P \\ Q \times Q \times L^A \end{array} \right\| \left\| \begin{array}{cc} id & 0 \\ 0 & q \times p \Rightarrow q \times pu_L(p + 0) \\ 0 & q_1 \times q_2 \times l \Rightarrow q_1 \times (x \Rightarrow 0 + q_2 \times x)^{\uparrow L} l \end{array} \right\| .
 \end{aligned}$$

Then compute the composition $\text{sw}^{\uparrow M} ; \text{sw}$ as

$$\begin{aligned}
& \text{sw}^{\uparrow M} ; \text{sw} \\
&= \left\| \begin{array}{c} \text{id} \\ 0 \\ 0 \end{array} \middle| \begin{array}{c} 0 \\ q \times p \Rightarrow q \times \text{pu}_L(p + 0) \\ q_1 \times q_2 \times l \Rightarrow q_1 \times (x \Rightarrow 0 + q_2 \times x)^{\uparrow L} l \end{array} \right\| \left\| \begin{array}{c} (x:P \Rightarrow x + 0) ; \text{pu}_L \\ q \times l \Rightarrow (x \Rightarrow 0 + q \times x)^{\uparrow L} l \end{array} \right\| \\
&= \left\| \begin{array}{c} (x:P \Rightarrow x + 0) ; \text{pu}_L \\ q \times p \Rightarrow (x^{M^A} \Rightarrow 0^P + q \times x)^{\uparrow L} (\text{pu}_L(p + 0)) \\ q_1 \times q_2 \times l \Rightarrow (x^{M^A} \Rightarrow 0^P + q_1 \times x)^{\uparrow L} (x \Rightarrow 0 + q_2 \times x)^{\uparrow L} l \end{array} \right\| \\
&= \left\| \begin{array}{c} (x:P \Rightarrow x + 0) ; \text{pu}_L \\ q \times p \Rightarrow \text{pu}_L(0^P + q \times (p + 0)) \\ q_1 \times q_2 \times l \Rightarrow (x^{M^A} \Rightarrow 0 + q_1 \times (0 + q_2 \times x))^{\uparrow L} l \end{array} \right\| .
\end{aligned}$$

Now we need to post-compose $\text{ftn}_M^{\uparrow L}$ with this column:

$$\begin{aligned}
\text{sw}^{\uparrow M} ; \text{sw} ; \text{ftn}_M^{\uparrow L} &= \left\| \begin{array}{c} (x:P \Rightarrow x + 0) ; \underline{\text{pu}_L ; \text{ftn}_M^{\uparrow L}} \\ (q \times p \Rightarrow 0^P + q \times (p + 0)) ; \underline{\text{pu}_L ; \text{ftn}_M^{\uparrow L}} \\ q_1 \times q_2 \times l \Rightarrow l \triangleright (x^{M^A} \Rightarrow 0 + q_1 \times (0 + q_2 \times x))^{\uparrow L} ; \underline{\text{ftn}_M^{\uparrow L}} \end{array} \right\| \\
\text{pu}_L \text{'s naturality : } &= \left\| \begin{array}{c} (x:P \Rightarrow x + 0) ; \text{ftn}_M ; \text{pu}_L \\ (q \times p \Rightarrow 0^P + q \times (p + 0)) ; \text{ftn}_M ; \text{pu}_L \\ q_1 \times q_2 \times l \Rightarrow l \triangleright (x^{M^A} \Rightarrow \underline{\text{ftn}_M(0 + q_1 \times (0 + q_2 \times x))}^{\uparrow L}) \end{array} \right\| \\
\text{compute ftn}_M(\dots) : &= \left\| \begin{array}{c} (x:P \Rightarrow \text{ftn}_M(x + 0)) ; \text{pu}_L \\ (q \times p \Rightarrow \text{ftn}_M(0^P + q \times (p + 0))) ; \text{pu}_L \\ q_1 \times q_2 \times l \Rightarrow l \triangleright (x^{M^A} \Rightarrow 0 + (q_1 \oplus q_2) \times x)^{\uparrow L} \end{array} \right\| \\
\text{compute ftn}_M(\dots) : &= \left\| \begin{array}{c} (x:P \Rightarrow x + 0) ; \text{pu}_L \\ (q \times p \Rightarrow p + 0) ; \text{pu}_L \\ q_1 \times q_2 \times l \Rightarrow (x^{M^A} \Rightarrow 0 + (q_1 \oplus q_2) \times x)^{\uparrow L} l \end{array} \right\| .
\end{aligned}$$

After renaming l to a , this is the same as the column in Eq. (13.22).

Monadic naturality laws Verify the laws of Theorem 13.4.8.2,

$$\text{SW}_{\text{Id},M} = \text{id} , \quad \text{SW}_{L,M} ; \phi = \phi^{\uparrow M} ; \text{SW}_{N,M} , \quad \text{SW}_{L,M} ; \theta^{\uparrow L} = \theta .$$

for arbitrary monadic morphisms $\phi : L \rightsquigarrow N$ and $\theta : M \rightsquigarrow \text{Id}$.

The first law is the swap identity law, $\text{sw}_{\text{Id},M} = \text{id}$:

$$\begin{aligned}
& \text{Eq. (13.18) with } L = \text{Id} : \quad \text{sw}_{\text{Id},M} \\
&= \left\| \begin{array}{c} P \\ Q \times \text{Id}^A \end{array} \middle| \begin{array}{c} (x:P \Rightarrow x + 0^Q \times A) ; \text{pu}_{\text{Id}} \\ q \times l \Rightarrow (a:A \Rightarrow 0^P + q \times a)^{\text{Id} l} \end{array} \right\| \\
\text{matrix notation : } &= \left\| \begin{array}{c} P \\ Q \times A \end{array} \middle| \begin{array}{c} x:P \Rightarrow x + 0^Q \times A \\ q \times a \Rightarrow 0^P + q \times a \end{array} \right\| \\
&= \text{id} .
\end{aligned}$$

Begin with the left-hand side of the second law,

$$\begin{aligned}
 & \text{sw}_{L,M} \circ \phi \\
 \text{definition of } \text{sw}_{L,M} : &= \left\| \begin{array}{c} (x:P \Rightarrow x + \mathbb{0}^{Q \times A}) \circ \text{pu}_L \\ q \times l \Rightarrow (a \Rightarrow \mathbb{0}^P + q \times a)^{\uparrow L} l \end{array} \right\| \circ \underline{\phi} \\
 \text{compose with } \phi : &= \left\| \begin{array}{c} (x:P \Rightarrow x + \mathbb{0}^{Q \times A}) \circ \text{pu}_L \circ \phi \\ q \times l \Rightarrow l \triangleright (a \Rightarrow \mathbb{0}^P + q \times a)^{\uparrow L} \circ \phi \end{array} \right\| \\
 \text{naturality of } \phi : &= \left\| \begin{array}{c} (x:P \Rightarrow x + \mathbb{0}^{Q \times A}) \circ \text{pu}_N \\ q \times l \Rightarrow l \triangleright \phi \circ (a \Rightarrow \mathbb{0}^P + q \times a)^{\uparrow N} \end{array} \right\| .
 \end{aligned}$$

The right-hand side is

$$\begin{aligned}
 & \phi^{\uparrow M} \circ \text{sw}_{N,M} \\
 = & \left\| \begin{array}{ccc} \text{id} & \mathbb{0} & (x:P \Rightarrow x + \mathbb{0}^{Q \times A}) \circ \text{pu}_N \\ \mathbb{0} & q \times l \Rightarrow q \times \phi(l) & q \times n \Rightarrow n \triangleright (a \Rightarrow \mathbb{0}^P + q \times a)^{\uparrow N} \end{array} \right\| \\
 \text{composition :} &= \left\| \begin{array}{c} (x:P \Rightarrow x + \mathbb{0}^{Q \times A}) \circ \text{pu}_N \\ q \times l \Rightarrow n \triangleright \phi \circ (a \Rightarrow \mathbb{0}^P + q \times a)^{\uparrow N} \end{array} \right\| .
 \end{aligned}$$

Both sides of the second law are now shown to be equal.

The left-hand side of the third law is

$$\begin{aligned}
 & \text{sw}_{L,M} \circ \theta^{\uparrow L} \\
 \text{compose with } \theta^{\uparrow L} : &= \left\| \begin{array}{c} (x:P \Rightarrow x + \mathbb{0}^{Q \times A}) \circ \text{pu}_L \circ \theta^{\uparrow L} \\ q \times l \Rightarrow l \triangleright (a \Rightarrow \mathbb{0}^P + q \times a)^{\uparrow L} \circ \theta^{\uparrow L} \end{array} \right\| \\
 \text{naturality of } \text{pu}_L : &= \left\| \begin{array}{c} (x:P \Rightarrow x + \mathbb{0}^{Q \times A}) \circ \theta \circ \text{pu}_L \\ q \times l \Rightarrow l \triangleright (a \Rightarrow \theta(\mathbb{0}^P + q \times a))^{\uparrow L} \end{array} \right\| .
 \end{aligned} \tag{13.23}$$

We expect this to equal the right-hand side, which we write as

$$\begin{aligned}
 & m^{M^{LA}} \Rightarrow \theta(m) \\
 \text{matrix notation :} &= \left\| \begin{array}{c} x:P \Rightarrow \theta(x + \mathbb{0}^{Q \times L^A}) \\ q \times l \Rightarrow \theta(\mathbb{0}^P + q \times l) \end{array} \right\| .
 \end{aligned} \tag{13.24}$$

Now consider each line in Eq. (13.23) separately. The upper line can be transformed as

$$\begin{aligned}
 & (x:P \Rightarrow x + \mathbb{0}^{Q \times A}) \circ \theta \circ \text{pu}_L \\
 \text{naturality of } \theta : &= (x:P \Rightarrow x + \mathbb{0}^{Q \times A}) \circ \text{pu}_L^{\uparrow M} \circ \theta \\
 \text{definition of } \uparrow M : &= x:P \Rightarrow \left\| \begin{array}{cc} x & \mathbb{0} \end{array} \right\| \triangleright \left\| \begin{array}{ccc} \text{id} & & \mathbb{0} \\ \mathbb{0} & q \times l \Rightarrow q \times \text{pu}_L l & \end{array} \right\| \circ \theta \\
 \text{matrix notation :} &= x:P \Rightarrow (x + \mathbb{0}^{Q \times L^A}) \triangleright \theta .
 \end{aligned}$$

This is now equal to the upper line of Eq. (13.24).

To proceed with the proof for the lower line of Eq. (13.23), we need to evaluate the monadic morphism $\theta : M^A \Rightarrow A$ 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 not related 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 mapping it to a within the naturality law. For any values q^Q, a^A , we define

$$\begin{aligned} m^{P+Q \times A} &\triangleq \mathbb{0}^P + q \times a \quad , \\ m_1^{P+Q \times \mathbb{1}} &\triangleq \mathbb{0}^P + q \times \mathbb{1} \quad . \end{aligned}$$

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 $f : \mathbb{1} \Rightarrow A$ as $f \triangleq (_ \Rightarrow a)$ using the fixed value a . Then we have $m = f^{\uparrow M} m_1$. Now we apply both sides of the naturality law $f^{\uparrow M} ; \theta = \theta ; f$ to the value m_1 :

$$m_1 \triangleright f^{\uparrow M} ; \theta = m_1 \triangleright \theta ; f \quad .$$

Simplify the left-hand side to

$$m_1 \triangleright f^{\uparrow M} ; \theta = \theta(f^{\uparrow M} m_1) = \theta(m) = \theta(\mathbb{0}^P + q \times a) \quad .$$

Simplify the right-hand side to

$$m_1 \triangleright \theta ; f = f(\theta(m_1)) = a \quad ,$$

since f always returns a . Therefore

$$\theta(\mathbb{0}^P + q \times a) = a \quad . \tag{13.25}$$

We can now compute the second line in Eq. (13.23) as

$$\begin{aligned} q \times l &\Rightarrow l \triangleright (a \Rightarrow \theta(\mathbb{0}^P + q \times a))^{\uparrow L} \\ \text{use Eq. (13.25)} : &= q \times l \Rightarrow l \triangleright (a \Rightarrow a)^{\uparrow L} \\ \text{identity law} : &= q \times l \Rightarrow l \quad . \end{aligned}$$

The second line in Eq. (13.24) is the same function, $q \times l \Rightarrow l$.

This concludes the proof of the swap laws for linear monads. It follows that linear monads have monad transformers of composed-inside kind.

13.5.3 Composition of transformers for linear monads

We have just shown that the “Either/Writer” monad $M^A \triangleq P + Q \times A$ has the `swap` function that satisfies the laws necessary for a composed-inside transformer. The other type of linear monad is the “Writer/Either” monad $W^A \triangleq Q \times (P + A)$. Do we need to show separately that the monad W has a lawful `swap` function? Actually, this follows from the stacking property of monad transformers (see Sections 13.3.6–13.3.7). 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 composed-inside.

We can show in general 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 of the composed-inside kind:

$$T_{M_1}^N = N \circ M_1 \quad , \quad T_{M_2}^N = N \circ M_2 \quad .$$

The functor composition of M_1 and M_2 can be seen as a monad stack,

$$M_1 \circ M_2 = T_{M_2}^{M_1} \quad .$$

What is the transformer for the monad $M_1 \circ M_2$? For any foreign monad N , we have the transformer stack

$$T_{M_2}^{T_{M_1}^N} = T_{M_2}^{N \circ M_1} = N \circ M_1 \circ M_2 \quad .$$

Since this is a transformer stack, it is a lawful monad transformer, as we have seen in Section 13.3.6. So, this is the monad transformer for $M_1 \circ M_2$, and it is of the composed-inside kind.

13.6 Composed-outside transformers: Rigid monads

Section 13.5 shows that the composed-inside monad transformers are available only for a limited subset of all monads, namely the monads of the form $M^A = P + Q \times A$ and $M^A = Q \times (P + A)$, called “linear”. It turns out that the composed-outside transformers are available for a significantly wider range of monads. Those monads are called “rigid” because one of their general properties is having a single “shape” (Theorem 13.6.5.8 in Section 13.6.5). (There does not seem to be an already accepted name for monads of this kind.)

Definition of rigid monads 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 does not explain what monads are rigid or how to recognize a non-rigid monad. These questions will be answered below.

Two simplest examples of rigid monads are the `Reader` monad and the `Search` monad,

$$\begin{aligned} \text{(the Reader monad)} : \quad & R^A \triangleq Z \Rightarrow A \quad , \\ \text{(the Search monad)} : \quad & S^A \triangleq (A \Rightarrow Z) \Rightarrow A \quad , \end{aligned}$$

where Z is a fixed type. These monads have composed-outside transformers:

$$\begin{aligned} \text{(the ReaderT transformer)} : \quad & T_R^{M,A} \triangleq Z \Rightarrow M^A \quad , \\ \text{(the SearchT transformer)} : \quad & T_S^{M,A} \triangleq (M^A \Rightarrow Z) \Rightarrow M^A \quad . \end{aligned}$$

To build intuition for rigid monads, we will look at some general constructions that create new rigid monads or combine existing rigid monads into new ones. In this section, we will prove that the following four constructions produce rigid monads:

1. Choice: $C^A \triangleq H^A \Rightarrow A$ is a rigid monad if H is any contrafunctor.
2. Composition: $P \circ R$ is a rigid monad if P and R are rigid monads.
3. Product: $P^A \times R^A$ is a rigid monad if P and R are rigid monads.
4. Selector: $S^A \triangleq F^{A \Rightarrow R^Q} \Rightarrow R^A$ is a rigid monad for any rigid monad R , any functor F , and any fixed type Q .

I do not know whether these four constructions are the only possible ways of creating new rigid monads. Below I will also mention other open questions I have about rigid monads.

13.6.1 Rigid monad construction 1: choice

The construction I call the **choice** monad, $R^A \triangleq H^A \Rightarrow A$, defines a rigid monad R for *any* given contrafunctor H .

This monad chooses a value of type A given a contrafunctor H that may *consume* values of type A (and presumably could check some conditions on those values). The contrafunctor H could be a constant contrafunctor $H^A \triangleq Q$, a function such as $H^A \triangleq A \Rightarrow Q$, or a more complicated contrafunctor.

Different choices of the contrafunctor H give specific examples of rigid monads, such as $R^A \triangleq \mathbb{1}$ (the unit monad), $R^A \triangleq A$ (the identity monad), $R^A \triangleq Z \Rightarrow A$ (the reader monad), as well as the **search² monad** $R^A \triangleq (A \Rightarrow Q) \Rightarrow A$.

The search monad represents the effect of searching for a value of type A that satisfies a condition expressed through a function of type $A \Rightarrow Q$. The simplest example of a search monad is found by setting $Q \triangleq \text{Bool}$. One may implement a function of type $(A \Rightarrow \text{Bool}) \Rightarrow A$ that *somewhat* finds a value of type A that might satisfy the given predicate of type $A \Rightarrow \text{Bool}$. The intention is to return a value that, if possible, 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-with-failure monad, $R^A \triangleq (A \Rightarrow \text{Bool}) \Rightarrow \mathbb{1} + A$. This (non-rigid) monad will return an empty value $\mathbb{1} + \emptyset^A$ if no value satisfying the predicate was found. There is a natural transformation from the search monad to the search-with-failure monad, implemented by checking whether the value returned by the search monad does actually satisfies the predicate.

Assume that H is a contrafunctor and M is a monad, and denote for brevity

$$T^A \triangleq R^{M^A} \triangleq H^{M^A} \Rightarrow M^A .$$

We will first give a self-contained proof that T is a monad. To verify the laws of monad transformers for T , we will derive the `swap` function and verify its laws.

Statement 13.6.1.1 $T^\bullet \triangleq R^{M^\bullet} \triangleq H^{M^\bullet} \Rightarrow M^\bullet$ is a monad if M is any monad and H is any contrafunctor. (If we set $M^A \triangleq A$, this also proves that R itself is a monad.)

Proof We need to define the monad instance for T and prove the identity and the associativity laws for T , assuming that the monad M satisfies these laws.

To define the monad instance for T , it is convenient to use the Kleisli formulation of the monad. In this formulation, we consider Kleisli morphisms of type $A \Rightarrow T^B$ and then define the Kleisli identity morphism, $\text{pu}_T : A \Rightarrow T^A$, and the Kleisli product operation \diamond_T ,

$$f:A \Rightarrow T^B \diamond_T g:B \Rightarrow T^C : A \Rightarrow T^C .$$

We are then required to define the operation \diamond_T and to prove identity and associativity laws for it.

We notice that since the type constructor R is itself a function type $H^A \Rightarrow A$, the type of the Kleisli morphism $A \Rightarrow T^B$ is actually $A \Rightarrow T^B \triangleq A \Rightarrow H^{M^B} \Rightarrow M^B$. While proving the monad laws for T , we will need to use the monad laws for M (since M is an arbitrary, unknown monad). In order to use the monad laws for M , it would be helpful if we had the Kleisli morphisms for M of type $A \Rightarrow M^B$ more easily available. If we flip the curried arguments of the Kleisli morphism type $A \Rightarrow H^{M^B} \Rightarrow M^B$ and instead consider the **flipped Kleisli** morphisms of type $H^{M^B} \Rightarrow A \Rightarrow M^B$, the type $A \Rightarrow M^B$ will be easier to reason about. Since the type $A \Rightarrow H^{M^B} \Rightarrow M^B$ is equivalent to $A \Rightarrow H^{M^B} \Rightarrow M^B$, any laws we prove for the flipped Kleisli morphisms will yield the corresponding laws for the standard Kleisli morphisms. The use of flipped Kleisli morphisms makes the proof significantly shorter.

We temporarily denote by $\tilde{\text{pu}}_T$ and $\tilde{\diamond}_T$ the flipped Kleisli operations:

$$\begin{aligned} \tilde{\text{pu}}_T &: H^{M^A} \Rightarrow A \Rightarrow M^A \\ f:H^{M^B} \Rightarrow A \Rightarrow M^B \tilde{\diamond}_T g:H^{M^C} \Rightarrow B \Rightarrow M^C &: H^{M^C} \Rightarrow A \Rightarrow M^C . \end{aligned}$$

To define the operations $\tilde{\text{pu}}_T$ and $\tilde{\diamond}_T$, we may use the methods pu_M and flm_M as well as the Kleisli product \diamond_M for the given monad M . The definitions are

$$\begin{aligned} \tilde{\text{pu}}_T &= - \Rightarrow \text{pu}_M \quad (\text{the argument is unused}) , \\ f \tilde{\diamond}_T g &= q \Rightarrow (f p) \diamond_M (g q) \quad \text{where} \\ p:H^{M^B} &= (\text{flm}_M (g q)) \downarrow^H q . \end{aligned}$$

²See <http://math.andrej.com/2008/11/21/>

This definition works by using the Kleisli product \diamond_M on values $f p : A \Rightarrow M^B$ and $g q : B \Rightarrow M^C$. To obtain a value $p : H^{M^B}$, we use the function $\text{flm}_M(g q) : M^B \Rightarrow M^C$ to H -contramap $q : H^{M^C}$ into $p : H^{M^B}$.

Written as a single expression, the definition of $\tilde{\diamond}_T$ is

$$f \tilde{\diamond}_T g = q \Rightarrow f \left((\text{flm}_M(g q))^{\downarrow H} q \right) \diamond_M (g q) . \quad (13.26)$$

Checking the left identity law:

$$\begin{aligned} & \tilde{\text{pu}}_T \tilde{\diamond}_T g \\ \text{definition of } \tilde{\diamond}_T : &= q \Rightarrow \underline{\tilde{\text{pu}}_T \left((\text{flm}_M(g q))^{\downarrow H} q \right)} \diamond_M (g q) \\ \text{definition of } \tilde{\text{pu}}_T : &= q \Rightarrow \underline{\text{pu}_M \diamond_M g q} \\ \text{left identity law for } M : &= q \Rightarrow g q \\ \text{function expansion :} &= g \end{aligned}$$

Checking the right identity law:

$$\begin{aligned} & f \tilde{\diamond}_T \tilde{\text{pu}}_T \\ \text{definition of } \tilde{\diamond}_T : &= q \Rightarrow f \left((\text{flm}_M(\tilde{\text{pu}}_T q))^{\downarrow H} q \right) \diamond_M \underline{(\tilde{\text{pu}}_T q)} \\ \text{definition of } \tilde{\text{pu}}_T : &= q \Rightarrow f \left((\text{flm}_M(\text{pu}_M))^{\downarrow H} q \right) \diamond_M \underline{\text{pu}_M} \\ \text{right identity law for } M : &= q \Rightarrow f \left((\text{id})^{\downarrow H} q \right) \\ \text{identity law for } H : &= q \Rightarrow f q \\ \text{function expansion :} &= f \end{aligned}$$

Checking the associativity law: $(f \tilde{\diamond}_T g) \tilde{\diamond}_T h$ must equal $f \tilde{\diamond}_T (g \tilde{\diamond}_T h)$. We have

$$\begin{aligned} & (f \tilde{\diamond}_T g) \tilde{\diamond}_T h \\ &= \left(s \Rightarrow f \left((\text{flm}_M(g s))^{\downarrow H} s \right) \diamond_M (g s) \right) \tilde{\diamond}_T h \\ &= q \Rightarrow f \left((\text{flm}_M(g r))^{\downarrow H} r \right) \diamond_M (g r) \diamond_M (h q) \quad \text{where} \\ & \quad r \triangleq (\text{flm}_M(h q))^{\downarrow H} q ; \end{aligned}$$

while

$$\begin{aligned} & f \tilde{\diamond}_T (g \tilde{\diamond}_T h) \\ &= f \tilde{\diamond}_T \left(q \Rightarrow g \left((\text{flm}_M(h q))^{\downarrow H} q \right) \diamond_M (h q) \right) \\ &= q \Rightarrow f \left((\text{flm}_M k)^{\downarrow H} q \right) \diamond_M u \quad \text{where} \\ & \quad r \triangleq (\text{flm}_M(h q))^{\downarrow H} q \quad \text{and} \\ & \quad u \triangleq (g r) \diamond_M (h q) . \end{aligned}$$

It remains to show that the following two expressions are equal,

$$\begin{aligned} & f \left((\text{flm}_M(g r))^{\downarrow H} r \right) \diamond_M (g r) \diamond_M (h q) \quad \text{and} \\ & f \left((\text{flm}_M((g r) \diamond_M (h q)))^{\downarrow H} q \right) \diamond_M (g r) \diamond_M (h q), \quad \text{where} \\ & \quad r \triangleq (\text{flm}_M(h q))^{\downarrow H} q . \end{aligned}$$

These two expressions differ only by the following sub-expressions,

$$(\text{flm}_M(g r))^{\downarrow H} r$$

and

$$(\text{flm}_M((g r) \diamond_M (h q)))^{\downarrow H} q ,$$

where $r \triangleq (\text{flm}_M(h q))^{\downarrow H} q$. Writing out the value r in the last argument of $(\text{flm}_M(g r))^{\downarrow H} r$ but leaving r unexpanded everywhere else, we now rewrite the differing sub-expressions as

$$\begin{aligned} & (\text{flm}_M(g r))^{\downarrow H} (\text{flm}_M(h q))^{\downarrow H} q \quad \text{and} \\ & (\text{flm}_M((g r) \diamond_M (h q)))^{\downarrow H} q . \end{aligned}$$

Now it becomes apparent that we need to put the two “ flm_M ”s closer together and to combine them by using the associativity law of the monad M . Then we can rewrite the first sub-expression and transform it into the second one:

$$\begin{aligned} & \frac{(\text{flm}_M(g r))^{\downarrow H} (\text{flm}_M(h q))^{\downarrow H} q}{\text{composition law for } H :} = (\text{flm}_M(g r) ; \text{flm}_M(h q))^{\downarrow H} q \\ & \text{associativity law for } M : = \underline{(\text{flm}_M((g r) ; \text{flm}_M(h q)))^{\downarrow H} q} \\ & \text{definition of } \diamond_M \text{ via } \text{flm}_M : = (\text{flm}_M((g r) \diamond_M (h q)))^{\downarrow H} q . \end{aligned}$$

This proves the associativity law for \circ_T .

