J.N. Oliveira

University of Minho

PROGRAM DESIGN BY CALCULATION

(DRAFT of textbook in preparation)

Last update: October 2020

CONTENTS

Preamble 1

1 INTRODUCTION 3
- 1.1 A bit of archaeology 3
- 1.2 The future ahead 9
- 1.3 Summary 12
- 1.4 Why this Book? 14

I CALCULATING WITH FUNCTIONS 15

2 AN INTRODUCTION TO POINTFREE PROGRAMMING 16
- 2.1 Introducing functions and types 17
- 2.2 Functional application 18
- 2.3 Functional equality and composition 18
- 2.4 Identity functions 21
- 2.5 Constant functions 21
- 2.6 Monics and epics 23
- 2.7 Isos 24
- 2.8 Gluing functions which do not compose — products 25
- 2.9 Gluing functions which do not compose — coproducts 31
- 2.10 Mixing products and coproducts 34
- 2.11 Elementary datatypes 36
- 2.12 Natural properties 39
- 2.13 Universal properties 40
- 2.14