Statement 13.6.1.2 The monad methods for T defined in Statement 13.6.1.1 can be written equivalently as

$$\begin{aligned} \text{pu}_T(a^A) &: H^{M^A} \Rightarrow M^A , \\ \text{pu}_T(a) &\triangleq \left(_ \Rightarrow \text{pu}_M a \right) ; \\ \text{flm}_T(f:A \Rightarrow H^{M^B} \Rightarrow M^B) &: (H^{M^A} \Rightarrow M^A) \Rightarrow H^{M^B} \Rightarrow M^B , \\ \text{flm}_T f &\triangleq t^{\cdot R^{M^A}} \Rightarrow q^{\cdot H^{M^B}} \Rightarrow (\text{flm}_M(x^A \Rightarrow f x q))^{\uparrow R} t q . \end{aligned} \tag{13.27}$$

Expressed through R ’s `flatMap` method, which is implemented as

$$\text{flm}_R g^{\cdot A \Rightarrow R^B} = t^{\cdot R^A} \Rightarrow q^{\cdot H^B} \Rightarrow (x^A \Rightarrow g x q)^{\uparrow R} t q , \tag{13.28}$$

the method flm_T can be written as

$$\text{flm}_T f = \text{flm}_R(y \Rightarrow q \Rightarrow \text{flm}_M(x \Rightarrow f x q) y) . \tag{13.29}$$

Proof The definition of \circ_T in Statement 13.6.1.1 used the flipped types of Kleisli morphisms, which is not the standard way of defining the methods of a monad. To restore the standard type signatures, we need to unflip the arguments:

$$\begin{aligned} & f^{\cdot A \Rightarrow H^{M^B} \Rightarrow M^B} \diamond_T g^{\cdot B \Rightarrow H^{M^C} \Rightarrow M^C} : A \Rightarrow H^{M^C} \Rightarrow M^C ; \\ & f \diamond_T g = t \Rightarrow q \Rightarrow \left(\tilde{f} \left((\text{flm}_M(b \Rightarrow g b q))^{\downarrow H} q \right) \diamond_M (b \Rightarrow g b q) \right) t , \end{aligned}$$

where $\tilde{f} \triangleq h \Rightarrow k \Rightarrow f k h$ is the flipped version of f . To replace \diamond_M by flm_M , express $x \diamond_M y = x ; \text{flm}_M y$ to find

$$f \diamond_T g = t \Rightarrow q \Rightarrow \left(\tilde{f} \left(p^{\downarrow H} q \right) ; p \right) t \quad \text{where } p = \text{flm}_M(x \Rightarrow g x q) .$$

To obtain an implementation of flm_T , express flm_T through \diamond_T as

$$\text{flm}_T g^{:A \Rightarrow T^B} = \text{id}^{:T^A \Rightarrow T^A} \diamond_T g \quad .$$

Now we need to substitute $f^{:T^A \Rightarrow T^A} = \text{id}$ into $f \diamond_T g$. Noting that \tilde{f} will then become

$$\tilde{f} = (h \Rightarrow k \Rightarrow \text{id} \, k \, h) = (h \Rightarrow k \Rightarrow k \, h) \quad ,$$

we get

$$\begin{aligned} \text{flm}_T g^{:A \Rightarrow T^B} &= \text{id} ; \text{flm}_T g \\ \text{definition of } \diamond_T : \quad &= t^{:T^A} \Rightarrow q^{H^M{}^B} \Rightarrow (\tilde{f}(p^{\downarrow H} q) ; p) t \\ &\quad \text{where } p \triangleq \text{flm}_M(x \Rightarrow g \, x \, q) \\ \text{substitute } f = \text{id} : \quad &= t \Rightarrow q \Rightarrow ((h \Rightarrow k \Rightarrow k \, h)(p^{\downarrow H} q) ; p) t \\ \text{apply } k \text{ to } p^{\downarrow H} q : \quad &= t \Rightarrow q \Rightarrow ((k \Rightarrow k(p^{\downarrow H} q)) ; p) t \\ \text{definition of } ; : \quad &= t \Rightarrow q \Rightarrow p(t(p^{\downarrow H} q)) \quad . \end{aligned}$$

By definition of the functor $R^A \triangleq H^A \Rightarrow A$, we raise any function $p^{:A \Rightarrow B}$ into R as

$$\begin{aligned} p^{\uparrow R} : (H^A \Rightarrow A) &\Rightarrow H^B \Rightarrow B \quad , \\ p^{\uparrow R} r^{H^A \Rightarrow A} \triangleq p^{\downarrow H} ; r ; p \\ &= q^{H^B} \Rightarrow p(r(p^{\downarrow H} q)) \quad . \end{aligned}$$

Finally, renaming g to f , we obtain the desired code,

$$\text{flm}_T f = t \Rightarrow q \Rightarrow p^{\uparrow R} t \, q \quad \text{where } p \triangleq \text{flm}_M(x \Rightarrow f \, x \, q) \quad .$$

To express flm_T via flm_R , we just need to choose the value of g such that Eq. (13.28) becomes equal to Eq. (13.27). Comparing these two expressions, we find that we need

$$\text{flm}_M(x \Rightarrow f \, x \, q) = (y \Rightarrow g \, y \, q) \quad .$$

This is achieved if we define $g \, y \, q = \text{flm}_M(x^{:A} \Rightarrow f \, x \, q) \, y$, or equivalently

$$g = y \Rightarrow q \Rightarrow \text{flm}_M(x \Rightarrow f \, x \, q) \, y \quad .$$

This gives the desired Eq. (13.29).

Statement 13.6.1.3 The monad T has the methods `flatten` and `swap` defined by

$$\text{ftn}_T = t^{:T^A} \Rightarrow q^{:H^M{}^A} \Rightarrow q \triangleright \left(t \triangleright (\text{flm}_M(x^{:R^M{}^A} \Rightarrow x \, q))^{\uparrow R} \right) \quad , \quad (13.30)$$

$$\text{sw}_{R,M} = m^{:M^{R^A}} \Rightarrow q^{:H^M{}^A} \Rightarrow (r^{:R^A} \Rightarrow r(\text{pu}_M^{\downarrow H} q))^{\uparrow M} m \quad . \quad (13.31)$$

These functions are computationally equivalent (can be derived from each other). In the \triangleright -notation, the formula for $\text{sw}_{R,M}$ is

$$q \triangleright (m \triangleright \text{sw}_{R,M}) = m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M} \quad . \quad (13.32)$$

Proof Using Eq. (13.27) and the relationship $\text{ftn}_T = \text{flm}_T \text{id}^{\cdot T^A \Rightarrow T^A}$, we find

$$\begin{aligned} \text{ftn}_T t^{\cdot T^A} &= \underline{\text{flm}_T(\text{id})} t \\ \text{use Eq. (13.27)} : \quad &= q \Rightarrow (\text{flm}_M(x^{\cdot A} \Rightarrow f x q))^{\uparrow R} \underline{t} q \\ \text{definition of } \triangleright : \quad &= q \Rightarrow \underline{q \triangleright (t \triangleright (\text{flm}_M(x^{\cdot A} \Rightarrow f x q))^{\uparrow R})} . \end{aligned}$$

Using Eq. (13.29) instead of Eq. (13.27), we get

$$\begin{aligned} \text{ftn}_T &= \text{flm}_T(\text{id}) \\ &= \text{flm}_R(y \Rightarrow q \Rightarrow \text{flm}_M(x \Rightarrow x q) y) . \end{aligned}$$

Deriving the formulas for $\text{sw}_{R,M}$ We start with ftn_T as just obtained and substitute into Eq. (13.16):

$$\begin{aligned} \text{sw}_{R,M}(m) &= m \triangleright \text{pu}_M^{\uparrow R \uparrow M} ; \text{pu}_R ; \underline{\text{ftn}_T} \\ \text{use Eq. (13.29)} : \quad &= m \triangleright \text{pu}_M^{\uparrow R \uparrow M} ; \underline{\text{pu}_R} ; \text{flm}_R(y \Rightarrow q \Rightarrow \text{flm}_M(x \Rightarrow x q) y) \\ \text{left identity law of } R : \quad &= m \triangleright \text{pu}_M^{\uparrow R \uparrow M} ; \underline{(y \Rightarrow q \Rightarrow \text{flm}_M(x \Rightarrow x q)) y} \\ \triangleright \text{ notation} : \quad &= m \triangleright \text{pu}_M^{\uparrow R \uparrow M} \triangleright (y \Rightarrow q \Rightarrow \underline{y \triangleright \text{flm}_M(x \Rightarrow x q)}) \end{aligned} \tag{13.33}$$

$$\begin{aligned} \text{apply to argument } y : \quad &= q \Rightarrow m \triangleright \text{pu}_M^{\uparrow R \uparrow M} \triangleright \underline{\text{flm}_M(x \Rightarrow x q)} \\ \text{express } \text{flm}_M \text{ via } \text{ftn}_M : \quad &= q \Rightarrow m \triangleright \text{pu}_M^{\uparrow R \uparrow M} ; \underline{(x \Rightarrow x q)^{\uparrow M}} ; \underline{\text{ftn}_M} \\ \text{composition law of } M : \quad &= q \Rightarrow m \triangleright (\text{pu}_M^{\uparrow R} ; (x \Rightarrow x q))^{\uparrow M} ; \underline{\text{ftn}_M} . \end{aligned} \tag{13.34}$$

It appears that simplifying this expression requires to rewrite the function $\text{pu}_M^{\uparrow R} ; (x \Rightarrow x q)$. To proceed further, we need to use the definition of raising a function $f^{\cdot A \Rightarrow B}$ to the functor R ,

$$f^{\uparrow R} \triangleq r^{\cdot R^A} \Rightarrow f^{\downarrow H} ; r ; f ,$$

so we can write

$$\begin{aligned} &\text{pu}_M^{\uparrow R} ; (x \Rightarrow x q) \\ \text{function composition} : \quad &= r \Rightarrow \underline{\text{pu}_M^{\uparrow R} r q} \\ \text{definition of } \uparrow R : \quad &= r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r ; \underline{\text{pu}_M} \\ \text{forward composition} : \quad &= (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r) ; \underline{\text{pu}_M} . \end{aligned} \tag{13.35}$$

In deriving Eq. (13.35), we used the general property of the forward composition,

$$x \Rightarrow y \triangleright f(x, y) ; g = (x \Rightarrow y \triangleright f(x, y)) ; g ,$$

where g must not depend on x or y . We can now rewrite Eq. (13.34) as

$$\begin{aligned} &q \Rightarrow m \triangleright (\text{pu}_M^{\uparrow R} ; (x \Rightarrow x q))^{\uparrow M} ; \underline{\text{ftn}_M} \\ \text{use Eq. (13.35)} : \quad &= q \Rightarrow m \triangleright ((r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r) ; \underline{\text{pu}_M})^{\uparrow M} ; \underline{\text{ftn}_M} \\ \text{functor composition for } M : \quad &= q \Rightarrow m \triangleright ((r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M} ; \underline{\text{pu}_M^{\uparrow M}} ; \underline{\text{ftn}_M} \\ \text{identity law of } M : \quad &= q \Rightarrow \underline{m \triangleright ((r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M}} \\ \triangleright \text{ notation} : \quad &= q \Rightarrow (r \Rightarrow r(\text{pu}_M^{\downarrow H} q))^{\uparrow M} m . \end{aligned}$$

The last expression coincides with Eq. (13.31).

The formula (13.32) follows by applying Eq. (13.31) to the arguments m and q . To make the computation clearer, we rename the bound variables m and q inside Eq. (13.31) to m_1 and q_1 :

$$\begin{aligned} q \triangleright (m \triangleright (m_1 \Rightarrow q_1 \Rightarrow (r \Rightarrow r(\text{pu}_M^{\downarrow H} q_1))^{\uparrow M} m_1)) \\ \text{apply to argument } m : = q \triangleright (q_1 \Rightarrow m \triangleright (r \Rightarrow r(\text{pu}_M^{\downarrow H} q_1))^{\uparrow M}) \\ \text{apply to argument } q : = m \triangleright (r \Rightarrow \underline{r(\text{pu}_M^{\downarrow H} q)})^{\uparrow M} \\ \triangleright \text{ notation} : = m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M} . \end{aligned}$$

Deriving ftn_T from $\text{sw}_{R,M}$ Given the swap function defined by Eq. (13.31), we can recover the original ftn_T function from Eq. (13.30) via the standard formula (13.6), $\text{ftn}_T = \text{sw}^{\uparrow R} ; \text{ftn}_R ; \text{ftn}_M^{\uparrow R}$:

$$\begin{aligned} & \text{sw}^{\uparrow R} ; \underline{\text{ftn}_R ; \text{ftn}_M^{\uparrow R}} \\ \text{naturality of } \text{ftn}_R : & = \underline{\text{sw}^{\uparrow R} ; \text{ftn}_M^{\uparrow R} ; \text{ftn}_R} \\ \text{composition under } R : & = (\text{sw} ; \underline{\text{ftn}_M^{\uparrow R}})^{\uparrow R} ; \underline{\text{ftn}_R} \\ \text{relating } \text{flm}_R \text{ and } \text{ftn}_R : & = \underline{\text{flm}_R} (\text{sw} ; \underline{\text{ftn}_M^{\uparrow R}}) \\ \text{use Eq. (13.28)} : & = t \Rightarrow q \Rightarrow (x^A \Rightarrow (\text{sw} ; \text{ftn}_M^{\uparrow R}) x q)^{\uparrow R} t q . \end{aligned} \quad (13.36)$$

To proceed, we need to transform $\text{sw} ; \text{ftn}_M^{\uparrow R}$ in some way:

$$\begin{aligned} & \text{sw} ; \underline{\text{ftn}_M^{\uparrow R}} \\ \text{definitions} : & = (m \Rightarrow q \Rightarrow m \triangleright ((r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M}) ; \underline{(r \Rightarrow \text{ftn}_M^{\downarrow H} ; r ; \text{ftn}_M)} \\ \text{composition} : & = m \Rightarrow \underline{\text{ftn}_M^{\downarrow H}} ; (q \Rightarrow m \triangleright ((r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M}) ; \underline{\text{ftn}_M} \\ \text{expansion} : & = m \Rightarrow (q \Rightarrow \underline{q \triangleright \text{ftn}_M^{\downarrow H}}) ; (q \Rightarrow m \triangleright ((r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M}) ; \underline{\text{ftn}_M} \\ \text{composition} : & = m \Rightarrow (q \Rightarrow m \triangleright ((r \Rightarrow q \triangleright \text{ftn}_M^{\downarrow H} ; \text{pu}_M^{\downarrow H} ; r))^{\uparrow M}) ; \underline{\text{ftn}_M} . \end{aligned} \quad (13.37)$$

We can transform the sub-expression $(r \Rightarrow q \triangleright \text{ftn}_M^{\downarrow H} ; \text{pu}_M^{\downarrow H} ; r)$ to

$$\begin{aligned} \triangleright \text{ notation} : & r \Rightarrow q \triangleright \underline{\text{ftn}_M^{\downarrow H} ; \text{pu}_M^{\downarrow H}} ; r \\ \text{composition law of } H : & = r \Rightarrow q \triangleright (\text{pu}_M ; \underline{\text{ftn}_M})^{\downarrow H} ; r \\ \text{left identity law of } M : & = r \Rightarrow \underline{q \triangleright r} \\ \triangleright \text{ notation} : & = r \Rightarrow r(q) . \end{aligned} \quad (13.38)$$

Using this simplification, we continue transforming Eq. (13.37) as

$$\begin{aligned} m \Rightarrow (q \Rightarrow m \triangleright ((r \Rightarrow q \triangleright \text{ftn}_M^{\downarrow H} ; \text{pu}_M^{\downarrow H} ; r))^{\uparrow M}) ; \underline{\text{ftn}_M} \\ \text{use Eq. (13.38)} : & = m \Rightarrow (q \Rightarrow m \triangleright (r \Rightarrow r(q))^{\uparrow M}) ; \underline{\text{ftn}_M} \\ \text{composition} : & = m \Rightarrow q \Rightarrow m \triangleright (r \Rightarrow r(q))^{\uparrow M} ; \underline{\text{ftn}_M} \\ \text{relating } \text{flm}_M \text{ and } \text{ftn}_M : & = m \Rightarrow q \Rightarrow \text{flm}_M (r \Rightarrow r(q)) m . \end{aligned}$$

Substituting this instead of $\text{sw} ; \text{ftn}_M^{\uparrow R}$ into Eq. (13.36), we get

$$\begin{aligned} t \Rightarrow q \Rightarrow (x \Rightarrow (\text{sw} ; \underline{\text{ftn}_M^{\uparrow R}}) x q)^{\uparrow R} t q \\ = t \Rightarrow q \Rightarrow (x \Rightarrow \text{flm}_M (r \Rightarrow r(q)) x)^{\uparrow R} t q . \end{aligned}$$

The last expression is the same as Eq. (13.30).

Statement 13.6.1.4 Without assuming the monad laws for the function ftn_T , the laws in Theorem 13.4.2.1 hold for the swap function defined by Eq. (13.31).

Proof After replacing the base monad L by R , the required laws are

$$\begin{aligned} \text{pu}_R^{\uparrow M} ; \text{sw} &= \text{pu}_R , & \text{pu}_M ; \text{sw} &= \text{pu}_M^{\uparrow R} , \\ \text{ftn}_R^{\uparrow M} ; \text{sw} &= \text{sw} ; \text{sw}^{\uparrow R} ; \text{ftn}_R , & \text{ftn}_M ; \text{sw} &= \text{sw}^{\uparrow M} ; \text{sw} ; \text{ftn}_M^{\uparrow R} . \end{aligned}$$

Proof of the inner-identity law Compute

$$\begin{aligned} &\text{pu}_R^{\uparrow M} ; \text{sw} \\ \text{use Eq. (13.31)} : &= (m \Rightarrow \underline{m \triangleright \text{pu}_R^{\uparrow M}}) ; (m \Rightarrow q \Rightarrow \underline{m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M}}) \\ \text{function composition} : &= m \Rightarrow q \Rightarrow m \triangleright \underline{\text{pu}_R^{\uparrow M} ; (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M}} \\ \text{functor law of } M : &= m \Rightarrow q \Rightarrow m \triangleright (\text{pu}_R ; (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M} . \end{aligned} \quad (13.39)$$

To proceed, we simplify the expression $\text{pu}_R ; (r \Rightarrow \dots)$:

$$\begin{aligned} &\text{pu}_R ; (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r) \\ \text{argument expansion} : &= (m \Rightarrow m \triangleright \text{pu}_R) ; (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r) \\ \text{function composition} : &= m \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; (m \triangleright \text{pu}_R) . \end{aligned} \quad (13.40)$$

We now have to use the definition of pu_R , which is $\text{pu}_R = x \Rightarrow y \Rightarrow x$, or in the forwarding notation,

$$y \triangleright (x \triangleright \text{pu}_R) = x . \quad (13.41)$$

With this simplification at hand, we continue from Eq. (13.40) to

$$\begin{aligned} &m \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; (m \triangleright \text{pu}_R) \\ \text{use Eq. (13.41)} : &= m \Rightarrow m = \text{id} . \end{aligned}$$

Therefore, Eq. (13.39) becomes

$$\begin{aligned} &m \Rightarrow q \Rightarrow m \triangleright (\text{pu}_R ; (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M} \\ &= (m \Rightarrow q \Rightarrow \underline{m \triangleright \text{id}}) \\ &= (m \Rightarrow q \Rightarrow m) = \text{pu}_R . \end{aligned}$$

This proves the inner-identity law.

Proof of the outer-identity law The left-hand side of this law is

$$\begin{aligned} &\text{pu}_M ; \text{sw} \\ \text{Eq. (13.31)} : &= (m \Rightarrow \underline{m \triangleright \text{pu}_M}) ; (m \Rightarrow q \Rightarrow \underline{m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M}}) \\ \text{function composition} : &= m \Rightarrow q \Rightarrow m \triangleright \underline{\text{pu}_M ; (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M}} \\ \text{naturality of } \text{pu}_M : &= m \Rightarrow q \Rightarrow m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r) ; \text{pu}_M \\ \triangleright \text{ notation} : &= m \Rightarrow q \Rightarrow \underline{m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r ; \text{pu}_M)} \\ \text{apply function to } m : &= m \Rightarrow q \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; m ; \text{pu}_M \\ \text{argument expansion} : &= m \Rightarrow \text{pu}_M^{\downarrow H} ; m ; \text{pu}_M \\ \text{definition of } \uparrow R : &= \text{pu}_M^{\uparrow R} . \end{aligned}$$

This is equal to the right-hand side of the law.

Proof of the inner-interchange law The law is written as

$$\text{ftn}_R^{\uparrow M} ; \text{sw} = \text{sw} ; \text{sw}^{\uparrow R} ; \text{ftn}_R . \quad (13.42)$$

We will apply both sides of the law to arbitrary $m : M^{R^A}$ and $q : H^M$, and transform both sides to the same expression.

Below, we will need a simplified formula for ftn_R derived from Eq. (13.28):

$$\begin{aligned} \text{ftn}_R &= \text{flm}_R(\text{id}) \\ \text{use Eq. (13.28)} : &= t \Rightarrow q \Rightarrow (x \Rightarrow x) q^{\uparrow R} t q \\ \text{definition of } \uparrow^R : &= t \Rightarrow q \Rightarrow (\underline{r \Rightarrow (x \Rightarrow x) q}^{\downarrow H} ; r ; (x \Rightarrow x)) t q \\ \text{apply to argument} : &= t \Rightarrow q \Rightarrow ((x \Rightarrow q \triangleright x)^{\downarrow H} ; t ; (x \Rightarrow x) q) q \\ \text{use } \triangleright \text{ notation} : &= t \Rightarrow q \Rightarrow q \triangleright (q \triangleright (x \Rightarrow q \triangleright x)^{\downarrow H} ; t) . \end{aligned} \quad (13.43)$$

We first apply the left-hand side of the law (13.42) to m and q :

$$\begin{aligned} &q \triangleright (m \triangleright \text{ftn}_R^{\uparrow M} ; \text{sw}) \\ &\triangleright \text{notation} : = q \triangleright (m \triangleright \text{ftn}_R^{\uparrow M} \triangleright \text{sw}) \\ \text{use Eq. (13.32)} : &= m \triangleright \underline{\text{ftn}_R^{\uparrow M} ; (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M}} \\ \text{composition law for } M : &= m \triangleright (\text{ftn}_R ; (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M} . \end{aligned}$$

We now need to simplify the sub-expression under $(...)^{\uparrow M}$:

$$\begin{aligned} &\text{ftn}_R ; (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} \triangleright r) \\ \text{function composition} : &= r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} \triangleright \underline{\text{ftn}_R(r)} \\ \text{use Eq. (13.43)} : &= r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} \triangleright (q \triangleright \text{pu}_M^{\downarrow H} \triangleright (x \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} \triangleright x)^{\downarrow H} ; r) \\ \text{composition law for } H : &= r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; (q \triangleright (x \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; x ; \text{pu}_M))^{\downarrow H} ; r \\ \text{definition of } \uparrow^R : &= r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; (q \triangleright (x \Rightarrow q \triangleright \text{pu}_M^{\uparrow R}(x)))^{\downarrow H} ; r . \end{aligned}$$

The left-hand side of the law (13.42) then becomes

$$m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; (q \triangleright (x \Rightarrow q \triangleright \text{pu}_M^{\uparrow R}(x)))^{\downarrow H} ; r))^{\uparrow M} .$$

Now apply the right-hand side of the law (13.42) to m and q :

$$\begin{aligned} &q \triangleright (m \triangleright \text{sw} ; \text{sw}^{\uparrow R} ; \text{ftn}_R) \\ \text{definition of } \uparrow^R : &= q \triangleright (\underline{m \triangleright \text{sw} \triangleright (x \Rightarrow \text{sw}^{\downarrow H} ; x ; \text{sw})} \triangleright \text{ftn}_R) \\ \text{apply to arguments} : &= q \triangleright (\underline{\text{ftn}_R(\text{sw}^{\downarrow H} ; \text{sw}(m))} ; \text{sw}) \\ \text{use Eq. (13.43)} : &= q \triangleright (q \triangleright (x \Rightarrow q \triangleright x)^{\downarrow H} ; \text{sw}^{\downarrow H} ; \text{sw}(m) ; \text{sw}) \\ \text{composition law of } H : &= q \triangleright (q \triangleright (\text{sw} ; (x \Rightarrow q \triangleright x))^{\downarrow H} ; \text{sw}(m) ; \text{sw}) . \end{aligned} \quad (13.44)$$

To proceed, we simplify the sub-expression $\text{sw}(m) ; \text{sw}$ separately by computing the function compositions:

$$\begin{aligned} &\text{sw}(m) ; \text{sw} \\ &= (q_1 \Rightarrow m \triangleright (r \Rightarrow q_1 \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M}) ; (y \Rightarrow q_2 \Rightarrow y \triangleright (r \Rightarrow q_2 \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M}) \\ &= q_1 \Rightarrow q_2 \Rightarrow (m \triangleright (r \Rightarrow q_1 \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M}) \triangleright (r \Rightarrow q_2 \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M} \\ &= q_1 \Rightarrow q_2 \Rightarrow m \triangleright ((r \Rightarrow q_1 \triangleright \text{pu}_M^{\downarrow H} ; r) ; (r \Rightarrow q_2 \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M} . \end{aligned}$$

Using this formula, we can write, for any z of a suitable type,

$$\begin{aligned} q \triangleright (z \triangleright \text{sw}(m) ; \text{sw}) &= m \triangleright ((r \Rightarrow z \triangleright \text{pu}_M^{\downarrow H} ; r) ; (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M} \\ \text{function composition : } &= m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; (z \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M} . \end{aligned} \quad (13.45)$$

Now we can substitute this into Eq. (13.44):

$$\begin{aligned} q \triangleright (q \triangleright (\text{sw} ; (x \Rightarrow q \triangleright x))^{\downarrow H} ; \text{sw}(m) ; \text{sw}) \\ \text{use Eq. (13.45) : } &= m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; (q \triangleright (\text{sw} ; (x \Rightarrow q \triangleright x))^{\downarrow H} ; \text{pu}_M^{\downarrow H} ; r))^{\uparrow M} \\ H's \text{ composition : } &= m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; (q \triangleright (\text{pu}_M ; \text{sw} ; (x \Rightarrow q \triangleright x))^{\downarrow H} ; r))^{\uparrow M} \\ \text{outer-identity : } &= m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; (q \triangleright (\text{pu}_M^{\uparrow R} ; (x \Rightarrow q \triangleright x))^{\downarrow H} ; r))^{\uparrow M} \\ \text{composition : } &= m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; (q \triangleright (x \Rightarrow q \triangleright \text{pu}_M^{\uparrow R} ; x))^{\downarrow H} ; r))^{\uparrow M} . \end{aligned}$$

We arrived at the same expression as the left-hand side of the law.

Proof of the outer-interchange law The law is written as

$$\text{ftn}_M ; \text{sw} = \text{sw}^{\uparrow M} ; \text{sw} ; \text{ftn}_M^{\uparrow R} . \quad (13.46)$$

We will apply both sides of the law to arbitrary $m^{\cdot M^{\cdot M^R}}$ and $q^{\cdot H^M}$, and transform both sides to the same expression. We begin with the more complicated right-hand side:

$$\begin{aligned} q \triangleright (m \triangleright \text{sw}^{\uparrow M} ; \text{sw} ; \text{ftn}_M^{\uparrow R}) \\ \triangleright \text{ notation : } &= q \triangleright ((m \triangleright \text{sw}^{\uparrow M} \triangleright \text{sw}) \triangleright \text{ftn}_M^{\uparrow R}) \\ \text{definition of } \uparrow R : &= q \triangleright (\text{ftn}_M^{\downarrow H} ; (m \triangleright \text{sw}^{\uparrow M} \triangleright \text{sw}) ; \text{ftn}_M) \\ \triangleright \text{ notation : } &= (q \triangleright \text{ftn}_M^{\downarrow H} ; (m \triangleright \text{sw}^{\uparrow M} \triangleright \text{sw})) \triangleright \text{ftn}_M \\ \text{use Eq. (13.32) : } &= (m \triangleright \text{sw}^{\uparrow M} \triangleright (r \Rightarrow q \triangleright \text{ftn}_M^{\downarrow H} ; \text{pu}_M^{\downarrow H} ; r))^{\uparrow M} \triangleright \text{ftn}_M \\ \text{composition for } H \text{ and } M : &= m \triangleright (\text{sw} ; (r \Rightarrow q \triangleright (\text{pu}_M ; \text{ftn}_M)^{\downarrow H} ; r))^{\uparrow M} ; \text{ftn}_M \\ \text{left identity law of } M : &= m \triangleright (\text{sw} ; (r \Rightarrow q \triangleright r))^{\uparrow M} ; \text{ftn}_M . \end{aligned} \quad (13.47)$$

Let us simplify the sub-expression $\text{sw} ; (r \Rightarrow q \triangleright r)$ separately:

$$\begin{aligned} \text{sw} ; (r \Rightarrow q \triangleright r) &= (x \Rightarrow x \triangleright \text{sw}) ; (r \Rightarrow q \triangleright r) \\ \text{function composition : } &= (x \Rightarrow q \triangleright (x \triangleright \text{sw})) \\ \text{use Eq. (13.32) : } &= x \Rightarrow x \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M} \\ \text{expand argument : } &= (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M} . \end{aligned} \quad (13.48)$$

Substituting this expression into Eq. (13.47), we get

$$\begin{aligned} m \triangleright (\text{sw} ; (r \Rightarrow q \triangleright r))^{\uparrow M} ; \text{ftn}_M \\ \text{use Eq. (13.48) : } &= m \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M} ; \text{ftn}_M^{\uparrow M} \\ \text{naturality of } \text{ftn}_M : &= m \triangleright \text{ftn}_M ; (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M} . \end{aligned}$$

Now write the left-hand side of the law:

$$\begin{aligned} q \triangleright (m \triangleright \text{ftn}_M ; \text{sw}) &= q \triangleright (m \triangleright \text{ftn}_M \triangleright \text{sw}) \\ \text{use Eq. (13.32) : } &= m \triangleright \text{ftn}_M \triangleright (r \Rightarrow q \triangleright \text{pu}_M^{\downarrow H} ; r)^{\uparrow M} . \end{aligned}$$

This is equal to the right-hand side we just obtained.

Statement 13.6.1.5 The monadic naturality laws in Theorem 13.4.8.1 hold for the `swap` function defined by Eq. (13.31) and the base monad $L \triangleq R$.

Proof The monadic naturality laws are

$$\text{sw}_{R,\text{Id}} = \text{id} , \quad \text{sw}_{R,M} ; \phi^{\uparrow R} = \phi ; \text{sw}_{R,N} , \quad \text{sw}_{R,M} ; \theta = \theta^{\uparrow M} ,$$

where $\phi : M \rightsquigarrow N$ and $\theta : R \rightsquigarrow \text{Id}$ are arbitrary monadic morphisms.

To verify the first law, set $M = \text{Id}$ in Eq. (13.32) and get ***

$$\begin{aligned} q \triangleright (m \triangleright \text{sw}_{R,\text{Id}}) & \\ \text{use Eq. (13.32)} : &= m \triangleright (r \Rightarrow q \triangleright \underline{\text{pu}_M^{\downarrow H}} ; r)^{\uparrow M} \\ \text{use } M = \text{Id} \text{ and } \text{pu}_M = \text{id} : &= \underline{m \triangleright (r \Rightarrow q \triangleright r)} \\ \text{apply to argument } m : &= q \triangleright \underline{m} = q \triangleright (m \triangleright \text{id}) . \end{aligned}$$

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:

$$\begin{aligned} q \triangleright (m \triangleright \text{sw}_{R,M} ; \phi^{\uparrow R}) &= q \triangleright (m \triangleright \text{sw}_{R,M} \triangleright \phi^{\uparrow R}) \\ \text{definition of } \uparrow^R : &= q \triangleright (\phi^{\downarrow H} ; (m \triangleright \text{sw}_{R,M}) ; \phi) \\ \triangleright \text{ notation} : &= (\underline{q \triangleright \phi^{\downarrow H}}) \triangleright (m \triangleright \underline{\text{sw}_{R,M}}) \triangleright \phi \\ \text{use Eq. (13.32)} : &= m \triangleright (r \Rightarrow q \triangleright \phi^{\downarrow H} \triangleright \underline{\text{pu}_M^{\downarrow H}} ; r)^{\uparrow M} \triangleright \phi \\ \text{composition law for } H : &= m \triangleright (r \Rightarrow q \triangleright (\underline{\text{pu}_M^{\downarrow H}} ; \phi)^{\downarrow H} ; r)^{\uparrow M} ; \phi \\ \text{identity law for } \phi : &= m \triangleright (r \Rightarrow q \triangleright \underline{\text{pu}_N^{\downarrow H}} ; r)^{\uparrow M} ; \phi \\ \text{naturality of } \phi : &= m \triangleright \phi ; (r \Rightarrow q \triangleright \underline{\text{pu}_N^{\downarrow H}} ; r)^{\uparrow N} . \end{aligned}$$

The right-hand side, when applied to m and q , gives the same expression:

$$\begin{aligned} q \triangleright (m \triangleright \phi ; \text{sw}_{R,N}) &= q \triangleright (m \triangleright \phi \triangleright \underline{\text{sw}_{R,N}}) \\ \text{use Eq. (13.32)} : &= m \triangleright \phi \triangleright (r \Rightarrow q \triangleright \underline{\text{pu}_N^{\downarrow H}} ; r)^{\uparrow N} . \end{aligned}$$

To argue that the third law holds,³ apply the left-hand side to m and q :

$$\begin{aligned} q \triangleright (m \triangleright \text{sw}_{R,M} ; \theta) &= q \triangleright (m \triangleright \text{sw}_{R,M} \triangleright \theta) \\ &= q \triangleright ((q_1 \Rightarrow m \triangleright (r \Rightarrow q_1 \triangleright \underline{\text{pu}_M^{\downarrow H}} ; r)^{\uparrow M}) \triangleright \theta) . \end{aligned} \tag{13.49}$$

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

$$\theta : \forall A. (H^A \Rightarrow A) \Rightarrow A .$$

To implement a function of this type, we need to write code that takes an argument of type $H^A \Rightarrow A$ 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. The only possibility to implement a function θ with the required type signature seems to be by substituting a value of type H^A into the given argument of type $H^A \Rightarrow A$, which will return the result of type A . So,⁴ we need to produce a value of type H^A for an

³I could not find a fully rigorous proof of the third monadic naturality law. Below I will indicate the part of the proof that lacks rigor.

⁴This is where an argument is lacking: I did not prove that the type $\forall A. (H^A \Rightarrow A) \Rightarrow A$ is really equivalent to $\forall A. H^A$. With this assumption, the proof is rigorous.

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^\dagger :

$$\begin{aligned} \forall A. H^A &\cong \forall A. \underline{1} \Rightarrow H^A \\ \text{use identity } (A \Rightarrow \underline{1}) \cong \underline{1} : & \quad \cong \forall A. (A \Rightarrow \underline{1}) \Rightarrow H^A \\ \text{contravariant Yoneda identity :} & \quad \cong H^\dagger \quad . \end{aligned}$$

So, we can construct a θ if we store a value h_1 of type H^\dagger and compute $h : H^A$ as

$$h^{H^A} = h_1^{H^\dagger} \triangleright (a^{A \Rightarrow \underline{1}} \Rightarrow 1)^{\downarrow H} \quad .$$

Given a fixed value $h_1 : H^\dagger$, the code of θ is therefore

$$(r^{H^A \Rightarrow A}) \triangleright \theta \triangleq h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H} \triangleright r \quad . \quad (13.50)$$

Let us check whether this θ is a monadic morphism $R \rightsquigarrow \text{Id}$. We need to verify the two laws of monadic morphisms,

$$\text{pu}_R ; \theta = \text{id} \quad , \quad \text{ftn}_R ; \theta = \theta^{\uparrow R} ; \theta = \theta ; \theta \quad .$$

The identity law, applied to an arbitrary $x : A$, is

$$\begin{aligned} x \triangleright \text{pu}_R ; \theta &= (x \triangleright \text{pu}_R) \triangleright \theta \\ \text{definition of } r \triangleright \theta : &= h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H} \triangleright (x \triangleright \text{pu}_R) \\ \text{definition of } x \triangleright \text{pu}_R : &= (h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H}) \triangleright (_ \Rightarrow x) \\ \text{function composition :} &= x \quad . \end{aligned}$$

This verifies the identity law.

The composition law, applied to an arbitrary $r : R^{R^A}$, expands to

$$\begin{aligned} r \triangleright \text{ftn}_R ; \theta &= r \triangleright \text{ftn}_R \triangleright \theta \\ \text{definition of } \text{ftn}_R : &= r \triangleright (t \Rightarrow q \Rightarrow q \triangleright (q \triangleright (x \Rightarrow q \triangleright x)^{\downarrow H} ; t)) \triangleright \theta \\ \text{apply to } r : &= (q \Rightarrow q \triangleright (q \triangleright (x \Rightarrow q \triangleright x)^{\downarrow H} ; r)) \triangleright \theta \\ \text{definition (13.50) of } \theta : &= h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H} \triangleright (q \Rightarrow q \triangleright (q \triangleright (x \Rightarrow q \triangleright x)^{\downarrow H} ; r)) \\ \text{apply to } h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H} : &= h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H} \triangleright (h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H} ; (x \Rightarrow \dots)^{\downarrow H} ; r) \\ \text{composition under } H : &= h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H} \triangleright (h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H} \triangleright r) \\ \text{definition (13.50) of } \theta : &= r \triangleright \theta \triangleright \theta = r \triangleright \theta ; \theta \quad . \end{aligned}$$

This verifies the composition law; so θ is indeed a monadic morphism.

Using the code of θ defined in Eq. (13.50), 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. (13.49) and is rewritten as

$$\begin{aligned} &q \triangleright ((q_1 \Rightarrow m \triangleright (r \Rightarrow q_1 \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M} ; \theta) \\ \text{definition of } \theta : &= q \triangleright (h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H} ; (q_1 \Rightarrow m \triangleright (r \Rightarrow q_1 \triangleright \text{pu}_M^{\downarrow H} ; r))^{\uparrow M}) \\ \text{apply to argument :} &= q \triangleright (m \triangleright (r \Rightarrow h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H} ; \text{pu}_M^{\downarrow H} ; r))^{\uparrow M} \\ H's \text{ composition law :} &= q \triangleright (m \triangleright (r \Rightarrow h_1 \triangleright (\text{pu}_M^{\downarrow H} ; (_ \Rightarrow 1)^{\downarrow H} ; r))^{\uparrow M}) \\ \text{compose functions :} &= q \triangleright (m \triangleright (r \Rightarrow h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H} ; r))^{\uparrow M} \quad . \end{aligned}$$

The right-hand side is

$$\begin{aligned}
 q &\triangleright (m \triangleright \theta^{\uparrow M}) \\
 \text{function expansion : } &= q \triangleright (m \triangleright (r \Rightarrow r \triangleright \theta)^{\uparrow M}) \\
 \text{definition (13.50) of } \theta : &= q \triangleright (m \triangleright (r \Rightarrow h_1 \triangleright (_ \Rightarrow 1)^{\downarrow H} \triangleright r)^{\uparrow M}) \quad .
 \end{aligned}$$

This expression is now the same as the left-hand side.

13.6.2 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 13.6.2.1 The composition $R_1^{R_2}$ 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 Theorem 13.3.6, 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

$$T_{R_1}^{T_{R_2}^M} = R_1 \circ T_{R_2}^M = R_1 \circ (R_2 \circ M) = R_1 \circ R_2 \circ M \quad .$$

Therefore $T^M \triangleq R_1 \circ R_2 \circ M$ is a 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.

Example 13.6.2.2 Consider the functor composition of the `Search` monad $R_1^A \triangleq (A \Rightarrow Q) \Rightarrow A$ and the `Reader` monad $R_2^A \triangleq Z \Rightarrow A$:

$$P^A \triangleq ((Z \Rightarrow A) \Rightarrow Q) \Rightarrow Z \Rightarrow A \quad .$$

It follows from Statement 13.6.2.1 that the functor P^\bullet is a rigid monad; so P 's transformer is of the composed-outside kind. The transformed monad for any foreign monad M is

$$T^A \triangleq ((Z \Rightarrow M^A) \Rightarrow Q) \Rightarrow Z \Rightarrow M^A \quad .$$

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 `Search` and the `Reader` monads are special cases of the `Choice` monad construction (Section 13.6.1) where the contrafunctor H is chosen to be $H^A \triangleq A \Rightarrow Q$ and $H^A \triangleq Z$ respectively, we can use Eq. (13.27) 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)
}

```

Does the composed monad have swap ? The definitions of the monad methods for the composed monads are somewhat complicated. In Section 13.6.1, we have proved the monad transformer laws for $T_R^M \triangleq R \circ M$ by defining a swap function with type signature

$$\text{sw}_{R,M} : M \circ R \rightsquigarrow R \circ M ,$$

and proving its laws. Suppose S is also a rigid monad; then the composed monad $R \circ S$ is a rigid monad. Does $T \triangleq R \circ S$ have a suitable swap function,

$$\text{sw}_{T,M} : M \circ R \circ S \rightsquigarrow R \circ S \circ M ,$$

satisfying all the required laws? 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, because 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 .

Theorem 13.6.2.3 Assume that two monads R and S both have swap methods, $\text{sw}_{R,M}$ and $\text{sw}_{S,M}$, satisfying the 8 laws listed in Theorems 13.4.2.1 and 13.4.8.1. Additionally, assume that the **3-swap law** holds with respect to an arbitrary monad M ,

$$\text{sw}_{R,S}^{\uparrow M} ; \text{sw}_{R,M} ; \text{sw}_{S,M}^{\uparrow R} = \text{sw}_{S,M} ; \text{sw}_{R,M}^{\uparrow S} ; \text{sw}_{R,S} . \quad (13.51)$$

$$\begin{array}{ccccc} & \xrightarrow{\text{sw}_{R,S}^{\uparrow M}} & & & \\ M \circ S \circ R & \xrightarrow{\text{sw}_{R,S}^{\uparrow M}} & M \circ R \circ S & \xrightarrow{\text{sw}_{R,M}} & R \circ M \circ S \\ \text{sw}_{S,M} \downarrow & & & & \downarrow \text{sw}_{S,M}^{\uparrow R} \\ S \circ M \circ R & \xrightarrow{\text{sw}_{R,M}^{\uparrow S}} & S \circ R \circ M & \xrightarrow{\text{sw}_{R,S}} & R \circ S \circ M \end{array}$$

Then the composed monad $T \triangleq R \circ S$ also has a swap method defined by

$$\text{sw}_{T,M} = \text{sw}_{R,M} ; \text{sw}_{S,M}^{\uparrow R} , \quad (13.52)$$

$$\begin{array}{ccc} M \circ R \circ S & \xrightarrow{\text{sw}_{R,M}} & R \circ M \circ S \\ & \searrow \text{sw}_{T,M} \triangleq & \downarrow \text{sw}_{S,M}^{\uparrow R} \\ & & R \circ S \circ M \end{array}$$

which 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. (13.51) holds, where M is an arbitrary monad. The monad methods of T are defined by Eqs. (13.14)–(13.15) after renaming $L \triangleq R$ and $M \triangleq S$:

$$\text{pu}_T = \text{pu}_S ; \text{pu}_R , \quad (13.53)$$

$$\text{ftn}_T = \text{sw}_{R,S}^{\uparrow R} ; \text{ftn}_R ; \text{ftn}_S^{\uparrow R} . \quad (13.54)$$

As usual, we do not need to verify the naturality law for $\text{sw}_{T,M}$.

Identity laws To verify the outer-identity law:

$$\begin{aligned}
 & \text{expect to equal } \text{pu}_T : \quad \text{pu}_T^{\uparrow M} ; \text{sw}_{T,M} \\
 & \text{use Eqs. (13.52)-(13.53)} : \quad = \text{pu}_S^{\uparrow M} ; \underline{\text{pu}_R^{\uparrow M} ; \text{sw}_{R,M}} ; \text{sw}_{S,M}^{\uparrow R} \\
 & \text{outer-identity law for } \text{sw}_{R,M} : \quad = \text{pu}_S^{\uparrow M} ; \underline{\text{pu}_R ; \text{sw}_{S,M}^{\uparrow R}} \\
 & \text{naturality of } \text{pu}_R : \quad = \underline{\text{pu}_S^{\uparrow M} ; \text{sw}_{S,M} ; \text{pu}_R} \\
 & \text{outer-identity law for } \text{sw}_{S,M} : \quad = \text{pu}_S ; \text{pu}_R \\
 & \text{use Eq. (13.53)} : \quad = \text{pu}_T .
 \end{aligned}$$

To verify the inner-identity law:

$$\begin{aligned}
 & \text{expect to equal } \text{pu}_M^{\uparrow T} : \quad \text{pu}_M ; \underline{\text{sw}_{T,M}} \\
 & \text{use Eq. (13.52)} : \quad = \underline{\text{pu}_M ; \text{sw}_{R,M}} ; \text{sw}_{S,M}^{\uparrow R} \\
 & \text{inner-identity law for } \text{sw}_{R,M} : \quad = \text{pu}_M^{\uparrow R} ; \text{sw}_{S,M}^{\uparrow R} = (\underline{\text{pu}_M ; \text{sw}_{S,M}})^{\uparrow R} \\
 & \text{inner-identity law for } \text{sw}_{S,M} : \quad = \underline{\text{pu}_M^{\uparrow S \uparrow R}} = \text{pu}_M^{\uparrow T} .
 \end{aligned}$$

Interchange laws The outer-interchange law is

$$\text{ftn}_T^{\uparrow M} ; \text{sw}_{T,M} = \text{sw}_{T,M} ; \text{sw}_{T,M}^{\uparrow T} ; \text{ftn}_T . \quad (13.55)$$

We will use the outer-interchange laws for $\text{sw}_{R,M}$ and $\text{sw}_{S,M}$:

$$\text{ftn}_R^{\uparrow M} ; \text{sw}_{R,M} = \text{sw}_{R,M} ; \text{sw}_{R,M}^{\uparrow R} ; \text{ftn}_R , \quad (13.56)$$

$$\text{ftn}_S^{\uparrow M} ; \text{sw}_{S,M} = \text{sw}_{S,M} ; \text{sw}_{S,M}^{\uparrow S} ; \text{ftn}_S . \quad (13.57)$$

Begin with the right-hand side of Eq. (13.55) since it is more complicated:

$$\begin{aligned}
 & \text{sw}_{T,M} ; \text{sw}_{T,M}^{\uparrow T} ; \text{ftn}_T \\
 & \text{definitions} : \quad = \text{sw}_{R,M} ; \underline{\text{sw}_{S,M}^{\uparrow R} ; (\text{sw}_{R,M} ; \text{sw}_{S,M}^{\uparrow R})^{\uparrow S \uparrow R} ; \text{sw}_{R,S}^{\uparrow R} ; \text{ftn}_R ; \text{ftn}_S^{\uparrow R}} \\
 & \uparrow R\text{-composition} : \quad = \text{sw}_{R,M} ; (\text{sw}_{S,M} ; \text{sw}_{R,M}^{\uparrow S} ; \underline{\text{sw}_{S,M}^{\uparrow R \uparrow S} ; \text{sw}_{R,S}})^{\uparrow R} ; \text{ftn}_R ; \text{ftn}_S^{\uparrow R} \\
 & \text{sw's naturality} : \quad = \text{sw}_{R,M} ; (\text{sw}_{S,M} ; \underline{\text{sw}_{R,M}^{\uparrow S} ; \text{sw}_{R,S}} ; \text{sw}_{S,M}^{\uparrow S \uparrow R})^{\uparrow R} ; \text{ftn}_R ; \text{ftn}_S^{\uparrow R} \\
 & \text{Eq. (13.51)} : \quad = \text{sw}_{R,M} ; (\underline{\text{sw}_{R,S}^{\uparrow M} ; \text{sw}_{R,M} ; \text{sw}_{S,M}^{\uparrow R}} ; \text{sw}_{S,M}^{\uparrow S \uparrow R})^{\uparrow R} ; \text{ftn}_R ; \text{ftn}_S^{\uparrow R} .
 \end{aligned}$$

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. (13.55) into the same form:

$$\begin{aligned}
 & \text{ftn}_T^{\uparrow M} ; \text{sw}_{T,M} \\
 & (13.52), (13.54) : \quad = (\text{sw}_{R,S}^{\uparrow R} ; \text{ftn}_R ; \underline{\text{ftn}_S^{\uparrow R}})^{\uparrow M} ; \text{sw}_{R,M} ; \text{sw}_{S,M}^{\uparrow R} \\
 & \text{naturality} : \quad = \text{sw}_{R,S}^{\uparrow R \uparrow M} ; \underline{\text{ftn}_R^{\uparrow M} ; \text{sw}_{R,M}} ; \text{ftn}_S^{\uparrow M \uparrow R} ; \text{sw}_{S,M}^{\uparrow R} \\
 & \text{Eq. (13.56)} : \quad = \text{sw}_{R,S}^{\uparrow R \uparrow M} ; \text{sw}_{R,M} ; \text{sw}_{R,M}^{\uparrow R} ; \text{ftn}_R ; (\underline{\text{ftn}_S^{\uparrow M} ; \text{sw}_{S,M}})^{\uparrow R} \\
 & \text{Eq. (13.57)} : \quad = \text{sw}_{R,M} ; \text{sw}_{R,S}^{\uparrow M \uparrow R} ; \text{sw}_{R,M}^{\uparrow R} ; \text{ftn}_R ; (\underline{\text{sw}_{S,M} ; \text{sw}_{S,M}^{\uparrow S}} ; \text{ftn}_S)^{\uparrow R} \\
 & \text{naturality} : \quad = \text{sw}_{R,M} ; \underline{\text{sw}_{R,S}^{\uparrow M \uparrow R} ; \text{sw}_{R,M}^{\uparrow R} ; (\text{sw}_{S,M} ; \text{sw}_{S,M}^{\uparrow S})^{\uparrow R \uparrow R}} ; \text{ftn}_R ; \text{ftn}_S^{\uparrow R} \\
 & \text{composition} : \quad = \text{sw}_{R,M} ; (\underline{\text{sw}_{R,S}^{\uparrow M} ; \text{sw}_{R,M} ; \text{sw}_{S,M}^{\uparrow R}} ; \text{sw}_{S,M}^{\uparrow S \uparrow R})^{\uparrow R} ; \text{ftn}_R ; \text{ftn}_S^{\uparrow R} .
 \end{aligned}$$

Both sides of the outer-interchange law (13.55) are now equal.

The proof of the inner-interchange law is simpler: the law says

$$\text{ftn}_M \circ \text{sw}_{T,M} = \text{sw}_{T,M}^{\uparrow M} \circ \text{sw}_{T,M} \circ \text{ftn}_M^{\uparrow T} . \quad (13.58)$$

We will use the inner-interchange laws for $\text{sw}_{R,M}$ and $\text{sw}_{S,M}$:

$$\text{ftn}_M \circ \text{sw}_{R,M} = \text{sw}_{R,M}^{\uparrow M} \circ \text{sw}_{R,M} \circ \text{ftn}_M^{\uparrow R} , \quad (13.59)$$

$$\text{ftn}_M \circ \text{sw}_{S,M} = \text{sw}_{S,M}^{\uparrow M} \circ \text{sw}_{S,M} \circ \text{ftn}_M^{\uparrow S} . \quad (13.60)$$

Begin with the left-hand side of Eq. (13.58):

$$\begin{aligned} \text{ftn}_M \circ \text{sw}_{T,M} &= \underline{\text{ftn}_M \circ \text{sw}_{R,M} \circ \text{sw}_{S,M}^{\uparrow R}} \\ \text{use Eq. (13.59)} : &= \text{sw}_{R,M}^{\uparrow M} \circ \text{sw}_{R,M} \circ \underline{\text{ftn}_M^{\uparrow R} \circ \text{sw}_{S,M}^{\uparrow R}} \\ \text{use Eq. (13.60) under } \uparrow R : &= \text{sw}_{R,M}^{\uparrow M} \circ \text{sw}_{R,M} \circ (\text{sw}_{S,M}^{\uparrow M} \circ \text{sw}_{S,M} \circ \text{ftn}_M^{\uparrow S})^{\uparrow R} \\ \text{composition under } \uparrow R : &= \text{sw}_{R,M}^{\uparrow M} \circ \text{sw}_{R,M} \circ \underline{\text{sw}_{S,M}^{\uparrow M \uparrow R} \circ \text{sw}_{S,M}^{\uparrow R} \circ \text{ftn}_M^{\uparrow S \uparrow R}} \end{aligned}$$

The right-hand side of Eq. (13.58) is

$$\begin{aligned} \text{sw}_{T,M}^{\uparrow M} \circ \text{sw}_{T,M} \circ \text{ftn}_M^{\uparrow T} \\ \text{use Eq. (13.52)} : &= (\text{sw}_{R,M} \circ \text{sw}_{S,M}^{\uparrow R})^{\uparrow M} \circ \text{sw}_{R,M} \circ \text{sw}_{S,M}^{\uparrow R} \circ \text{ftn}_M^{\uparrow S \uparrow R} \\ \text{naturality of } \text{sw}_{R,M} : &= \text{sw}_{R,M}^{\uparrow M} \circ \underline{\text{sw}_{R,M} \circ \text{sw}_{S,M}^{\uparrow M \uparrow R} \circ \text{sw}_{S,M}^{\uparrow R} \circ \text{ftn}_M^{\uparrow S \uparrow R}} . \end{aligned}$$

The right-hand side is now equal to the left-hand side.

Monadic naturality laws We need to verify that the three laws,

$$\text{sw}_{T,\text{Id}} = \text{id} , \quad \text{sw}_{T,M} \circ \phi^{\uparrow T} = \phi \circ \text{sw}_{T,N} , \quad \text{sw}_{T,M} \circ \theta = \theta^{\uparrow M} ,$$

hold for any monadic morphisms $\phi : M \rightsquigarrow N$ and $\theta : T \rightsquigarrow \text{Id}$. We may assume that these laws already hold for $\text{sw}_{R,M}$ and $\text{sw}_{S,M}$.

To verify the first law, write

$$\begin{aligned} \text{expect to equal id} : & \text{sw}_{T,\text{Id}} \\ \text{use Eq. (13.52)} : &= \text{sw}_{R,\text{Id}} \circ \text{sw}_{S,\text{Id}}^{\uparrow R} \\ \text{first law for } \text{sw}_{R,\text{Id}} \text{ and } \text{sw}_{S,\text{Id}} : &= \text{id} \circ \text{id}^{\uparrow R} = \text{id} . \end{aligned}$$

To verify the second law, write

$$\begin{aligned} \text{expect to equal } \phi \circ \text{sw}_{T,N} : & \text{sw}_{T,M} \circ \phi^{\uparrow T} \\ \text{use Eq. (13.52)} : &= \text{sw}_{R,M} \circ \text{sw}_{S,M}^{\uparrow R} \circ \phi^{\uparrow S \uparrow R} \\ \text{second law for } \text{sw}_{S,M} \text{ under } \uparrow R : &= \underline{\text{sw}_{R,M} \circ (\phi \circ \text{sw}_{S,N})^{\uparrow R}} \\ \text{second law for } \text{sw}_{R,M} : &= \phi \circ \text{sw}_{R,N} \circ \text{sw}_{S,N}^{\uparrow R} = \phi \circ \underline{\text{sw}_{T,N}} . \end{aligned}$$

To verify the third law, we begin with the left-hand side,

$$\text{sw}_{T,M} \circ \theta = \text{sw}_{R,M} \circ \text{sw}_{S,M}^{\uparrow R} \circ \theta .$$

At this point, no relationship or law applies to the intermediate expression $\text{sw}_{S,M}^{\uparrow R} ; \theta$, so we need additional information to proceed. This information is given by Lemma 13.6.2.4, which expresses $\theta = \theta_R ; \theta_S$ where θ_R and θ_S are monadic morphisms. So, we may use θ_R and θ_S with the monadic morphism laws for R and S , in particular with the third law:

$$\text{sw}_{R,M} ; \theta_R = \theta_R^{\uparrow M} , \quad \text{sw}_{S,M} ; \theta_S = \theta_S^{\uparrow M} .$$

Now we can finish the proof of the third monadic naturality law:

$$\begin{aligned} \text{expect to equal } \theta^{\uparrow M} : & \text{ sw}_{T,M} ; \underline{\theta} \\ \text{Lemma 13.6.2.4} : & = \text{sw}_{R,M} ; \underline{\text{sw}_{S,M}^{\uparrow R} ; \theta_R ; \theta_S} \\ \text{naturality of } \theta_R : & = \underline{\text{sw}_{R,M} ; \theta_R} ; \underline{\text{sw}_{S,M} ; \theta_S} \\ \text{third law for } R \text{ and } S : & = \theta_R^{\uparrow M} ; \theta_S^{\uparrow M} = \theta^{\uparrow M} . \end{aligned}$$

Lemma If R is a rigid monad with a lawful `swap` method $\text{sw}_{R,M}$ and S is any (not necessarily rigid) monad then any monadic morphism $\theta : R \circ S \rightsquigarrow \text{Id}$ can be expressed as a composition $\theta = \theta_R ; \theta_S$ where

$$\theta_R \triangleq \text{pu}_S^{\uparrow R} ; \theta , \quad \theta_S \triangleq \text{pu}_R ; \theta$$

are monadic morphisms. In other words, all “runners” θ for the composed monad $R \circ S$ can be written as a function composition of some “runners” $\theta_R : R \rightsquigarrow \text{Id}$ and $\theta_S : S \rightsquigarrow \text{Id}$ for the monads R and S .

Proof How can we find θ_R and θ_S from the given “runner” θ ? Consider that θ will evaluate the operations of both monads R and S , while θ_R can evaluate 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 $R \rightsquigarrow R \circ S$. A suitable function of this type is

$$\text{pu}_S^{\uparrow R} : R^A \Rightarrow R^{S^A} .$$

So, $\theta_R = \text{pu}_S^{\uparrow R} ; \theta$ has the correct type signature, $R^A \Rightarrow A$. Similarly, $\theta_S = \text{pu}_R ; \theta$ has the correct type signature, $S^A \Rightarrow A$. So, we can define θ_R and θ_S from the given “runner” θ as

$$\theta_R \triangleq \text{pu}_S^{\uparrow R} ; \theta , \quad \theta_S \triangleq \text{pu}_R ; \theta .$$

Since pu_R and $\text{pu}_S^{\uparrow R}$ are the lifting and the base lifting of the monad transformer $T_R^S = R \circ S$, Theorem 13.4.8.1 shows that pu_R and $\text{pu}_S^{\uparrow R}$ are monadic morphisms. Since the composition of monadic morphisms is again a monadic morphism (Statement 13.3.4.3), we see that θ_R and θ_S as defined above are monadic morphisms.

It remains to verify that $\theta = \theta_R ; \theta_S$:

$$\begin{aligned} \text{expect to equal } \theta : & \theta_R ; \theta_S = \text{pu}_S^{\uparrow R} ; \theta ; \underline{\text{pu}_R ; \theta} \\ \text{naturality of } \theta : & = \text{pu}_S^{\uparrow R} ; \text{pu}_R^{\uparrow T} ; \theta ; \theta = \text{pu}_S^{\uparrow R} ; \text{pu}_R^{\uparrow S \uparrow R} ; \underline{\theta ; \theta} \\ \text{composition law of } \theta : & = \text{pu}_S^{\uparrow R} ; \text{pu}_R^{\uparrow S \uparrow R} ; \underline{\text{ftn}_T ; \theta} . \end{aligned}$$

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 identity:

$$\begin{aligned} \text{expect to equal id} : & = \text{pu}_S^{\uparrow R} ; \text{pu}_R^{\uparrow S \uparrow R} ; \underline{\text{ftn}_T} . \\ \text{use Eq. (13.15)} : & = \text{pu}_S^{\uparrow R} ; \underline{\text{pu}_R^{\uparrow S \uparrow R} ; \text{sw}_{R,S}^{\uparrow R} ; \text{ftn}_R ; \text{ftn}_S^{\uparrow R}} \\ \text{outer-identity law for } \text{sw}_{R,S} : & = \text{pu}_S^{\uparrow R} ; \underline{\text{pu}_R^{\uparrow R} ; \text{ftn}_R ; \text{ftn}_S^{\uparrow R}} \\ \text{identity law for } R : & = \underline{\text{pu}_S^{\uparrow R} ; \text{ftn}_S^{\uparrow R}} \\ \text{identity law for } S : & = \text{id} . \end{aligned}$$

13.6.3 Rigid monad construction 3: product

Statement 13.6.3.1 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. It follows from Theorem 13.8.1.1 (whose proof does not depend on any of the results of this section) that the lawful monad transformer $T^{M,A}$ for the product monad $R_1^A \times R_2^A$ is the product of transformers

$$T_{R_1}^{M,A} \times T_{R_2}^{M,A} = R_1^{M^A} \times R_2^{M^A},$$

where M is the foreign monad. This is the required composed-outside transformer for the monad $R_1^A \times R_2^A$.

13.6.4 Rigid monad construction 4: selector

The **selector monad** $S^A \triangleq F^{A \Rightarrow R^Q} \Rightarrow R^A$ is rigid if R^\bullet is a rigid monad, F^\bullet is any functor, and Q is any fixed type.

13.6.5 Rigid functors

The 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 13.6), does not refer to any monad transformer.

Definition of rigid functors R is a **rigid functor** if there exists a natural transformation `fuseIn` (short notation “ fi_R ”) with the type signature

$$\text{fi}_R : \forall(A, B). (A \Rightarrow R^B) \Rightarrow R^{A \Rightarrow B}$$

satisfying the non-degeneracy law 13.62 (see below).

Not all functors admit a natural transformation with the type signature of `fuseIn`. For example, the functor $F^A \triangleq Z + A$ is not rigid because the required type signature

$$\forall(A, B). (A \Rightarrow Z + B) \Rightarrow Z + (A \Rightarrow B)$$

cannot be implemented. However, any functor R admits the opposite natural transformation `fuseOut` (short notation fo_R), defined by

$$\begin{aligned} \text{fo} &: \forall(A, B). R^{A \Rightarrow B} \Rightarrow A \Rightarrow R^B, \\ \text{fo}(r) &= a \Rightarrow (f^{A \Rightarrow B} \Rightarrow f(a))^{\uparrow R} r. \end{aligned} \tag{13.61}$$

The method `fuseIn` must satisfy the nondegeneracy law

$$\text{fi}_R ; \text{fo}_R = \text{id}^{(A \Rightarrow R^B) \Rightarrow (A \Rightarrow R^B)}. \tag{13.62}$$

The opposite relation does not hold in general, $\text{fo}_R ; \text{fi}_R \neq \text{id}$ (see Example 13.6.5.2).

Note that the type signature of fi_R is the same as the type signature of `swap` with respect to the Reader monad,

$$\text{sw}_{R,M} : M^{R^A} \Rightarrow R^{M^A} \cong (Z \Rightarrow R^A) \Rightarrow R^{Z \Rightarrow A} \text{ if we set } M^A \triangleq Z \Rightarrow A.$$

So we are prompted to ask whether any rigid monad having a `swap` method might also admit `fuseIn`. It turns out that all rigid monads (regardless of the existence of `swap`) are also rigid functors. While proving that statement, we will not assume that `swap` exists for the monad R , but will use directly the definition of rigid monads via the composed-outside monad transformer.

Statement 13.6.5.1 A rigid monad R is also a rigid functor.

Proof By assumption, the monad R has the composed-outside monad transformer $T_R^M = R \circ M$ and the corresponding monad method ftn_T . We can define the transformation fi_R in the same way as Eq. (13.16) defined the `swap` method via ftn_T . To use that formula, we need to set the foreign monad M to be the `Reader` monad, $M^B \triangleq A \Rightarrow B$, with the fixed environment type A :

$$\text{fi}_R(f : A \Rightarrow R^B) = \text{pu}_M^{\uparrow R \uparrow M} ; \text{pu}_R ; \text{ftn}_T \quad (13.63)$$

$$\begin{array}{ccccc} & & \text{pu}_M^{\uparrow R \uparrow M} & & \\ A \Rightarrow R^B & \xrightarrow{\text{pu}_M^{\uparrow R \uparrow M}} & A \Rightarrow R^{A \Rightarrow B} & \xrightarrow{\text{pu}_R} & R^{A \Rightarrow R^{A \Rightarrow B}} \\ & \searrow & & & \downarrow \text{ftn}_T \\ & & \text{fi}_R \triangleq & & R^{A \Rightarrow B} \end{array}$$

Since M is the `Reader` monad with the environment type A , we have

$$\text{pu}_M(x) = (_ : A \Rightarrow x) \quad , \quad (f : X \Rightarrow Y)^{\uparrow M}(r : A \Rightarrow X) = r ; f \quad .$$

The type signature of the function `fuseOut`,

$$\text{fo} : R^{A \Rightarrow B} \Rightarrow A \Rightarrow R^B \quad ,$$

resembles “running” the composed monad R^{M^B} into R^B , given a value of the environment (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

$$\phi_a^{M^X \Rightarrow X} = (m^{A \Rightarrow X} \Rightarrow m(a)) \quad . \quad (13.64)$$

So we are inspired to use the runner law for the monad transformer T_R^M . That law (which holds since we assumed that T_R^M satisfies all laws) says that the lifted “runner” $\phi_a^{\uparrow R}$ is a monadic morphism $T_R^M \rightsquigarrow T_R^{\text{Id}} \cong T_R^M \rightsquigarrow R$. The monadic morphism law for $\phi_a^{\uparrow R}$ is then written as

$$\text{ftn}_T ; \phi_a^{\uparrow R} = \phi_a^{\uparrow R \uparrow M \uparrow R} ; \phi_a^{\uparrow R} ; \text{ftn}_R \quad . \quad (13.65)$$

How could we use this law to obtain Eq. (13.62)? Compare Eqs. (13.61) and (13.64) and derive the connection between the “runner” ϕ_a and the `fuseOut` function (which always exists for any functor R),

$$\text{fo}_R = (r \Rightarrow a \Rightarrow r \triangleright \phi_a^{\uparrow R}) \quad ,$$

and then rewrite the law (13.62) as

$$\begin{aligned} \text{expect to equal } m : & m : M^{R^B} \triangleright \text{fi}_R ; \text{fo}_R = (m \triangleright \text{fi}_R) \triangleright \text{fo}_R \\ \text{use Eq. (13.61)} : & = a \Rightarrow m \triangleright \underline{\text{fi}_R} \triangleright \phi_a^{\uparrow R} \\ \text{use Eq. (13.63)} : & = a \Rightarrow m \triangleright \text{pu}_M^{\uparrow R \uparrow M} ; \text{pu}_R ; \underline{\text{ftn}_T ; \phi_a^{\uparrow R}} \\ \text{use Eq. (13.65)} : & = a \Rightarrow m \triangleright \text{pu}_M^{\uparrow R \uparrow M} ; \underline{\text{pu}_R ; \phi_a^{\uparrow R \uparrow M \uparrow R}} ; \phi_a^{\uparrow R} ; \text{ftn}_R \\ \text{naturality of } \text{pu}_R : & = a \Rightarrow m \triangleright \text{pu}_M^{\uparrow R \uparrow M} ; \phi_a^{\uparrow R \uparrow M} ; \phi_a ; \underline{\text{pu}_R ; \text{ftn}_R} \\ \text{left identity law of } R : & = a \Rightarrow m \triangleright (\text{pu}_M ; \phi_a)^{\uparrow R \uparrow M} ; \phi_a \\ \text{identity law for } \phi_a : & = a \Rightarrow m \triangleright \underline{\phi_a} \\ \text{definition (13.64) for } \phi_a : & = \underline{(a \Rightarrow m(a))} = m \quad . \end{aligned}$$

Here we used the monadic morphism identity law for ϕ_a ,

$$\text{pu}_M ; \phi_a = (x \Rightarrow (_ \Rightarrow x)) ; (m \Rightarrow a \triangleright m) = (x \Rightarrow a \triangleright (_ \Rightarrow x)) = (x \Rightarrow x) = \text{id} \quad .$$

Example 13.6.5.2 To show that the opposite of the non-degeneracy law does not always hold, consider the rigid monads $P^A \triangleq Z \Rightarrow A$ and $R^A \triangleq (A \Rightarrow Q) \Rightarrow A$, where Q and Z are fixed types. Since all rigid monads are rigid functors, it follows that the monads P and R have methods $\text{fi}_P, \text{fo}_P, \text{fi}_R, \text{fo}_R$ satisfying the non-degeneracy law (13.62). It turns out that additionally $\text{fo}_P \circ \text{fi}_P = \text{id}$, i.e. the methods fo_P and fi_P are inverses of each other, but $\text{fo}_R \circ \text{fi}_R \neq \text{id}$.

To show that $\text{fo}_P \circ \text{fi}_P = \text{id}$, consider the type signatures of fo_P and fi_P :

$$\begin{aligned}\text{fo}_P : P^{A \Rightarrow B} &\Rightarrow A \Rightarrow P^B \cong (Z \Rightarrow A \Rightarrow B) \Rightarrow (A \Rightarrow Z \Rightarrow B) , \\ \text{fi}_P : (A \Rightarrow P^B) &\Rightarrow P^{A \Rightarrow B} \cong (A \Rightarrow Z \Rightarrow B) \Rightarrow (Z \Rightarrow A \Rightarrow B) .\end{aligned}$$

The implementations of these functions are derived uniquely from type signatures, as long as we require that these implementations are natural transformations. 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 $\text{fo}_P \circ \text{fi}_P$,

$$(\text{fo}_P \circ \text{fi}_P) : P^{A \Rightarrow B} \Rightarrow P^{A \Rightarrow B} \cong (Z \Rightarrow A \Rightarrow B) \Rightarrow (Z \Rightarrow A \Rightarrow B) .$$

There is only one implementation for this type signature as a natural transformation, namely the identity function $\text{id}^{Z \Rightarrow A \Rightarrow B}$.

For the monad R , the type signatures of fi_R and fo_R are

$$\begin{aligned}\text{fi}_R : (A \Rightarrow (B \Rightarrow Q) \Rightarrow B) &\Rightarrow ((A \Rightarrow B) \Rightarrow Q) \Rightarrow A \Rightarrow B , \\ \text{fo}_R : (((A \Rightarrow B) \Rightarrow Q) \Rightarrow A \Rightarrow B) &\Rightarrow A \Rightarrow (B \Rightarrow Q) \Rightarrow B ,\end{aligned}$$

and the implementations are again derived uniquely from type signatures,

$$\begin{aligned}f : A \Rightarrow (B \Rightarrow Q) \Rightarrow B &\triangleright \text{fi}_R = x : (A \Rightarrow B) \Rightarrow Q \Rightarrow a : A \Rightarrow f(a)(b : B \Rightarrow x(_ \Rightarrow b)) , \\ g : ((A \Rightarrow B) \Rightarrow Q) \Rightarrow A \Rightarrow B &\triangleright \text{fo}_R = a : A \Rightarrow y : B \Rightarrow Q \Rightarrow g(h : A \Rightarrow B \Rightarrow y(h(a)))(a) .\end{aligned}$$

We notice that the implementation of fi_R uses a constant function, $(_ \Rightarrow b)$, which is likely to lose information. Indeed, while $\text{fi}_R \circ \text{fo}_R = \text{id}$ as it must be due to the rigid non-degeneracy law, we find that

$$\begin{aligned}\text{expect not to equal } g : \quad g \triangleright \text{fo}_R \circ \text{fi}_R &= (g \triangleright \text{fo}_R) \triangleright \text{fi}_R \\ \text{definition of } \text{fi}_R : \quad &= x \Rightarrow a \Rightarrow \text{fo}_R(g)(a)(b \Rightarrow x(_ \Rightarrow b)) \\ \text{definition of } \text{fo}_R : \quad &= x \Rightarrow a \Rightarrow g(h : A \Rightarrow B \Rightarrow h(a) \triangleright (b \Rightarrow x(_ \Rightarrow b)))(a) \\ \text{apply to argument } b : \quad &= x \Rightarrow a \Rightarrow g(h \Rightarrow x(_ \Rightarrow h(a)))(a) .\end{aligned}$$

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 $x(_ \Rightarrow h(a))$. If the last line were equal to g , we would expect it to be $x \Rightarrow a \Rightarrow g(x)(a)$. The difference is in the first argument of g , namely we have $h \Rightarrow x(_ \Rightarrow h(a))$ instead of x . The two last expressions are not always equal; they would be equal if we had

$$(h \Rightarrow x(h)) = (h \Rightarrow x(k \Rightarrow h(k)))$$

instead of $h \Rightarrow x(_ \Rightarrow h(a))$. Consider again the argument of x in the two last expressions: $k \Rightarrow h(a)$ instead of $k \Rightarrow h(k)$. Since h is not always a constant function (h is an arbitrary function of type $A \Rightarrow B$), the two expressions $k \Rightarrow h(a)$ and $k \Rightarrow h(k)$ are generally not equal. So, we must conclude that $\text{fo}_R \circ \text{fi}_R \neq \text{id}$.

Exercise 13.6.5.3 Show that the functor $F^A \triangleq A \times A \times A$ is rigid.

Exercise 13.6.5.4 Show that the functor $F^A \triangleq \mathbb{1} + A \times A$ is not rigid.

Exercise 13.6.5.5 Show that the functor $F^A \triangleq (A \Rightarrow Z) \Rightarrow Z$ is not rigid. (Here Z is a fixed type.)

Since all rigid monads are rigid functors, we can reuse all the rigid monad constructions to obtain new rigid functors. The following statement demonstrates a construction of rigid functors that does not assume any monadic properties. It shows that the set of all rigid functors is larger than the set of all rigid monads.

Statement 13.6.5.6 The functor $S^\bullet \triangleq H^\bullet \Rightarrow P^\bullet$ 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 non-degeneracy law (13.62). The function fi_S is then defined by

$$\begin{aligned}\text{fi}_S : (A \Rightarrow H^B \Rightarrow P^B) &\Rightarrow H^{A \Rightarrow B} \Rightarrow P^{A \Rightarrow B} , \\ \text{fi}_S = f^{A \Rightarrow H^B \Rightarrow P^B} &\Rightarrow h^{H^{A \Rightarrow B}} \Rightarrow \text{fi}_P(a \Rightarrow f(a)((b \Rightarrow _) \Rightarrow b)^{\downarrow H} h) ,\end{aligned}$$

or, using the forwarding notation,

$$h \triangleright \text{fi}_S(f) = (a \Rightarrow h \triangleright (b \Rightarrow _) \Rightarrow b)^{\downarrow H} \triangleright f(a) \triangleright \text{fi}_P .$$

Let us write the definition of fo_S as well,

$$\begin{aligned}\text{fo}_S : (H^{A \Rightarrow B} \Rightarrow P^{A \Rightarrow B}) &\Rightarrow A \Rightarrow H^B \Rightarrow P^B , \\ \text{fo}_S = g^{H^{A \Rightarrow B} \Rightarrow P^{A \Rightarrow B}} &\Rightarrow a^{A \Rightarrow H^B} \Rightarrow h^{H^B} \Rightarrow \text{fo}_P\left(g(\left(p^{A \Rightarrow B} \Rightarrow p(a)\right)^{\downarrow H} h)\right)(a) ,\end{aligned}$$

or, using the forwarding notation,

$$\text{fo}_S(g) = a \triangleright (h \triangleright (p \Rightarrow p(a))^{\downarrow H} ; g ; \text{fo}_P) .$$

To verify the non-degeneracy law for S , apply both sides to some arguments; we expect $f \triangleright (\text{fi}_S ; \text{fo}_S)$ to equal f for an arbitrary $f : A \Rightarrow H^B \Rightarrow P^B$. 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 $f(a)(h)$:

$$\begin{aligned}(f \triangleright \text{fi}_S ; \text{fo}_S)(a)(h) &= (f \triangleright \text{fi}_S \triangleright \text{fo}_S)(a)(h) \\ \text{expand } \text{fo}_S : &= a \triangleright (h \triangleright (p \Rightarrow a \triangleright p)^{\downarrow H} \triangleright \text{fi}_S(f) \triangleright \text{fo}_P) \\ \text{expand } \text{fi}_S : &= a \triangleright ((a \Rightarrow h \triangleright (p \Rightarrow p(a))^{\downarrow H} ; (b \Rightarrow _) \Rightarrow b)^{\downarrow H} ; f(a)) \triangleright \text{fi}_P \triangleright \text{fo}_P \\ \text{compose } {}^{\downarrow H} &= a \triangleright ((a \Rightarrow h \triangleright ((b \Rightarrow _) \Rightarrow b) ; (p \Rightarrow p(a)))^{\downarrow H} ; f(a)) \triangleright \text{fi}_P \triangleright \text{fo}_P .\end{aligned}$$

Computing the function composition

$$(b \Rightarrow _) ; (p \Rightarrow p(a)) = (b \Rightarrow (_ \Rightarrow b)(a)) = (b \Rightarrow b) = \text{id} ,$$

and using the non-degeneracy law $\text{fi}_P ; \text{fo}_P = \text{id}$, we can simplify further:

$$\begin{aligned}a \triangleright ((a \Rightarrow h \triangleright ((b \Rightarrow _) \Rightarrow b) ; (p \Rightarrow p(a)))^{\downarrow H} ; f(a)) \triangleright \text{fi}_P \triangleright \text{fo}_P \\ \text{identity law for } H : &= a \triangleright ((a \Rightarrow h \triangleright f(a)) \triangleright \text{fi}_P ; \text{fo}_P) \\ \text{non-degeneracy :} &= a \triangleright (a \Rightarrow h \triangleright f(a)) = h \triangleright f(a) .\end{aligned}$$

This equals $f(a)(h)$, as required.

Statement 13.6.5.7 A rigid functor R is pointed; the method `pure` can be defined as

$$\text{pu}_R(x^{:A}) \triangleq \text{id}^{:R^A \Rightarrow R^A} \triangleright \text{fi}_R \triangleright (_ \Rightarrow x)^{\uparrow R} .$$

In particular, there is a selected value r_1 of type $R^{\mathbb{I}}$ (“wrapped unit”), computed as

$$r_1 \triangleq \text{pu}_R(1) = \text{id} \triangleright \text{fi}_R \triangleright (_ \Rightarrow 1)^{\uparrow R} . \quad (13.66)$$

Proof The method $\text{fi}_R : (X \Rightarrow R^Y) \Rightarrow R^{X \Rightarrow Y}$ with type parameters $X = R^A$ and $Y = A$ is applied to the identity function $\text{id} : R^A = R^A$, considered as a value of type $X \Rightarrow R^Y$. The result is a value

$$\text{fi}_R(\text{id}) : R^{R^A \Rightarrow A} .$$

The result is transformed via the raised constant function $(_\Rightarrow x)^{\uparrow R}$, which takes $R^{R^A \Rightarrow A}$ and returns R^A . The resulting code can be written as

$$\text{pu}_R(x) \triangleq (_\Rightarrow x)^{\uparrow R}(\text{fi}_R(\text{id})) = \text{id} \triangleright \text{fi}_R \triangleright (_\Rightarrow x)^{\uparrow R} .$$

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 the selected value r_1 of type R^1 .

The next theorem shows that r_1 is the *only* distinct value of the type R^1 . This means, in particular, that a rigid functor cannot be a disjunctive type defined with several constructors, such as $\mathbb{1} + A + A$ or List^A . A rigid functor's type definition must have a single constructor. This is one motivation for the name "rigid": the container R^A has a fixed shape and bears no extra information other than holding some values of type A .

Theorem 13.6.5.8 For a rigid functor R with the method fi_R satisfying the non-degeneracy law (13.62), the type R^1 is equivalent to the unit type, $R^1 \cong \mathbb{1}$. The value r_1 defined by Eq. (13.66) is the only available distinct value of type R^1 . The isomorphism map between $\mathbb{1}$ and R^1 is the function $(1 \Rightarrow r_1)$.

Proof The plan of the proof is to apply both sides of the non-degeneracy law $\text{fi}_R \circ \text{fo}_R = \text{id}$ to the identity function of type $R^1 \Rightarrow R^1$. To adapt the type parameters, consider the generic type signature of $\text{fi}_R \circ \text{fo}_R$,

$$(\text{fi}_R \circ \text{fo}_R) : (A \Rightarrow R^B) \Rightarrow (A \Rightarrow R^B) ,$$

and set $A = R^1$ and $B = \mathbb{1}$. The left-hand side of the law can be now applied to the identity function $\text{id} : R^1 \Rightarrow R^1$, which yields a value of type $R^1 \Rightarrow R^1$, i.e. a function

$$f_1 : R^1 \Rightarrow R^1 , \quad f_1 \triangleq \text{fo}_R(\text{fi}_R(\text{id})) .$$

We will show that f_1 is a constant function, $f_1 = (_\Rightarrow r_1)$, always returning the same value r_1 defined in Statement 13.6.5.7. However, the right-hand side of the non-degeneracy law applied to id is the identity function of type $R^1 \Rightarrow R^1$. So, the non-degeneracy law means that $f_1 = \text{id}$. If the identity function of type $R^1 \Rightarrow R^1$ always returns the same value (r_1), it means that r_1 is the only distinct value of the type R^1 .

To begin the proof, note that for any fixed type A , the function type $A \Rightarrow \mathbb{1}$ is equivalent to $\mathbb{1}$. This is so because there exists only one pure function of type $A \Rightarrow \mathbb{1}$, namely $(_\Rightarrow 1)$. In other words, there is only one distinct value of the type $A \Rightarrow \mathbb{1}$, and the value is the function $(_\Rightarrow 1)$. The code of this function is uniquely determined by its type signature.

The isomorphism between the types $A \Rightarrow \mathbb{1}$ and $\mathbb{1}$ is realized by the functions $u : \mathbb{1} \Rightarrow A \Rightarrow \mathbb{1}$ and $v : (A \Rightarrow \mathbb{1}) \Rightarrow \mathbb{1}$. The code of these functions is also uniquely determined by their type signatures:

$$u = (1 \Rightarrow _\stackrel{:A}{\Rightarrow} 1) , \quad v = (_\stackrel{:A \Rightarrow \mathbb{1}}{\Rightarrow} 1) .$$

Applying fi_R to the identity function of type $R^1 \Rightarrow R^1$, we obtain a value g ,

$$g : R^{R^1 \Rightarrow \mathbb{1}} , \quad g \triangleq \text{id} \triangleright \text{fi}_R .$$

Since the type $R^1 \Rightarrow \mathbb{1}$ is equivalent to $\mathbb{1}$, the type $R^{R^1 \Rightarrow \mathbb{1}}$ is equivalent to R^1 . To use this equivalence explicitly, we need to raise the isomorphisms u and v into the functor R . The isomorphism will then map $g : R^{R^1 \Rightarrow \mathbb{1}}$ to some $g_1 : R^1$ by

$$g_1 \triangleq v^{\uparrow R}(g) .$$

Substituting the definitions of g , v , and r_1 , we find that actually $g_1 = r_1$:

$$g_1 = \text{id} \triangleright \text{fi}_R \triangleright v^{\uparrow R} = \text{id} \triangleright \text{fi}_R \triangleright (_\Rightarrow 1)^{\uparrow R} = r_1 .$$

We can now map g_1 back to g via the raised isomorphism u :

$$\begin{aligned} g &= g_1 \triangleright u^{\uparrow R} = r_1 \triangleright u^{\uparrow R} \\ \text{definition of } u : &= r_1 \triangleright (1 \Rightarrow \underline{} \Rightarrow 1)^{\uparrow R} . \end{aligned} \quad (13.67)$$

Compute $\text{fo}_R(g)$ as

$$\begin{aligned} \text{fo}_R(g) &= g \triangleright \text{fo}_R \\ \text{use Eq. (13.61)} : &= a^{:R^1} \Rightarrow g \triangleright (f^{:A \Rightarrow 1} \Rightarrow f(a))^{\uparrow R} \\ \text{use Eq. (13.67)} : &= a \Rightarrow r_1 \triangleright (1 \Rightarrow \underline{}^{:A} \Rightarrow 1)^{\uparrow R} \triangleright (f^{:A \Rightarrow 1} \Rightarrow f(a))^{\uparrow R} \\ \text{composition under } \uparrow^R : &= a \Rightarrow r_1 \triangleright (1 \Rightarrow \underline{})^{\uparrow R} \\ (1 \Rightarrow 1) \text{ is identity} : &= (a \Rightarrow r_1 \triangleright \text{id}) = (a^{:R^1} \Rightarrow r_1) = (\underline{}^{:R^1} \Rightarrow r_1) . \end{aligned}$$

So, $\text{fo}_R(g) = \text{fo}_R(\text{fi}_R(\text{id}))$ is a function of type $R^1 \Rightarrow R^1$ that ignores its argument and always returns the same value r_1 .

By virtue of the non-degeneracy law, $\text{fo}_R(g) = \text{id}$. We see that the identity function $\text{id} : R^1 \Rightarrow R^1$ always returns the same value r_1 . Applying this function to an arbitrary value $x : R^1$, we get

$$x = x \triangleright \text{id} = x \triangleright \text{fo}_R(g) = x \triangleright (\underline{} \Rightarrow r_1) = r_1 .$$

It means that all values of type R^1 are equal to r_1 . So the function $1 \Rightarrow r_1$ is indeed an isomorphism between the types 1 and R^1 .

It follows from Theorem 13.6.5.8 that a rigid functor cannot be a disjunctive type with more than one constructor, so functors such as `Option` or `List` cannot be rigid. This partially explains the choice of the name “rigid”: a rigid functor has a fixed “shape” of the placement of data in it.

Statement 13.6.5.9 Product of rigid functors is rigid.

Proof ***

Statement 13.6.5.10 Product of rigid functors is rigid.

Proof ***

Some more use cases for rigid functors are shown in the next statements. The “ R -valued `flatMap`” is a generalization of `flatMap` that can handle multiple M -effects at once; more precisely, an R -container of M -effects.

Statement 13.6.5.11 ***For a rigid functor R and a monad M , an “ R -valued `flatMap`” can be defined,

$$\text{rflm}_{M,R} : (A \Rightarrow R^{M^B}) \Rightarrow M^A \Rightarrow R^{M^B} .$$

A “refactoring” is a program transformation that does not significantly change the functionality.

Statement 13.6.5.12 ***Given a rigid functor R , a refactoring function can be implemented,

$$\text{refactor} : ((A \Rightarrow B) \Rightarrow C) \Rightarrow (A \Rightarrow R^B) \Rightarrow R^C .$$

This function transforms a program $p(f^{:A \Rightarrow B}) : C$ into a program $\tilde{p}(\tilde{f}^{:A \Rightarrow R^B}) : R^C$.

13.7 Recursive monad transformers

13.7.1 Transformer for the free monad FreeT

13.7.2 Transformer for the list monad ListT

13.8 Monad transformers for monad constructions

13.8.1 Product of monad transformers

Statement 13.8.1.1 The transformer for a product of two monads is the product of transformers.

13.8.2 Free pointed monad transformer

13.9 Irregular and incomplete monad transformers

13.9.1 The state monad transformer StateT

13.9.2 The continuation monad transformer ContT

13.9.3 The codensity monad transformer CodT

The **codensity monad** over a functor F is defined as

$$\text{Cod}^{F,A} \triangleq \forall B. (A \Rightarrow F^B) \Rightarrow F^B$$

Properties:

- $\text{Cod}^{F,\bullet}$ is a monad for any functor F
- If F^\bullet is itself a monad then we have monadic morphisms $\text{inC} : F^\bullet \rightsquigarrow \text{Cod}^{F,\bullet}$ and $\text{outC} : \text{Cod}^{F,\bullet} \rightsquigarrow F^\bullet$ such that $\text{inC} ; \text{outC} = \text{id}$
- A monad transformer for the codensity monad is

$$T_{\text{Cod}}^{M,A} = \forall B. (A \Rightarrow M^{F^B}) \Rightarrow M^{F^B}$$

However, this transformer does not have the base lifting morphism

$$\text{blift} : (\forall B. (A \Rightarrow F^B) \Rightarrow F^B) \Rightarrow \forall C. (A \Rightarrow M^{F^C}) \Rightarrow M^{F^C}$$

since this type signature cannot be implemented. The codensity transformer also does not have any of the required “runner” transformations mrun and brun ,

$$\begin{aligned} \text{mrun} : (M^\bullet \rightsquigarrow N^\bullet) &\Rightarrow (\forall B. (A \Rightarrow M^{F^B}) \Rightarrow M^{F^B}) \Rightarrow \forall C. (A \Rightarrow N^{F^C}) \Rightarrow N^{F^C} , \\ \text{brun} : ((\forall B. (A \Rightarrow F^B) \Rightarrow F^B) \Rightarrow A) &\Rightarrow (\forall C. (A \Rightarrow M^{F^C}) \Rightarrow M^{F^C}) \Rightarrow M^A . \end{aligned}$$

13.10 Summary and discussion

13.10.1 Exercises

Exercise 13.10.1.1 Suppose T_L^M is a lawful monad transformer for the base monad L and a foreign monad M . Can we modify T_L^M and construct another monad transformer for L that still satisfies all transformer laws?

One possibility is to compose T with an extra layer of the monads L or M . Define U and V by

$$U^A \triangleq L^{T_L^{M,A}} , \quad V^A \triangleq M^{T_L^{M,A}} .$$

In a shorter notation, $U \triangleq L \circ T_L^M$ and $V \triangleq M \circ T_L^M$. We have the `swap` functions

$$\begin{aligned} \text{sw}_{L,T} &: T_L^M \circ L \rightsquigarrow L \circ T_L^M , \\ \text{sw}_{M,T} &: T_L^M \circ M \rightsquigarrow M \circ T_L^M , \end{aligned}$$

defined using the already given methods `lift` and `blift` of T_L^M as

$$\begin{aligned} \text{sw}_{L,T} &= \text{blift}^{\uparrow T} ; \text{ftn}_T ; \text{pu}_L , \\ \text{sw}_{M,T} &= \text{lift}^{\uparrow T} ; \text{ftn}_T ; \text{pu}_M . \end{aligned}$$

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 13.10.1.2 Show that there exist monadic morphisms between the selection $(A \Rightarrow R) \Rightarrow A$ and the continuation $(A \Rightarrow R) \Rightarrow R$ monads.

14 Recursive types

Recursion is a translation of mathematical induction into code. Since a type is the set of allowed values of a function argument, a recursive type is a set that is defined using mathematical induction.

Consider this code:

```
sealed trait NInt
final case class One(x: Int) extends NInt
final case class Two(n: NInt) extends NInt
```

14.1 Fixpoints and type recursion schemes

14.2 Row polymorphism and OO programming

14.3 Column polymorphism

14.4 Discussion

15 Co-inductive typeclasses. Comonads

15.1 Practical use

15.2 Laws and structure

15.3 Co-semigroups and co-monoids

15.4 Co-free constructions

15.5 Co-free comonads

15.6 Comonad transformers

15.7 Discussion

16 Irregular typeclasses

16.1 Distributive functors

16.2 Monoidal monads

16.3 Lenses and prisms

16.4 Discussion

Part IV

Discussions

17 Summary and outlook

Compute the smallest integer expressible as a sum of two cubed integers in more than one way.

Read a text file, split it by spaces into words, and print the word counts, sorted by decreasing count.

FPIS exercise 2.2: Check whether a sequence `Seq[A]` is sorted according to a given ordering function of type `(A, A) => Boolean`.

FPIS exercise 3.24: Implement a function `hasSubsequence` that checks whether a `List` contains another `List` as a subsequence. For instance, `List(1,2,3,4)` would have `List(1,2)`, `List(2,3)`, and `List(4)` as subsequences, among others. (Dynamic programming?)

18 “Applied functional type theory”: A proposal

What exactly is the extent of “theory” that a practicing functional programmer should know in order to be effective at writing code in the functional paradigm? This question is not yet resolved. This book presents a coherent body of theoretical knowledge that, in the author’s view, fits the description of “practicable functional programming theory”. This body of knowledge could be understood as a separate branch of computer science called **applied functional type theory** (AFTT). This is the area of theoretical computer science that serves the needs of functional programmers working as software engineers.

It is for these practitioners, rather than for academic researchers, that this book sets out to examine the functional programming inventions over the last 30 years, – such as the “functional pearls” papers¹ – and to determine the scope of theoretical material that has demonstrated its pragmatic usefulness and thus belongs to AFTT, as opposed to material that is purely academic and may be tentatively omitted. This book is a first step towards formulating AFTT.

In this book, code is written in Scala because the author is fluent in that language. However, most of this material will work equally well in Haskell, OCaml, and other FP languages. This is so because the science of functional programming, call ed AFTT, is not a set of tricks specific to Scala or Haskell. An advanced user of any other functional programming language will have to face the same questions and struggle with the same practical issues.

18.1 AFTT is not covered by computer science curricula

Traditional courses of theoretical computer science (algorithms and data structures, complexity theory, distributed systems, databases, network systems, compilers, operating systems) are largely not relevant to AFTT.

Here is an example: To an academic computer scientist, the “science behind Haskell” is the theory of lambda-calculus, the type-theoretic “System $F\omega$ ”, and formal semantics. These theories guided the design of the Haskell language and define rigorously what a Haskell program “means” in a mathematical sense. Academic computer science courses teach these theories, although typically only at the graduate level.

However, a practicing Haskell or Scala programmer is not concerned with designing Haskell or Scala, or with proving any theoretical properties of those languages. A practicing programmer is mainly concerned with *using* a chosen programming language to *write code*.

Neither the theory of lambda-calculus, nor proofs of type-theoretical properties of “System $F\omega$ ”, nor theories of formal semantics will actually help a programmer to write code. So all these theories are not within the scope of AFTT. It appears that functional programming does not require graduate-level theoretical studies.

As an example of theoretical material that *is* within the scope of AFTT, consider the equational laws imposed on applicative functors (see Chapter 10). It is essential for a practicing functional programmer to be able to recognize and use applicative functors. An applicative functor is a data structure specifying declaratively a set of operations that run independently of each other. Programs can then easily combine these operations, for example, in order to execute them in parallel, or to refactor the program for better maintainability.

¹https://wiki.haskell.org/Research_papers/Functional_pearls

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, a data structure may be dictated in part by the business logic rather than by a programmer's choice. The programmer first writes down the type of that data structure and the code implementing the required methods, and then checks that the laws hold. The data structure may need to be adjusted in order to fit the definition of an applicative functor or its laws.

This work is done using pen and paper, in a mathematical notation. Once the applicative laws are verified, the programmer proceeds to write code using that data structure.

Because of the mathematical proofs, it is assured that the data structure satisfies the known properties of applicative functors, no matter how the rest of the program is written. So, for example, it is assured that the relevant effects can be automatically parallelized and will still work correctly. In this way, AFTT directly guides the programmer and helps to write correct code.

Applicative functors were discovered by practitioners who were using Haskell for writing code, in applications such as parser combinators, compilers, and domain-specific languages for parallel computations. However, applicative functors are not a feature of Haskell: they are the same in Scala, OCaml, or any other functional programming language. And yet, no standard computer science textbook defines applicative functors, motivates their laws, explores their structure on basic examples, or shows data structures that are *not* applicative functors and explains why. (Books on category theory and type theory also do not mention applicative functors.)

18.2 AFTT is not category theory, type theory, or formal logic

So far it appears that AFTT includes a selection of certain areas of category theory, formal logic, and type theory. However, software engineers would not derive much benefit from following traditional academic courses in these subjects, because their presentation is too abstract and at the same time lacks specific results necessary for practical programming. In other words, the traditional academic courses answer questions that academic computer scientists have, not questions that software engineers have.

There exist several books intended as presentations of category theory "for computer scientists" or "for programmers". However, these books do not explain certain concepts relevant to programming, such as applicative or traversable functors. Instead, these books contain purely theoretical topics such as limits, adjunctions, or toposes, – concepts that have no applications in practical functional programming today.

Typical questions in these books are "Is X an introduction rule or an elimination rule" and "Does the property Y hold in non-small categories, or only in the category of sets". Questions a Scala programmer might have 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 foundations of functional programming will not find the concepts of filterable, applicative, or traversable functors in any books on category theory, including books intended for programmers. And yet, these concepts are necessary to obtain a mathematically correct implementation of such foundationally important operations as `filter`, `zip`, and `traverse` – operations that functional programmers often use in their code.

To compensate for the lack of AFTT textbooks, programmers have written many online tutorials for each other, trying to explain the theoretical concepts necessary for practical work. There are the infamous "monad tutorials", but also tutorials about applicative functors, traversable functors, free monads, and so on. These tutorials tend to be hands-on ("run this code now and see what happens") and narrow in scope, limited to one or two specific questions and specific applications. Such tutorials usually do not present sufficient mathematical insights to help programmers develop the necessary mathematical intuition.

For example, "free monads" became popular in the Scala community around 2015. Many talks about free monads were presented at Scala engineering conferences, each giving their own slightly

different implementation but never formulating rigorously the required properties for a piece of code to be a valid implementation of the free monad.

Without knowledge of mathematical principles behind free monads, a programmer cannot make sure that a given implementation is correct. However, books on category theory present free monads in a way that is unsuitable for programming applications: a free monad is just an adjoint functor to a forgetful functor into the category of sets. (“What’s your problem?” as the joke would go.) This definition is too abstract and, for instance, cannot be used to check whether a given implementation of the free monad in Scala is correct.

Perhaps the best selection of AFTT tutorial material can be found in the Haskell Wikibooks.² 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 proofs or derivations of equational laws.

Apart from referring to some notions from category theory, AFTT also uses some concepts from type theory and formal logic. However, existing textbooks on type theory and formal logic focus on domain theory and proof theory – which is a lot of information that practicing programmers will have difficulty assimilating and yet will have no chance of ever applying in their daily work. At the same time, these books never mention practical techniques used in many functional programming libraries today, such as quantified types, types parameterized by type constructors, or partial type-level functions (known as “typeclasses”).

Theory can help the programmer with certain practical tasks, such as:

- 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)]`.
- Checking whether a definition of a recursive type is “reasonable”, i.e. does not lead to a useless infinite recursion. An example of a useless recursive type definition is `case class Bad(x: Bad)`.
- Deciding whether a function with a given type signature can be implemented. For example, `def f[A, B]: (A => B) => A` cannot be implemented but `def g[A, B]: A => (B => A)` can be.
- Deriving an implementation of a function from its type signature and checking the required laws; for example, deriving the `flatMap` method for the `Reader` monad from the type signature

```
def flatMap[Z, A, B](r: Z => A)(f: A => Z => B): Z => B
```

and verifying that the monad laws hold.

These practical tasks are actual real-world-coding applications of domain theory and the Curry-Howard correspondence. However, existing books on type theory and logic do not give practical recipes for performing these tasks.

On the other hand, books such as *Scala with Cats*³ and *Functional programming, simplified*⁴ are focused on explaining the practical aspects of programming and do not adequately treat the equational laws that the mathematical structures require (such as the laws for applicative or monadic functors).

The only existing Scala-based AFTT textbook aiming at the proper scope is *Functional Programming in Scala*⁵, which balances practical considerations with theoretical developments such as equational laws. The present book, *Science of Functional Programming*⁶, is written at about the same level but goes deeper into the mathematical foundations and at the same time gives a wider range of examples. It is an attempt to delineate the proper scope of AFTT and to develop a rigorous yet clear and approachable presentation of the chosen material.

²<https://en.wikibooks.org/wiki/Haskell>

³<https://underscore.io/books/scala-with-cats/>

⁴<https://alvinalexander.com/scala/functional-programming-simplified-book>

⁵<https://www.manning.com/books/functional-programming-in-scala>

⁶<http://www.lulu.com/content/paperback-book/the-science-of-functional-programming/24915714>

19 Essay: Software engineers and software artisans

Let us look at the differences between the kind of activities we ordinarily call engineering, as opposed to artisanship or craftsmanship. It will then become apparent that today's computer programmers are better understood as "software artisans" rather than software engineers.

19.1 Engineering disciplines

Consider what kinds of process a mechanical engineer, a chemical engineer, or an electrical engineer follows in their work, and what kind of studies they require for proficiency in their work.

A mechanical engineer studies¹ calculus, linear algebra, differential geometry, and several areas of physics such as theoretical mechanics, thermodynamics, and elasticity theory, and then uses calculations to guide the design of a bridge, say. A chemical engineer studies² chemistry, thermodynamics, calculus, linear algebra, differential equations, some areas of physics such as thermodynamics and kinetic theory, and uses calculations to guide the design of a chemical process, say. An electrical engineer studies³ advanced calculus, linear algebra, and several areas of physics such as electrodynamics and quantum theory, and uses calculations to design an antenna or a microchip.

The pattern here is that an engineer uses mathematics and natural sciences in order to design new devices. Mathematical calculations and scientific reasoning are required *before* drawing a design, let alone building a real device or machine.

Some of the studies required for engineers include arcane abstract concepts such as a "rank-4 elasticity tensor"⁴ (used in calculations of elasticity of materials), "Lagrangian with non-holonomic constraints"⁵ (used in robotics), the "Gibbs free energy" (for chemical reactor design⁶), or the "Fourier transform of the delta function"⁷ and the "inverse Z-transform"⁸ (for digital signal processing).

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.

19.2 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.

A novice shoemaker, for example, begins by copying some drawings⁹ and goes on to cutting leather in a home workshop. Apprenticeships proceed via learning by doing, with comments and

¹<https://www.colorado.edu/mechanical/undergraduate-students/curriculum>

²<https://www.colorado.edu/engineering/sample-undergraduate-curriculum-chemical>

³<http://archive.is/XYLyE>

⁴https://serc.carleton.edu/NAGTWorkshops/mineralogy/mineral_physics/tensors.html

⁵<https://arxiv.org/abs/math/0008147>

⁶<https://www.amazon.com/Introduction-Chemical-Engineering-Kinetics-Reactor/dp/1118368258>

⁷<https://www.youtube.com/watch?v=KAbqISZ6SHQ>

⁸<http://archive.is/SsJqP>

⁹<https://youtu.be/cY5MY0czMAk?t=141>

instructions from an expert. After a few years of apprenticeship (for example, a painter apprenticeship in California¹⁰ can be as short as 2 years), a new specialist is ready to start productive work.

All these trades operate entirely from tradition and practical experience. The trades do not require academic study because there is no formal theory from which to proceed. Of course, there is *a lot* to learn in the crafts, and it takes prolonged effort to become a good artisan in any profession. But there are no rank-4 tensors to calculate, nor any differential equations to solve; no Fourier transforms to apply to delta functions, and no Lagrangians to check for non-holonomic constraints.

Artisans do not study any formal science or mathematics because their professions do not make use of any *formal computation* for guiding their designs or processes.

19.3 Programmers today are artisans, not engineers

Programmers are *not engineers* in the sense we normally see the engineering professions.

19.3.1 No requirement of formal study

According to this recent Stack Overflow survey¹¹, about half of the programmers do not have a degree in Computer Science. The author of this book is a self-taught programmer who has degrees in physics but never formally studied computer science or taken any academic courses in algorithms, data structures, computer networks, compilers, programming languages, or other 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. Robert C. Martin¹² is a prominent example; an outspoken guru in the arts of programming who has seen it all, he routinely refers to programmers as artisans¹³ and uses the appropriate imagery: novices, 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 posts¹⁴, he started working at age 17 as a self-taught programmer, and then went on to more jobs in the software industry; he never mentions going to college. It is clear that R. C. Martin is an expert craftsman, and that he did *not* need academic study to master his craft.

Here is another opinion¹⁵ (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”¹⁶ – vocational schools that educate new programmers in a few months through purely practical training, with no formal theory or mathematics involved. These vocational schools are successful¹⁷ in job placement. But it is unimaginable that a 6-month crash course or even a 2-year vocational school could prepare an engineer to work successfully on say, quantum computers¹⁸ without ever having studied quantum physics or calculus.

¹⁰http://www.calapprenticeship.org/programs/painter_apprenticeship.php

¹¹<https://thenextweb.com/insider/2016/04/23/dont-need-go-college-anymore-programmer/>

¹²https://en.wikipedia.org/wiki/Robert_C._Martin

¹³<https://blog.cleancoder.com/uncle-bob/2013/02/01/The-Humble-Craftsman.html>

¹⁴<https://blog.cleancoder.com/uncle-bob/2013/11/25/Novices-Coda.html>

¹⁵<http://archive.is/tAKQ3>

¹⁶<http://archive.is/Gk0L9>

¹⁷<http://archive.is/E9FXP>

¹⁸<https://www.dwavesys.com/quantum-computing>

19.3.2 No mathematical formalism guides software development

Most books on software engineering contain no formulas or equations, no mathematical derivations of any results, and no precise definitions of the various technical terms they are using (such as “object-oriented” or “software architecture”). Some of those books¹⁹ have almost no program code in them; instead they are filled with words and illustrative diagrams. These books talk about how programmers should approach their job, how to organize the work flow and the code architecture, etc., in vague and general terms: “code is about detail”, “you must never abandon the big picture”, “you should avoid tight coupling in your modules”, “a class must serve a single responsibility”, and so on. Practitioners such as R. C. Martin never studied any formalisms and do not think in terms of formalisms; instead they think in vaguely formulated, heuristic “principles”.²⁰

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 equations²¹ coming from electrodynamics.

Donald Knuth’s classic textbook is called “*The Art of Programming*”. It is full of tips and tricks about how to program; but it does not provide any formal theory that could guide programmers in actually *writing* programs. There is nothing in that book that would be similar to the way mathematical formalism guides designs in electrical or mechanical engineering. If Knuth’s books were based on such formalism, they would have looked quite differently: some theory would be first explained and then applied to help us write code.

Knuth’s books provide many rigorously derived algorithms. But algorithms are similar to patented inventions: they can be used immediately without further study. Understanding an algorithm is not similar to understanding a mathematical theory. Knowing one algorithm does not make it easier to develop another algorithm in an unrelated domain. In comparison, knowing how to solve differential equations will be applicable to thousands of different areas of science and engineering.

A book exists²² with the title “Science of Programming”, but the title is misleading. The author does not propose a science, similar to physics, at the foundation of the process of designing programs, similarly to how calculations in quantum physics predict the properties of a quantum device. The book claims to give precise methods that guide programmers in writing code, but the scope of proposed methods is narrow: the design of simple algorithms for iterative manipulation of data. The procedure suggested in that book is far from a formal mathematical *derivation* of programs from specifications. (A book with that title²³ also exists, and similarly disappoints.) In any case, programmers today are oblivious to these books and do not use the methods explained there.

Standard computer science courses today do not teach a true *engineering* approach to software construction. They do teach analysis of programs using formal mathematical methods; the main such methods are complexity analysis²⁴ (the “big-*O* notation”) and formal verification²⁵. But programs are analyzed or verified only *after* they are complete. Theory does not guide the actual *process* of writing code, does not suggest good ways of organizing the code (e.g. choosing which classes or functions or modules should be defined), and does not tell programmers which data structures or APIs would be best to implement. Programmers make these design decisions purely on the basis of experience and intuition, trial-and-error, copy-paste, and guesswork.

The theory of program analysis and verification is somewhat analogous to writing a mathematical equation for the surface of a shoe made by a fashion designer. True, the “shoe surface equations” are mathematically rigorous and can be “analyzed” or “verified”; but the equations are written after the fact and do not guide the fashion designers in actually making shoes. It is understandable that fashion designers do not study the mathematical theory of surfaces.

¹⁹E.g. <https://www.amazon.com/Object-Oriented-Software-Engineering-Unified-Methodology/dp/0073376256>

²⁰<https://blog.cleancoder.com/uncle-bob/2016/03/19/GivingUpOnTDD.html>

²¹<https://youtu.be/8KpfVsJ5Jw4?t=447>

²²<https://www.amazon.com/Science-Programming-Monographs-Computer/dp/0387964800>

²³<https://www.amazon.com/Program-Derivation-Development-Specifications-International/dp/0201416247>

²⁴<https://www.cs.cmu.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html>

²⁵https://en.wikipedia.org/wiki/Formal_verification

19.3.3 Programmers avoid academic terminology

Programmers appear to dislike terminology such as “functor”, “monad”, or “lambda-functions”:

Those fancy words used by functional programmers purists really annoy me. Monads, functors... Nonsense!!!²⁶

Perhaps only a small minority of software engineers actually complain about this; the vast majority remain unaware of “functors” or “monads”.

However, chemical engineers accept the need for studying differential equations and do not wince at the terms “phase diagram” or “Gibbs free energy”. Electrical engineers do not complain that the word “Fourier” is difficult to spell, or that “delta-function” is a weird thing to say. Mechanical engineers take it for granted that they need to calculate with “tensors” and “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.

Similarly, software engineers would not complain about the word “functor”, or about having to study the derivation of the algebraic laws for “monads,” – if they were actually *engineers*. True software engineers’ textbooks would be full of equations and derivations, which would be used to perform calculations required *before* starting to write code.

19.4 Towards software engineering

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.

One could point out that numerical simulations required for physics or the matrix calculations required for machine learning are “mathematical”. True, these programming *tasks* are mathematical in nature and require formal theory to be *formulated*. However, mathematical *subject matter* (aerospace control, physics or astronomy experiments, mathematical statistics, etc.) does not mean that the process of programming is a form of engineering. Data scientists, aerospace engineers, and natural scientists all write code nowadays – and they are all working as artisans when they write code.

True software engineering means having a theory that guides and informs the process of creating programs, – not theory that describes or analyzes programs after they are *somewhat* written.

We expect that software engineers’ textbooks should be full of equations. What theory should those equations represent?

I believe this theory already exists, and I call it **applied functional type theory**. It is the algebraic foundation of the modern practice of functional programming, as implemented in languages such as OCaml, Haskell, and Scala. This theory is a blend of type theory, category theory, and logical proof theory. It has been in development since late 1990s and is still being actively worked on by a community of academic computer scientists and advanced software practitioners.

To appreciate that functional programming, unlike any other programming paradigm, *has a theory that guides coding*, we can look at some recent software engineering conferences such as “Scala By the Bay”²⁷ or BayHac²⁸, or at the numerous FP-related online tutorials and blogs. We cannot fail to notice that much time is devoted not to showing code but to a peculiar kind of 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”²⁹ or the “free monad”³⁰ – and illustrates its use for practical coding.

²⁶<http://archive.is/65K3D>

²⁷<http://2015.scala/bythebay.io/>

²⁸<http://bayhac.org/>

²⁹<http://www.youtube.com/watch?v=bmIxIslimVY>

³⁰<http://www.youtube.com/watch?v=U01K0hnbc4U>

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 – coming from formal logic and category theory – are now 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. 19.1). 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.

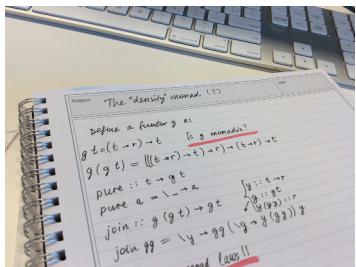


Figure 19.1: A programmer performs a derivation in Haskell before writing code.

programs. The new technique has distinct advantages over using monad transformers, which was the previous method of composing declarative side-effects.

The “free applicative / free monad” combination was designed and implemented by true software engineers. They first wrote down the types and derived the necessary algebraic properties; the obtained results directly guided them about how to proceed writing the library API.

Another example of a development in functional type theory is the “tagless final” encoding of data types, [first described in 2009](#). This technique, developed from category theory and type theory motivations, has several advantages over the free monad technique and can improve upon it in a number of cases – just as the free monad itself was designed to cure certain problems with monad transformers³⁴. The new technique is also not a trick in a specific programming language; rather, it is a theoretical development that is available to programmers in any language (even in Java³⁵).

This example shows that we may need several more years of work before the practical aspects of using “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 hypothesis³⁶ holds, we should expect 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.

A recent example of the hand-in-hand development of the functional type theory and its applications is seen in the “free applicative functor” construction. It was first described in a 2014 paper³¹; a couple of years later, a combined free applicative / free monad data type was designed and its implementation proposed³² as well as in Haskell³³. 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

³¹<https://arxiv.org/pdf/1403.0749.pdf>

³²<https://github.com/typelevel/cats/issues/983>

³³<https://elvishjerricco.github.io/2016/04/08/applicative-effects-in-free-monads.html>

³⁴<http://blog.ezyang.com/2013/09/if-youre-using-lift-youre-doing-it-wrong-probably/>

³⁵<http://archive.is/rLAh9>

³⁶<http://archive.is/rJc4A>

19.5 Does software need engineers, or are artisans good enough?

The demand for programmers is growing. “Software developer” was #1 best job³⁷ in the US in 2018. But is there a demand for engineers or just for artisans?

We do not seem to be able³⁸ to train enough software artisans. 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; at best, they train academic researchers who write code as software artisans. Recalling the situation in construction business, with a few architects and hundreds of construction workers, we might also conclude that, perhaps, only a few software engineers are required per hundred software artisans.

What is the price of *not* having engineers, of replacing them with artisans?

Software practitioners have long bemoaned the mysterious difficulty of software development. Code “rots with time”, its complexity grows “out of control”, and operating systems have been notorious for ever-appearing security flaws³⁹ despite many thousands of programmers and testers employed. Clearly, we overestimated the capacity of the human brain for artisanal programming.

It is precisely in designing large and robust software systems that we would benefit from true engineering. Humanity has been building bridges and using chemical reactions long before mechanical or chemical engineering disciplines were developed and founded upon rigorous theory. Once the theory became available, humanity proceeded to create unimaginably more complicated and powerful structures and devices than ever before. It is clear that trial, error, and adherence to tradition is inadequate for the 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. Functional type theory and functional programming are first steps in that direction.

³⁷<http://archive.is/cGJ2T>

³⁸<http://archive.is/137b8>

³⁹<http://archive.fo/HtQzw>

20 Essay: Towards functional data engineering with Scala

Data engineering is among the most in-demand¹ 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 tool² for crafting the foundational data engineering technologies such as Spark or Akka? Why is Scala in such demand³ within the world of big data?

There are reasons to believe that the choice of Scala was not accidental.

20.1 Data is math

Humanity has been working with data at least since Babylonian tax tables⁴ and the ancient Chinese number books⁵. 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, distances, prices, dates, times, – needs to be handled separately; e.g. 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. 20.1 for a real-life illustration).

The power of the principles of mathematics extends over all epochs and all cultures; math is the same in Rio de Janeiro, in Kuala-Lumpur, and in Pyongyang (Fig. 20.2).

20.2 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 system 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



Figure 20.1: Mixing incompatible data types gives nonsense results.

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<http://quatr.us/china/science/chinamath.htm>

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 the advantage of several millennia of mathematical experience.

A prominent example of how mathematics informs the design of programming languages is the connection between constructive logic⁶ and the programming language's type system, called the Curry-Howard (CH) correspondence⁷. 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** to the logic if for any two 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 are faithful to the CH correspondence. Having a *complete* logic in a language's type system enables declarative domain-driven code design⁸.



Figure 20.2: The Pyongyang method of error-free programming.

rent, distributed, and resilient to failure. These 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 functional 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 reap the benefits of the powerful abstractions Scala supports. In this way, Scala enabled those businesses to engineer reliably and to scale up their massively concurrent computations.

⁶https://en.wikipedia.org/wiki/Intuitionistic_logic

⁷https://en.wikipedia.org/wiki/Curry%2FHoward_correspondence

⁸<https://fsharpforfunandprofit.com/ddd/>

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 20.2).

20.3 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. These software companies used Scala as their main im-

It remains to see why Scala and not, say, Haskell became the *lingua franca* of big data.

20.4 Scala is Java on math

The recently invented general-purpose functional programming languages can be grouped into “industrial” (F#, Scala, Swift) and “academic” (OCaml, Haskell).

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 unencumbered by requirements of compatibility with any existing platform or libraries. Because of this, the “academic” languages are perfect playgrounds for taking various mathematical ideas to their logical conclusion. At the same time, software practitioners struggle to adopt these 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 necessitated 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 higher⁹ than that of the “academic” languages. The transition to the functional paradigm is also made 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 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, Squerly, or Quill. Similar libraries exist for Haskell – but not in languages with less powerful type systems.

20.5 Summary

The decisive advantages of Scala over other contenders (such as OCaml, Haskell, F#, or Swift) are:

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 great future as a main implementation language for big data engineering.

⁹<https://www.tiobe.com/tiobe-index/>, archived at <http://archive.is/RsNH8>

Part V

Appendices

A Notations

Certain notations and terms were chosen in this book 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 equational laws of typeclasses.

A.1 Summary of notations for types and code

F^A type constructor F with type argument A . In Scala, `F[A]`

$x:A$ value x has type A ; in Scala, `x:A`

$\mathbb{1}, 1$ the unit type and its value; in Scala, `Unit` and `()`

$\mathbb{0}$ the void type. In Scala, `Nothing`

$A + B$ a disjunctive type. 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)`

$A \Rightarrow B$ the function type, mapping from A to B

$x:A \Rightarrow f$ a nameless function (as a value). In Scala, `{ x:A => f }`

id the identity function; in Scala, `identity[A]`

\triangleq “equal by definition”

\cong for types, a natural isomorphism between types; for values, “equivalent” values according to an established isomorphism

A^{F^B} 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”

fmap $_F$ the standard method `fmap` pertaining to a functor F . In Scala, `Functor[F].fmap`

flatMap $_F$ the standard method `flatMap` pertaining to a monad F . In Scala, `Monad[F].flatMap`

flatten $_F$ the standard method `flatten` pertaining to a monad F . In Scala, `Monad[F].flatten`

pure $_F$ the standard method `pure` of a monad F . In Scala, `Monad[F].pure`

F^\bullet the type constructor F understood as a type-level function. In Scala, `F[_]`

$F^\bullet \rightsquigarrow G^\bullet$ or $F^A \rightsquigarrow G^A$ a natural transformation between functors F and G . In Scala, `F ~> G`

A Notations

$\forall A.P^A$ 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] }`

\circ the forward composition of functions: $f \circ g$ is $x \Rightarrow g(f(x))$. In Scala, `f andThen g`

\circ the backward composition of functions: $f \circ g$ is $x \Rightarrow f(g(x))$. In Scala, `f compose g`

\circ the backward composition of type constructors: $F \circ G$ is F^G

\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 raised to a functor G ; same as `fmap_G f`

$f^{\uparrow G \uparrow H}$ a function raised first to G and then to H . In Scala, `h.map(_.map(f))`

$f^{\downarrow H}$ a function f raised to a contrafunctor

\diamond_M the Kleisli product operation for the monad M

\oplus the binary operation of a monoid. In Scala, `x |+| y`

Δ the “diagonal” function of type $\forall A.A \Rightarrow A \times A$

$\nabla_1, \nabla_2, \dots$ the projections from a tuple to its first, second, ..., parts

\boxtimes pair product of functions, $(f \boxtimes g)(a \times b) = f(a) \times g(b)$

$[a, b, c]$ an ordered sequence of values. In Scala, `Seq(a, b, c)`

$\begin{array}{cc} x \Rightarrow x & 0 \\ 0 & a \Rightarrow a \times a \end{array}$	a function that works with disjunctive types
---	--

A.2 Detailed explanations

F^A means a type constructor F with a type parameter A . In Scala, this is `F[A]`. Type constructors with multiple type parameters are denoted by $F^{A,B,C}$.

$x^{:A}$ 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). The notation $x : A$ can be used as well, but $x^{:A}$ is easier to read when x is inside a larger code expression.

$\mathbb{1}$ means the unit type, and 1 means the value of the unit type. In Scala, the unit type is `Unit`, and its value is `()`. Example of using the unit type 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`. Example of using the void type is to denote the empty part of a disjunction. For example, in the disjunction $\mathbb{1} + A$ the non-empty part is $\mathbb{0} + A$, which in Scala corresponds to `Some[A]`. The empty part $\mathbb{1} + \mathbb{0}$ corresponds to `None`. Similarly, $A + \mathbb{0}$ denotes the left part of the type $A + B$ (in Scala, `Left[A]`), while $\mathbb{0} + B$ denotes its right part (in Scala, `Right[B]`). Values of disjunctive types are denoted similarly. For instance, $x^{:A} + \mathbb{0}^{:B}$ denotes a value of the left part of the type $A + B$; in Scala, this value is written as `Left[A,B](x)`.

$A + B$ means the disjunctive type made from types A and B (or, a disjunction of 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.

$A \Rightarrow B$ means a function type from A to B . In Scala, this is the function type `A => B`.

$x^A \Rightarrow y$ 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 $\text{id}^{A \Rightarrow A}$, or else should be unambiguous from the context. In Scala, `identity[A]` corresponds to id^A .

\triangleq means “equal by definition”. Examples:

- $f \triangleq (x^{\text{Int}} \Rightarrow x + 10)$ is a definition of a function f . In Scala, this is `val f = { x: Int => x + 10 }`.
- $F^A \triangleq \mathbb{1} + A$ is a definition of a type constructor F . 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 expressions means that the type constructor F^\bullet 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, which is denoted by $F^A \triangleq \mathbb{1}^{F^{\text{Int}}} + A^{F^{\text{A} \times \text{String}}}$.

fmap_F means the standard method `fmap` of the `Functor` typeclass, implemented for the functor F . 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 method fmap_F actually has *two* type parameters, which can be written out as $\text{fmap}_F^{A,B}$. Then the type signature of `fmap` is written in full as $\text{fmap}_F^{A,B} : (A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$. For clarity, we may sometimes write out the type parameters A, B in the expression $\text{fmap}_F^{A,B}$, but in most cases these 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 $A \Rightarrow F^A$ that contains 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 the `pure` method with a complicated type, e.g. $\mathbb{1} + P^A$, instead of the type parameter A , we might want to write this type parameter for clarity and write $\text{pu}_F^{\mathbb{1} + P^A}$. The type signature of that function is then

$$\text{pu}_F^{\mathbb{1} + P^A} : \mathbb{1} + P^A \Rightarrow F^{\mathbb{1} + P^A} .$$

But in most cases we will not need to write out the type parameters.

flm_F denotes a monad F ’s method `flatMap` with the type signature

$$\text{flm}_F : (A \Rightarrow F^B) \Rightarrow F^A \Rightarrow F^B .$$

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’s laws.

ftn_F denotes a monad F ’s method `flatten` with the type signature

$$\text{ftn}_F : F^{F^A} \Rightarrow F^A .$$

F^\bullet means the type constructor F understood as a type-level function, – that is, with a type argument unspecified. In Scala, this is `F[_]`. The bullet symbol, \bullet , is used as a placeholder for the missing type parameter. I also simply write F when no type argument is needed, and it 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 argument would appear. For instance, functor composition is denoted as F^{C^\bullet} ; in Scala, this is `F[G[?]]` when using the “kind projector” plugin.¹ As 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 $T_L^M{}^\bullet$, and so is the missing type parameter denoted by the placeholder symbol \bullet . (However, the base monad L is not a type parameter in $T_L^M{}^\bullet$ because the construction of the monad transformer depends sensitively on the internal details of L .)

$F^\bullet \rightsquigarrow G^\bullet$ or $F^A \rightsquigarrow G^A$ 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.

\circ means the forward composition of functions: $f \circ g$ (reads “ f before g ”) is the function defined as $x \Rightarrow g(f(x))$.

\circ means the backward composition of functions: $f \circ g$ (reads “ f after g ”) is the function defined as $x \Rightarrow f(g(x))$.

\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]]`.

$x \triangleright f$ means that x is inserted as the argument into the function f . This **pipe notation**, $x \triangleright f$, means the same expression as $f(x)$ or $f\ x$. 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 function f is to the *right* of x in this notation, forward compositions of functions such as $x \triangleright f \triangleright g$ are naturally grouped to the left, similarly to how this is done in Scala code, for example `x.toSeq.sorted`. The operation \triangleright (pronounced “pipe”) groups weaker than the forward composition (\circ), and so we have $x \triangleright f \circ g = x \triangleright f \triangleright g$ in this notation. Reasoning about code in the forwarding notation uses the identities

$$\begin{aligned} x \triangleright f &= f(x), & (x \triangleright f) \triangleright g &= x \triangleright f \triangleright g , \\ x \triangleright f \circ g &= x \triangleright (f \circ g), & x \triangleright f \triangleright g &= x \triangleright f \circ g . \end{aligned}$$

Some examples of reasoning in the pipe notation:

$$\begin{aligned} (a \Rightarrow a \triangleright f) &= (a \Rightarrow f(a)) = f , \\ f \triangleright (y \Rightarrow a \triangleright y) &= a \triangleright f = f(a) , \\ f(y(x)) &= x \triangleright y \triangleright f \neq x \triangleright (y \triangleright f) = f(y)(x) . \end{aligned}$$

The correspondence between the forward composition and the backward composition:

$$\begin{aligned} f \circ g &= g \circ f , \\ x \triangleright (f \circ g) &= x \triangleright f \circ g = x \triangleright f \triangleright g = g(f(x)) = (g \circ f)(x) . \end{aligned}$$

$f^{\uparrow G}$ means a function f lifted to a functor G . For a function $f^{A \Rightarrow B}$, 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 $(f^{\uparrow G})^{\uparrow H}$ and produces a function of type $H^{G^A} \Rightarrow H^{G^B}$. Applying a nested lifting to a value h of type H^{G^A} is written as $f^{\uparrow G \uparrow H} h$ or $h \triangleright f^{\uparrow G \uparrow H}$. In Scala, this is `h.map(_.map(f))`. The functor composition law is written as

$$p^{\uparrow G} \circ q^{\uparrow G} = (p \circ q)^{\uparrow G} .$$

Note the similarity between Scala code `x.map(p).map(q)` and the notation $x \triangleright p^{\uparrow G} \triangleright q^{\uparrow G}$.

¹<https://github.com/typelevel/kind-projector>

$f^{\downarrow H}$ means a function f lifted to a contrafunctor H . For a function $f: A \Rightarrow B$, the application of $f^{\downarrow H}$ to a value $h : H^B$ is written as $f^{\downarrow H} h$ or $h \triangleright f^{\downarrow H}$, and yields a value of type H^A . In Scala, this is `h.contramap(f)`. Nested lifting is denoted as $f^{\downarrow H \uparrow G} \triangleq (f^{\downarrow H})^{\uparrow G}$.

\diamond_M means the Kleisli product operation for a given monad M . This is a binary operation working on two Kleisli functions of types $A \Rightarrow M^B$ and $B \Rightarrow M^C$ and yields a new function of type $A \Rightarrow M^C$.

\oplus means the binary operation of a monoid, for example $x \oplus y$. The specific monoid type should be defined for this expression to make sense. For example, in Scala the monoidal operation is usually denoted by `x |+| y`.

Δ means the “diagonal” function of type $\forall A. A \Rightarrow A \times A$. There is only one implementation of this type signature,

```
def delta[A](a: A): (A, A) = (a, a)
```

$\nabla_1, \nabla_2, \dots$ denote the standard projection functions from a tuple to its first, second, ..., parts. In Scala, ∇_1 is `_._1`.

\boxtimes means the “pair product” of functions, where the result is a pair of the values of the two functions: $(f \boxtimes g)(a \times b) = f(a) \times g(b)$. In Scala, this operation can be defined by

```
def boxtimes[A,B,P,Q](f: A => P, g: B => Q): ((A, B)) => (P, Q) = {
  case (a, b) => (f(a), g(b))
}
```

The operations Δ, ∇_i (where $i = 1, 2, \dots$), and \boxtimes allow us to express any function operating on tuples. Useful properties for reasoning about code of such functions:

identity law : $\Delta ; \nabla_i = \text{id}$,

duplication law : $f ; \Delta = \Delta ; (f \boxtimes f)$,

projection law : $(f \boxtimes g) ; \nabla_1 = \nabla_1 ; f$,

projection law : $(f \boxtimes g) ; \nabla_2 = \nabla_2 ; g$,

composition law : $(f \boxtimes g) ; (p \boxtimes q) = (f ; p) \boxtimes (g ; q)$,

as well as the naturality laws for Δ and ∇_i :

$$\begin{aligned} f^{\uparrow F} ; \Delta &= \Delta ; f^{\uparrow (F \times F)} = \Delta ; (f^{\uparrow F} \boxtimes f^{\uparrow F}) \quad , \\ (f^{\uparrow F} \boxtimes f^{\uparrow G}) ; \nabla_1 &= f^{\uparrow (F \times G)} ; \nabla_1 = \nabla_1 ; f^{\uparrow F} \quad . \end{aligned}$$

$[a, b, c]$ 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.

$f: Z + A \Rightarrow Z + A \times A \triangleq \left\| \begin{array}{cc} z \Rightarrow z & 0 \\ 0 & a \Rightarrow a \times a \end{array} \right\|$ is the “matrix notation” for a function whose input and/or output type is a disjunctive type. In Scala, the function f is implemented 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.
}
```

At the initial stages of reasoning The rows of the matrix indicate the different `cases` 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 the output type is not disjunctive, there will be only one column.

An “annotated matrix” writes out all parts of the disjunctive types in a separate “type row” and “type column”:

$$f \triangleq \left\| \begin{array}{c|cc} & Z & A \times A \\ \hline Z & \text{id} & 0 \\ A & 0 & a \Rightarrow a \times a \end{array} \right\| \quad . \quad (\text{A.1})$$

A Notations

This notation clearly indicates the input and the output types of the function and is useful at the initial stages of reasoning about the code. The vertical double line separates input types from the function code. In the code above, the “type column” shows the parts of the input disjunction type $Z + A$. The “type row” shows the parts of the output disjunction type $Z + A \times A$.

The matrix notation is adapted to *forward* function compositions ($f ; g$). 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

$$g \triangleq \left\| \begin{array}{c|c} & A \\ \hline Z & _ \Rightarrow e^{:A} \\ A \times A & a_1 \times a_2 \Rightarrow a_1 \oplus a_2 \end{array} \right\| , \quad g \triangleq \left\| \begin{array}{c} _ \Rightarrow e^{:A} \\ a_1 \times a_2 \Rightarrow a_1 \oplus a_2 \end{array} \right\| .$$

The forward composition $f ; g$ is computed by forward-composing the matrix elements using the rules of the ordinary matrix multiplication, omitting any terms containing \emptyset :

$$\begin{aligned} f ; g &= \left\| \begin{array}{cc} \text{id} & \emptyset \\ \emptyset & a \Rightarrow a \times a \end{array} \right\| \circ \left\| \begin{array}{c} _ \Rightarrow e^{:A} \\ a_1 \times a_2 \Rightarrow a_1 \oplus a_2 \end{array} \right\| \\ &= \left\| \begin{array}{c} \text{id} ; (_ \Rightarrow e^{:A}) \\ (a \Rightarrow a \times a) ; (a_1 \times a_2 \Rightarrow a_1 \oplus a_2) \end{array} \right\| = \left\| \begin{array}{c} _ \Rightarrow e^{:A} \\ a \Rightarrow a \oplus a \end{array} \right\| . \end{aligned}$$

Applying a function to a value of a disjunctive type such as $x : Z + A$ is computed by writing x as a single-row matrix, for example

$$x = z^{:Z} + \emptyset^{:A} = \left\| \begin{array}{c} z^{:Z} \\ \emptyset \end{array} \right\| ,$$

and the computation $x \triangleright f ; g$ again follows the rules of matrix multiplication:

$$x \triangleright f ; g = \left\| \begin{array}{c} z^{:Z} \\ \emptyset \end{array} \right\| \triangleright \left\| \begin{array}{c} _ \Rightarrow e^{:A} \\ a \Rightarrow a \oplus a \end{array} \right\| = z \triangleright (_ \Rightarrow e) = e .$$

Since the standard rules of matrix multiplication are associative, the properties of the \triangleright -notation such as $x \triangleright (f ; g) = (x \triangleright f) \triangleright g$ are guaranteed to hold.

To use the matrix notation with backward compositions ($f \circ g$), all function 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 argument types will then appear in the top row and the result types in the left column; the double line is above the matrix since that is where the function inputs come from. The above calculations are then rewritten as

$$\begin{aligned} g \circ f &= \left\| \begin{array}{c|cc} & Z & A \times A \\ \hline A & _ \Rightarrow e^{:A} & a_1 \times a_2 \Rightarrow a_1 \oplus a_2 \end{array} \right\| \circ \left\| \begin{array}{c|cc} & Z & A \\ \hline Z & \text{id} & \emptyset \\ A \times A & \emptyset & a \Rightarrow a \times a \end{array} \right\| \\ &= \boxed{\text{id} ; (_ \Rightarrow e^{:A}) \quad (a \Rightarrow a \times a) ; (a_1 \times a_2 \Rightarrow a_1 \oplus a_2)} = \boxed{_ \Rightarrow e^{:A} \quad a \Rightarrow a \oplus a} . \\ (g \circ f)(x) &= \boxed{_ \Rightarrow e^{:A} \quad a \Rightarrow a \oplus a} \boxed{\begin{array}{c} z^{:Z} \\ \emptyset \end{array}} = (_ \Rightarrow e^{:A})(z) = e . \end{aligned}$$

The forward composition seems to 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 $f:A \Rightarrow B$. Nameless functions are denoted by $x:A \Rightarrow f$, 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 $f;g$ (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 saying “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 “sum” is more ambiguous to the ear than “disjunctive”.

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 therefore be fixed throughout the entire functor block. (The type constructor *must* be a functor and may additionally be filterable and/or monadic.) For example, in Scala the 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 Also called a Kleisli morphism or a Kleisli arrow. A function with type signature $A \Rightarrow M^B$ for some fixed monad M . More verbosely, “a morphism from the Kleisli category corresponding to the monad M ”. The standard monadic method $\text{pure}_M : A \Rightarrow M^A$ has the type signature of a Kleisli function. The Kleisli product operation, \diamond_M , is a binary operation that combines two Kleisli functions (of types $A \Rightarrow M^B$ and $B \Rightarrow M^C$) into a new Kleisli function (of type $A \Rightarrow M^C$).

Method This word is used in two ways: 1) A method₁ is a Scala function defined as a member of a typeclass. For example, `flatMap` is a method defined in the `Monad` typeclass. 2) A method₂ is a Scala function defined as a member of a data type declared as a Java-compatible `class` or `trait`. Trait methods₂ 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 methods₁ require type parameters, will use Scala `traits` with methods₂ 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]` 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 `x`, and function types by `=>`. The unit type is denoted by `1`, and the void type by `0`. The function arrow `=>` groups weaker than `+`, which in turn groups weaker than `x`. This means

$$Z + A \Rightarrow Z + A \times A \quad \text{is the same as} \quad (Z + A) \Rightarrow (Z + (A \times A)) \quad .$$

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

$$F^A \triangleq A \times (A \Rightarrow 1 + \text{Int}) + (\text{String} \Rightarrow \text{List}^A) \quad .$$

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 $F^A \triangleq (A \times A)^{F^{\text{Int}}} + (\text{Int} \times A)^{F^1}$. 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 On the current misuse of the term “algebra”

This book avoids using the terms “algebra” or “algebraic” because these terms are too ambiguous. The functional programming community uses the word “algebra” in at least *four* incompatible ways.

Definition 0. In mathematics, an “algebra” is a vector space with multiplication and certain standard properties. For example, we need $1 * x = x$, the addition must be commutative, the multiplication must be distributive over addition, etc. The set of all 10×10 matrices with real coefficients is an “algebra” in this sense. These matrices form a 100-dimensional vector space, and they can be multiplied and added. This definition of “algebra” is not used in functional programming.

Definition 1. An “algebra” is a function with type signature $F^A \Rightarrow A$, where F^A is some fixed functor. This definition comes from category theory, where such types are called *F-algebras*. There is no direct connection between this “algebra” and Definition 0, except when the functor F is defined by $F^A \triangleq A \times A$, and then a function of type $A \times A \Rightarrow A$ may be interpreted as a “multiplication” operation (however, A is a type and not a vector space, and there are no distributivity or commutativity laws). It is better to call such functions “*F-algebras*”, emphasizing that they characterize and depend on a chosen functor F . However, knowing how to reason about the theoretical properties of *F-algebras* does not help in practical programming, and so they are not used in this book.

Definition 2. Polynomial functors are often called “algebraic data types”. However, they are not “algebraic” in the sense of Definitions 0 or 1. For example, consider the “algebraic data type” `Either[Option[A], Int]`, which is $F^A \triangleq 1 + A + \text{Int}$ in the type notation. The set of all values of the type F^A does not have the addition and multiplication operations required by the mathematical definition of “algebra”. The type F^A may admit some binary or unary operations (e.g. that of a monoid), but these operations will not be commutative or distributive. Also, there is not necessarily a function with type $F^A \Rightarrow A$, as required for Definition 1. Rather, the word “algebra” refers to “school-level algebra” with polynomials, since these data types are built from sums and products of types. If the data contains a function type, e.g. `Option[Int => A]`, the type is no longer polynomial. So this book uses more precise terms: “polynomial type” and “exponential-polynomial type”.

Definition 3. People talk about the “algebra” of properties of functions such as `map` or `flatMap`, meaning that these functions satisfy certain 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 “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 in another sense from “differential” or “integral” equations. But the laws in functional programming are always “algebraic”: they are equations with compositions and applications of functions. We find that it is not useful to refer to “algebraic” laws in any of these two senses. This book talks about “equational laws” or just “laws”.

Definition 4. The Church encoding of a free monad (also known as the “final tagless style of programming”) is the type expression $\forall E^\bullet. (S^{E^\bullet} \rightsquigarrow E^\bullet) \Rightarrow E^A$ that uses a higher-order type constructor S parameterized by a *type constructor* parameter E . The sub-expression $S^{E^\bullet} \rightsquigarrow E^\bullet$ can be viewed as an S -algebra in the category of type constructors (functors in a category of types without non-trivial morphisms). So, Definition 4 is related to Definition 1, with a specific choice of a category. However, the knowledge 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 a free monad.

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”.

So, it seems that the current usage of the word “algebra” in functional programming is both inconsistent and unhelpful to practitioners. In this book, the word “algebra” always denotes the branch of mathematics, as in “school-level algebra”. Instead of “algebra” as in Definitions 1 to 4, this book talks about “*F-algebras*” with a specific functor F ; “polynomial types” or “polynomial functors” or “exponential-polynomial functors” etc.; “equational laws”; and an “effect constructor” S .

C The Curry-Howard correspondence

The [Gentzen-Vorobiev-Hudelmaier algorithm](#) and its generalizations

See also the [curryhoward](#) project

C.1 Slides

Type constructions in functional programming

The common ground between OCaml, Haskell, Scala, Rust, and other languages

Type constructions common in FP languages:

Tuple ("product") type: Int × String

Function type: Int ⇒ String

Disjunction ("sum") type: Int + String

Unit type ("empty tuple"): 1

Type parameters: List^T

Up to differences in syntax, the FP languages share all these features

Type constructions: Scala syntax

Tuple type: (Int, String)

Create: val pair: (Int, String) = (123, "abc")

Use: val y: String = pair._2

Function type: Int ⇒ String

Create: def f: (Int ⇒ String) = x ⇒ "Value is " + x.toString

Use: val y: String = f(123)

Disjunction type: Either[Int, String] defined in standard library

Create:

```
val x: Either[Int, String] = Left(123)
```

```
val y: Either[Int, String] = Right("abc")
```

Use: val z: Boolean = x match {

```
case Left(i) ⇒ i > 0
```

```
case Right(_) ⇒ false
```

}

Unit type: Unit

Create: val x: Unit = ()

Type constructions: OCaml syntax

Tuple type: int * string

Create: let pair: int * string = (123, "abc")

Use: let y: string = snd pair

Function type: int → string

Create: let f: int → string =

```
fun x -> Printf.sprintf "Value is %d" x
```

Use: let y: string = f 123

Disjunction type: type e = Left of int | Right of string

Create:

```
let x: e = Left 123
```

```
let y: e = Right "abc"
```

Use: let z: bool = match x with

```
Left i -> i > 0
```

```
Right _ -> false
```

Unit type: unit

Create: let x: unit = ()

Type constructions: Haskell syntax

Tuple type: `(Int, String)`

Create: `pair = (123, "abc")`

Use: `(_, y) = pair`

Function type: `Int -> String`

Create: `f = \x -> "Value is " ++ show x`

Use: `y = f 123`

Disjunction type: `data E = Left Int | Right String`

Create:

`x = Left 123`

`y = Right "abc"`

Use: `z = case x of`

`Left i -> i > 0`

`Right _ -> false`

Unit type: `Unit`

Create: `x = ()`

From types to propositions

The code `val x: T = ...` shows that we can compute a value of type `T` as part of our program expression

Let's denote this proposition by $\mathcal{CH}(T)$ – “Code \mathcal{H} has a value of type `T`”

Correspondence between types and propositions, for a given program:

Type	Proposition	Short notation
<code>T</code>	$\mathcal{CH}(T)$	T
<code>(A, B)</code>	$\mathcal{CH}(A) \text{ and } \mathcal{CH}(B)$	$A \wedge B; A \times B$
<code>Either[A, B]</code>	$\mathcal{CH}(A) \text{ or } \mathcal{CH}(B)$	$A \vee B; A + B$
<code>A -> B</code>	$\mathcal{CH}(A) \text{ implies } \mathcal{CH}(B)$	$A \Rightarrow B$
<code>Unit</code>	True	1

Type parameter `[T]` in a function type means $\forall T$

Example: `def dupl[A]: A -> (A, A)`. The type of this function, $A \Rightarrow A \times A$, corresponds to the theorem $\forall A : A \Rightarrow A \wedge A$

Translating language constructions into the logic I

How to represent logical relationships between $\mathcal{CH}(\dots)$ propositions?

Code expressions create logical relationships between propositions $\mathcal{CH}(\dots)$

“Logical relationships” = what will be true if something given is true

In formal logic, this statement is written in the syntax

$$X, Y, \dots, Z \vdash T$$

and is called a **sequent** having the premises X, Y, \dots, Z and the goal T .

A sequent in formal logic can be proved if proof task;

The elementary proof task is represented by a **sequent**

Notation: $A, B, C \vdash G$; the **premises** are A, B, C and the **goal** is G

Proofs are achieved via axioms and derivation rules

Axioms: such and such sequents are already true

Derivation rules: this sequent is true if such and such sequents are true

To make connection with logic, represent code fragments as **sequents**

$A, B \vdash C$ represents an expression of type `C` that uses `x: A` and `y: B`

Examples in Scala:

`(x: Int).toString + "abc"` is an expression of type `String` that uses an `x: Int` and is represented by the sequent `Int -> String`

C The Curry-Howard correspondence

`(x: Int) => x.toString + "abc"` is an expression of type `Int => String` and is represented by the sequent $\emptyset \vdash \text{Int} \Rightarrow \text{String}$

Sequents only describe the *types* of expressions and their parts

Translating language constructions into the logic II

What are the derivation rules for the logic of types?

Write all the constructions in FP languages as sequents

This will give all the derivation rules for the logic of types

Each type construction has an expression for creating it and an expression for using it

Tuple type $A \times B$

Create: $A, B \vdash A \times B$

Use: $A \times B \vdash A$ and also $A \times B \vdash B$

Function type $A \Rightarrow B$

Create: if we have $A \vdash B$ then we will have $\emptyset \vdash A \Rightarrow B$

Use: $A \Rightarrow B, A \vdash B$

Disjunction type $A + B$

Create: $A \vdash A + B$ and also $B \vdash A + B$

Use: $A + B, A \Rightarrow C, B \Rightarrow C \vdash C$

Unit type 1

Create: $\emptyset \vdash 1$

Translating language constructions into the logic III

Additional rules for the logic of types

In addition to constructions that use types, we have “trivial” constructions:

a single, unmodified value of type A is a valid expression of type A

For any A we have the sequent $A \vdash A$

if a value can be computed using some given data, it can also be computed if given *additional* data

If we have $A, \dots, C \vdash G$ then also $A, \dots, C, D \vdash G$ for any D

For brevity, we denote by Γ a sequence of arbitrary premises

the order in which data is given does not matter, we can still compute all the same things given the same premises in different order

If we have $\Gamma, A, B \vdash G$ then we also have $\Gamma, B, A \vdash G$

Syntax conventions:

the implication operation associates to the right

$A \Rightarrow B \Rightarrow C$ means $A \Rightarrow (B \Rightarrow C)$

precedence order: implication, disjunction, conjunction

$A + B \times C \Rightarrow D$ means $(A + (B \times C)) \Rightarrow D$

Quantifiers: implicitly, all our type variables are universally quantified

When we write $A \Rightarrow B \Rightarrow A$, we mean $\forall A : \forall B : A \Rightarrow B \Rightarrow A$

The logic of types I

Now we have all the axioms and the derivation rules of the logic of types.

What theorems can we derive in this logic?

Example: $A \Rightarrow B \Rightarrow A$

Start with an axiom $A \vdash A$; add an unused extra premise B : $A, B \vdash A$

Use the “create function” rule with B and A , get $A \vdash B \Rightarrow A$

Use the “create function” rule with A and $B \Rightarrow A$, get the final sequent $\emptyset \vdash A \Rightarrow B \Rightarrow A$ showing that $A \Rightarrow B \Rightarrow A$ is a **theorem** since it is derived from no premises

What code does this describe?

The axiom $A \vdash A$ represents the expression x^A where x is of type A

The unused premise B corresponds to unused variable y^B of type B

The “create function” rule gives the function $y^B \Rightarrow x^A$

The second “create function” rule gives $x^A \Rightarrow (y^B \Rightarrow x)$

Scala code: `def f[A, B]: A => B => A = (x: A) => (y: B) => x`

Any code expression's type can be translated into a sequent

A proof of a theorem directly guides us in writing code for that type

Correspondence between programs and proofs

By construction, any theorem can be implemented in code

Proposition	Code
$\forall A : A \Rightarrow A$	<code>def identity[A](x: A): A = x</code>
$\forall A : A \Rightarrow 1$	<code>def toUnit[A](x: A): Unit = ()</code>
$\forall A \forall B : A \Rightarrow A + B$	<code>def inLeft[A,B](x:A): Either[A,B] = Left(x)</code>
$\forall A \forall B : A \times B \Rightarrow A$	<code>def first[A,B](p: (A,B)): A = p._1</code>
$\forall A \forall B : A \Rightarrow B \Rightarrow A$	<code>def const[A,B](x: A): B ⇒ A = (y:B) ⇒ x</code>

Also, non-theorems *cannot be implemented* in code

Examples of non-theorems:

$\forall A : 1 \Rightarrow A; \quad \forall A \forall B : A + B \Rightarrow A;$

$\forall A \forall B : A \Rightarrow A \times B; \quad \forall A \forall B : (A \Rightarrow B) \Rightarrow A$

Given a type's formula, can we implement it in code? Not obvious.

Example: $\forall A \forall B : (((A \Rightarrow B) \Rightarrow A) \Rightarrow A) \Rightarrow B \Rightarrow B$

Can we write a function with this type? Can we prove this formula?

The logic of types II

What kind of logic is this? What do mathematicians call this logic?

This is called "intuitionistic propositional logic", IPL (also "constructive")

This is a "nonclassical" logic because it is different from Boolean logic

Disjunction works very differently from Boolean logic

Example: $A \Rightarrow B + C \vdash (A \Rightarrow B) + (A \Rightarrow C)$ does not hold in IPL

This is counter-intuitive!

We cannot implement a function with this type:

```
def q[A,B,C](f: A ⇒ Either[B, C]): Either[A ⇒ B, A ⇒ C]
```

Disjunction is "constructive": need to supply one of the parts

But `Either[A ⇒ B, A ⇒ C]` is not a function of `A`

Implication works somewhat differently

Example: $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ holds in Boolean logic but not in IPL

Cannot compute an `x: A` because of insufficient data

Conjunction works the same as in Boolean logic

Example:

$$A \Rightarrow B \times C \vdash (A \Rightarrow B) \times (A \Rightarrow C)$$

The logic of types III

How to determine whether a given IPL formula is a theorem?

The IPL cannot have a truth table with a fixed number of truth values

This was shown by Gödel in 1932 (see [Wikipedia page](#))

The IPL has a decision procedure (algorithm) that either finds a proof for a given IPL formula, or determines that there is no proof

There may be several inequivalent proofs of an IPL theorem

Each proof can be *automatically translated* into code

The `curryhoward` library implements an IPL prover as a Scala macro, and generates Scala code from types

The `djinn-ghc` compiler plugin and the `JustDoIt` plugin implement an IPL prover in Haskell, and generate Haskell code from types

All these IPL provers use the same basic algorithm called LJT

and all cite the same paper [\[Dyckhoff 1992\]](#)

because most other papers on this subject are incomprehensible to non-specialists, or describe algorithms that are too complicated

Proof search I: looking for an algorithm

Why our initial presentation of IPL does not give a proof search algorithm

The FP type constructions give nine axioms and three derivation rules:

- $\Gamma, A, B \vdash A \times B$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

- $\Gamma, A \times B \vdash A$

$$\frac{\Gamma \vdash G}{\Gamma, D \vdash G}$$

- $\Gamma, A \times B \vdash B$

$$\frac{\Gamma, A \Rightarrow B, A \vdash B}{\Gamma, A, B \vdash G}$$

- $\Gamma, A \Rightarrow B, A \vdash B$

$$\frac{}{\Gamma, B, A \vdash G}$$

- $\Gamma, A \vdash A + B$

- $\Gamma, B \vdash A + B$

- $\Gamma, A + B, A \Rightarrow C, B \Rightarrow C \vdash C \vdash C$

- $\Gamma \vdash 1$

- $\Gamma, A \vdash A$

Can we use these rules to obtain a finite and complete search tree? No.

Try proving $A, B + C \vdash A \times B + C$: cannot find matching rules

Need a better formulation of the logic

Proof search II: Gentzen's calculus LJ (1935)

A "complete and sound calculus" is a set of axioms and derivation rules that will yield all (and only!) theorems of the logic

$$\begin{array}{c}
 (X \text{ is atomic}) \frac{}{\Gamma, X \vdash X} Id \\
 \frac{\Gamma, A \Rightarrow B \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C} L \Rightarrow \\
 \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A + B \vdash C} L+ \\
 \frac{\Gamma, A_i \vdash C}{\Gamma, A_1 \times A_2 \vdash C} L \times_i
 \end{array}
 \quad
 \begin{array}{c}
 \frac{}{\Gamma \vdash \top} \top \\
 \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} R \Rightarrow \\
 \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2} R+i \\
 \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B} R \times
 \end{array}$$

Two axioms and eight derivation rules

Each derivation rule says: The sequent at bottom will be proved if proofs are given for sequent(s) at top

Use these rules "bottom-up" to perform a proof search

Sequents are nodes and proofs are edges in the proof search tree

Proof search example I

Example: to prove $((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$

Root sequent $S_0 : \emptyset \vdash ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$

S_0 with rule $R \Rightarrow$ yields $S_1 : (R \Rightarrow R) \Rightarrow Q \vdash Q$

S_1 with rule $L \Rightarrow$ yields $S_2 : (R \Rightarrow R) \Rightarrow Q \vdash R \Rightarrow R$ and $S_3 : Q \vdash Q$

Sequent S_3 follows from the Id axiom; it remains to prove S_2

S_2 with rule $L \Rightarrow$ yields $S_4 : (R \Rightarrow R) \Rightarrow Q \vdash R \Rightarrow R$ and $S_5 : Q \vdash R \Rightarrow R$

We are stuck here because $S_4 = S_2$ (we are in a loop)

We can prove S_5 , but that will not help

So we backtrack (erase S_4, S_5) and apply another rule to S_2

S_2 with rule $R \Rightarrow$ yields $S_6 : (R \Rightarrow R) \Rightarrow Q; R \vdash R$

Sequent S_6 follows from the Id axiom

Therefore we have proved S_0

Since $((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$ is derived from no premises, it is a theorem

Q.E.D.

Proof search III: The calculus LJT

Vorobieff-Hudelmaier-Dyckhoff, 1950-1990

The Gentzen calculus LJ will loop if rule $L \Rightarrow$ is applied ≥ 2 times

The calculus LJT keeps all rules of LJ except rule $L \Rightarrow$

Replace rule $L \Rightarrow$ by pattern-matching on A in the premise $A \Rightarrow B$:

$$\frac{(X \text{ is atomic})}{\Gamma, X, B \vdash D} L \Rightarrow_1$$

$$\frac{\Gamma, A \Rightarrow B \Rightarrow C \vdash D}{\Gamma, (A \times B) \Rightarrow C \vdash D} L \Rightarrow_2$$

$$\frac{\Gamma, A \Rightarrow C, B \Rightarrow C \vdash D}{\Gamma, (A + B) \Rightarrow C \vdash D} L \Rightarrow_3$$

$$\frac{\Gamma, B \Rightarrow C \vdash A \Rightarrow B \quad \Gamma, C \vdash D}{\Gamma, (A \Rightarrow B) \Rightarrow C \vdash D} L \Rightarrow_4$$

When using LJT rules, the proof tree has no loops and terminates

See [this paper](#) for an explicit decreasing measure on the proof tree

Proof search IV: The calculus LJT

"It is obvious that it is obvious" – a mathematician after thinking for a half-hour

Rule $L \Rightarrow_4$ is based on the key theorem:

$$((A \Rightarrow B) \Rightarrow C) \Rightarrow (A \Rightarrow B) \iff (B \Rightarrow C) \Rightarrow (A \Rightarrow B)$$

The key theorem for rule $L \Rightarrow_4$ is attributed to Vorobieff (1958):

be extracted from Lemma 7 in [22]. One could also go further and make subproofs sensible.

LEMMA 2. $\vdash_{LJ} \Gamma, (C \supset D) \supset B \Rightarrow C \supset D$ iff $\vdash_{LJ} \Gamma, D \supset B \Rightarrow C \supset D$.

PROOF. Trivial [34].

THEOREM 1. *The systems LJ and LJT are equivalent.*

PROOF. As noted earlier, it is routine to show that any sequent provable

[R. Dyckhoff, *Contraction-Free Sequent Calculi for Intuitionistic Logic*, 1992]

A stepping stone to this theorem:

$$((A \Rightarrow B) \Rightarrow C) \Rightarrow B \Rightarrow C$$

Proof (obviously trivial): $f^{(A \Rightarrow B) \Rightarrow C} \Rightarrow b^B \Rightarrow f(x^A \Rightarrow b)$

Details are left as exercise for the reader

Proof search V: From deduction rules to code

The new rules are equivalent to the old rules, therefore...

Proof of a sequent $A, B, C \vdash G \Leftrightarrow$ code/expression $t(a, b, c) : G$

Also can be seen as a function t from A, B, C to G

Sequent in a proof follows from an axiom or from a transforming rule

The two axioms are fixed expressions, $x^A \Rightarrow x$ and 1

Each rule has a *proof transformer* function: $PT_{R \Rightarrow}$, PT_{L+} , etc.

Examples of proof transformer functions:

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A + B \vdash C} L+$$

$$PT_{L+}(t_1^{A \Rightarrow C}, t_2^{B \Rightarrow C}) = x^{A+B} \Rightarrow x \text{ match } \begin{cases} a^A \Rightarrow t_1(a) \\ b^B \Rightarrow t_2(b) \end{cases}$$

$$\frac{\Gamma, A \Rightarrow B \Rightarrow C \vdash D}{\Gamma, (A \times B) \Rightarrow C \vdash D} L \Rightarrow_2$$

$$PT_{L \Rightarrow_2}(f^{(A \Rightarrow B \Rightarrow C) \Rightarrow D}) = g^{A \times B \Rightarrow C} \Rightarrow f(x^A \Rightarrow y^B \Rightarrow g(x, y))$$

Verify that we can indeed produce PTs for every rule of LJT

Proof search example II: deriving code

Once a proof tree is found, start from leaves and apply PTs

For each sequent S_i , this will derive a **proof expression** t_i

Example: to prove S_0 , start from S_6 backwards:

$$\begin{aligned} S_6 : (R \Rightarrow R) \Rightarrow Q; R \vdash R & \quad (\text{axiom } Id) \quad t_6(rrq, r) = r \\ S_2 : (R \Rightarrow R) \Rightarrow Q \vdash (R \Rightarrow R) & \quad PT_{R \Rightarrow}(t_6) \quad t_2(rrq) = (r \Rightarrow t_6(rrq, r)) \\ S_3 : Q \vdash Q & \quad (\text{axiom } Id) \quad t_3(q) = q \\ S_1 : (R \Rightarrow R) \Rightarrow Q \vdash Q & \quad PT_{L \Rightarrow}(t_2, t_3) \quad t_1(rrq) = t_3(rrq(t_2(rrq))) \\ S_0 : \emptyset \vdash ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q & \quad PT_{R \Rightarrow}(t_1) \quad t_0 = (rrq \Rightarrow t_1(rrq)) \end{aligned}$$

The proof expression for S_0 is then obtained as

$$\begin{aligned} t_0 &= rrq \Rightarrow t_3(rrq(t_2(rrq))) = rrq \Rightarrow rrq(r \Rightarrow t_6(rrq, r)) \\ &= rrq \Rightarrow rrq(r \Rightarrow r) \end{aligned}$$

Simplified final code having the required type:

$$t_0 : ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q = (rrq \Rightarrow rrq(r \Rightarrow r))$$

C.2 Intuitionistic propositional logic (IPL)

The intuitionistic propositional logic (sometimes also called the “constructive” propositional logic) describes how programs in functional programming languages may be able to compute values of different types.

The main formal difference between IPL and the classical (Boolean) logic is that IPL does not include the axiom of excluded middle (“*tertium non datur*”), which is

$$\forall A : (A \text{ or } (\text{not } (A))) \text{ is true} .$$

However, given just this information, it is not easy to understand the consequences of *not having* this axiom, or to figure out which statements are true in the IPL.

The reason this axiom is not included in IPL is that IPL propositions such as $\mathcal{CH}(A)$ correspond to the *practical possibility* of values of type A to be computed by a program. For the proposition $\mathcal{CH}(A)$ to be true in IPL, a program needs to actually compute a value of type A . It is not sufficient merely to show that the non-existence of such a value would be somehow contradictory. But in classical logic, the axiom of excluded middle says that either $\mathcal{CH}(A)$ or $\text{not } (\mathcal{CH}(A))$ is true. So showing that “ $\text{not } (\mathcal{CH}(A))$ ” is contradictory is sufficient for proving $\mathcal{CH}(A)$, without ever computing any values of type A . For this reason, classical (Boolean) logic does not adequately describe the logic of types in functional programming, i.e. it does not correctly predict the types of values that can be computed by functional programs.

C.3 Example: The logic of types is not Boolean

Here is an explicit example of obtaining an incorrect result when using classical logic to reason about values computed by functional programs. Consider the formula

$$(A \Rightarrow B + C) \Rightarrow (A \Rightarrow B) + (A \Rightarrow C) \tag{C.1}$$

or, putting in all the parentheses for clarity,

$$(A \Rightarrow (B + C)) \Rightarrow ((A \Rightarrow B) + (A \Rightarrow C)) \quad .$$

This formula is a true theorem in classical logic.

In the IPL, it turns out that Eq. (C.1) is not a valid theorem, i.e. it is impossible to find a proof of Eq. (C.1) using the axioms and the derivation rules of the IPL. To prove that there is no proof, one needs to use methods of proof theory that are beyond the scope of this book. A good introduction to the required technique is the book "*Proof and Disproof in Formal Logic*" by R. Bornat.¹

This example illustrates that it is precisely the valid theorems in the IPL, and not the valid theorems in the Boolean logic, that correspond to implementable functional programs.

C.4 Using truth values in Boolean logic and in IPL

Another significant difference between IPL and the Boolean logic is that propositions in IPL cannot be assigned a fixed set of "truth values". This was proved by Gdel in 1935. It means that a proposition in IPL cannot be decided by writing out a truth table, even if we allow more than two truth values.

¹ R. Bornat, "Proof and Disproof in Formal Logic", Oxford, 2005 - link to Amazon.com

D Category theory

Examples of categories

1. Objects: types Int, String, ...; morphisms (arrows) are functions $\text{Int} \rightarrow \text{String}$ etc. – this is the “standard” category corresponding to a given programming language
2. Objects: types A, B, \dots ; morphisms are pairs of functions $(A \rightarrow B), (B \rightarrow A)$
3. * Objects: types $\text{List}^A, \text{List}^B, \dots$; morphisms are functions of type $\text{List}^A \rightarrow \text{List}^B$
4. Objects: types A, B, \dots ; morphisms are functions of type $\text{List}^A \rightarrow \text{List}^B$
5. Objects: types A, B, \dots ; morphisms are functions of type $A \rightarrow \text{List}^B$
6. * Objects: types $\text{List}^A, \text{List}^B, \dots$; morphisms are functions $A \rightarrow B$
7. Objects: types A, B, \dots ; morphisms are $\text{List}^{A \rightarrow B}$
8. Objects: types A, B, \dots ; morphisms are functions $B \rightarrow A$
9. * Objects: things A, B, \dots ; morphisms are pairs (A, B) of things – this is the same as a preorder

Examples marked with * are for illustration only, they are probably not very useful

E A humorous disclaimer

The following text is quoted in part from an anonymous source ("Project Guten Tag") dating back at least to 1997. The original text is no longer available on the Internet.

WARRANTO LIMITENSIS; DISCLAMATANTUS DAMAGENSIS
Sonus exceptus "Rectum Replacator Refundiens" describitus ecci,

1. Projectus (etque nunquam partum quis hic etext remitibus cum PROJECT GUTEN TAG-tm iden-tificator) disclamabat omni liabilitus tuus damagensis, pecuniensisque, includibantus pecunia legalitus, et
2. REMEDIA NEGLIGENTIA 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 recidio, pecuniam tuum re-fundatorium 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 IM-PPLICATO, 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 dam-agaren consequentialis, ecco lo qua disclamatori exclusatorique non vobis applicant, et potat optia alia legali.

F 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. Secondly, 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.

F.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, L^AT_EX 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.

F.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 F.0.2.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

F.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.

F.0.3 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections F.0.1 and F.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 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 F0.2 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 F0.3. 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 F0.3) to Preserve its Title (section F0.0) 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	Mathematical notation for symbolic computations with code.	114
5.1	The correspondence between type constructions and \mathcal{CH} -propositions.	127
5.2	Examples of logical formulas that are true theorems in Boolean logic.	132
5.3	Examples of logical formulas that are <i>not</i> true in Boolean logic.	132
5.4	Proof rules for the constructive logic.	137
5.5	Logic identities with disjunction and conjunction, and the corresponding equivalences of types.	144
5.6	Logical identities with implication, and the corresponding type equivalences and arithmetic identities.	150
5.7	Proof rules of constructive logic are true also in the Boolean logic.	170
6.1	Example translations of functor blocks into <code>.map</code> 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	Type constructions for the <code>Extractor</code> typeclass.	230

List of Figures

3.1	The disjoint domain represented by the <code>RootsOfQ</code> type.	94
5.1	Proof tree for the sequent (5.8).	138
7.1	Using a <code>trait</code> for implementing the <code>Monoid</code> typeclass and a recursive instance.	243
19.1	A programmer performs a derivation in Haskell before writing code.	379
20.1	Mixing incompatible data types gives nonsense results.	381
20.2	The Pyongyang method of error-free programming.	382

Index

- F*-algebra, 393
- "trivial" semigroup, 237
- abstract syntax tree, 87
- accumulator argument, 37
- accumulator trick, 37
- aggregation, 1, 11, 39, 40
- algebra, 393
- algebraic, 393
- algebraic data type, 392
- applicative functors, 372, 373
- applied functional type theory, 372, 378
- argument group, 51
- assembly language, 19
- associativity law
 - of addition, 38
 - of function composition, 105
 - of semigroup, 224
- backward composition, 104, 388
- bifunctor, 197
- binary search, 51
- binary tree, 84
- Boolean logic, 131
- bound variable, 8
- cardinality, 147
- Cartesian product, 147
- case class, 64
- case expression, 22
- closure, 100
- co-pointed bifunctor, 250
- co-pointed functor, 248
- co-product type, 391
- code notation, 391
- codensity monad, 365
- Collatz sequence, 56
- commutative diagram, 176
- companion object, 220
- compatibility law
 - for pointed and co-pointed functors, 248
- composition law
 - of functors, 174
- computation expressions, 391
- computational equivalence, 102, 246
- conjunction in logic, 96
- constant contrafunctor, 207
- constant function, 104
- constant functor, 200
- constructive logic, 171
- constructive propositional logic, 136
- continuation-passing, 49
- contrafunctor, 187, 391
- contravariant functor, 391
- contravariant position, 191
- covariant position, 191
- curried arguments, 100
- curried function, 100
- Curry-Howard correspondence, 140, 382
- default value, 37
- dependent type, 155, 216
- derivation rule, 135
- destructuring, 22
- dictionary, 19
- disjoint domain, 94
- disjoint union, 95
- disjunction in logic, 96
- disjunctive type, 70, 391
 - matrix notation, 146
- do-notation (Haskell), 391
- dynamic programming, 55, 58
- eager collection, 60
- eager value, 60
- embedded 'if', 71
- empty case class, 67
- enumeration type, 89, 96
- Euler product, 15
- exception, 58, 78, 169
- exercises, 14, 32, 55, 84, 85, 87, 93, 118, 130, 147, 155, 165, 195, 211, 253, 314, 361
- expanded function, 233
- exponent, 149
- exponential functor, 203
- exponential-polynomial type, 167, 391
- expression, 4
- expression block, 6
- extension methods, 221
- extractor typeclass, 230

- factorial function, 4
- final tagless, 393
- first-order logic, 156
- flipped Kleisli, 343
- for-comprehension, 391
- formal logic, 134
- forward composition, 104, 388
- forward notation, 106
- free variable, 8
- fully parametric function, 103, 183
- function composition, 104
- function product, 200
- function value, 5, 102
- functional programming, 15
- functor, 176
- functor block, 179, 391
- functor co-product, 202
- functor composition, 204
- functor product, 200
- GADT, 183, 392
- generalized algebraic data types, 183
- generic functions, 105
- hashmap, 19
- higher-order function, 119
- identity function, 104
- identity functor, 199
- identity laws
 - of equality, 185
 - of function composition, 105
 - of functors, 173
 - of monoid, 227
- immutable value, 61
- implication, 130
- implicit value, 219
- inductive typeclass, 254
- infinite type recursion, 79, 80
- infix syntax, 7
- information loss, 92, 142, 143, 162, 173
- interpreter, 88
- intuitionistic propositional logic, 136
- inverse function, 214
- isomorphic types, 141
- iterator, 61
- Kleisli arrow, 391
- Kleisli function, 391
- Kleisli morphism, 391
- λ -calculus, 20
- lawful functor, 186
- lazy collection, 60
- lazy value, 60
- left inverse, 214
- lifting, 73, 75, 174
- LJT algorithm, 139
- local scope, 6, 19, 59
- logical axiom, 134
- loop detection, 53
- Machin's formula, 14
- map/reduce style, 11
- mathematical induction, 14, 17, 34
 - base case, 34
 - inductive assumption, 34
 - inductive step, 34
- matrix notation, 146, 389
- method, 392
- monad transformers
 - base lifting, 311
 - base runner laws, 311
 - identity law, 311
 - lifting law, 311
 - lifting laws of runner, 311
 - monad construction law, 311
 - non-degeneracy law, 311
 - runner laws, 311
 - stacking, 317
- monadic morphism, 313
- monadic naturality, 331
- monads, 26
 - 3-swap law, 355
 - choice monad, 342
 - linear, 335, 342
 - rigid, 342
 - search monad, 343
 - stack of, 317
- monoid, 227
- monoidal convolution, 313
- name shadowing, 120
- nameless function, 5, 392
- naturality law
 - of extract, 248
 - of pure, 244
 - of pure for contrafunctors, 250
- negation, 169
- non-empty list, 82
- on-call value, 60, 151
- opaque type, 64
- operator syntax, 120
- order of a function, 119

- palindrome integer, 57
- paradigm, 15
- parametric code, 66
- parametricity, 105
- partial application, 102, 122
- partial function, 58, 71, 114, 214, 216
- partial type-to-value function, 216, 392
- pattern matching, 22
 - infallible, 59
 - matrix notation, 146
- pattern variables, 22, 23
- Paweł Szulc, 185
- perfect number, 57
- pipe notation, 196, 388
- planned exception, 78
- pointed contrafunctor, 250
- pointed functor, 244
- pointed type, 224
- polynomial functor, 209, 392
- polynomial functors, 202
 - recursive, 179
- polynomial type, 167
- predicate, 7
- procedure, 93
- product functor, 200
- product type, 147, 392
- profunctor, 213, 392
- profunctors, 212
- proof, 124
- proposition in logic, 96
- PTVF, 216
- pure compatibility laws, 328
- pure function, 61
- recursion
 - accumulator argument, 37
 - accumulator trick, 37
- recursive function, 35
 - termination, 233
- recursive type, 79
- referential transparency, 61
- reflexivity law, 235
- reflexivity law of equality, 185
- regular-shaped tree, 85
- Riemann zeta function, 15
- rigid functor, 359
- Robert C. Martin, 376
- rose tree, 85
- run-time error, 58
- runner, 88
- saturated application, 102
- Scala method, 102
- Scala's Iterator is broken, 61–63
- search monad, 343
- selector monad, 359
- semigroup, 224
- sequent, 395
 - example, 124
 - goal, 124
 - premises, 124
- serializer, 234
- shadowed name, 59
- Simpson's rule, 19
- solved examples, 11, 27, 41, 49, 70, 74, 82, 86, 89, 110, 113, 127, 143, 148, 149, 156, 177, 191, 210, 222, 253, 361
- stack memory, 36
- standard type constructions, 125
- stream, 46
- structural analysis, 206
- sum type, 148, 391
- symbolic calculations, 108
- tagged union type, 391
- tail recursion, 36
- total function, 58, 216
- trampolines, 49
- transformation, 10
- transitivity law of equality, 185
- truth table, 131
- tuple accessor, 21
- tuple types
 - as function arguments, 103
- tuples, 21, 392
 - fields, 21
 - nested, 22, 48
 - parts, 21
- turnstile, 135
- type alias, 42, 64, 126
- type annotation, 74, 91, 386
- type checking, 113
- type constructor, 67
 - contravariant, 190
 - covariant, 190
- type conversion function, 189
- type diagram, 176
- type domain, 217
- type equivalence, 141
 - accidental, 148, 165
- type error, 21, 22, 31, 58, 65
- type inference, 112, 121
- type notation, 125, 392
 - operator precedence, 127
- type parameter, 24, 66

type reflection, 185
type safety, 168
type-level function, 204
type-to-type function, 216
type-to-value function, 216
typeclass, 215
 Eq, 234
 Pointed, 244
 Semigroup, 224
 Show, 234
typeclass constraint, 215
typeclass instance, 217
typed hole, 160
types, 17
 equivalent, 141
 exponential-polynomial, 167
 isomorphic, 141
 polynomial, 167, 177
 structural analysis, 229
 subtype of, 189

uncurried function, 100, 102
undecidable logic, 156
unevaluated expression, 87
unfold function, 53, 57
unfunctor, 188, 213, 217, 387, 392
unfunctors, 183
unit type, 67, 386
 named, 67, 95, 125, 128
universal quantifier, 126
unplanned exception, 78

value semantics, 61
variable, 16, 17
variance annotation, 190
void type, 74, 80, 143, 151, 169, 217, 386

Wallis product, 11
well-typed expression, 113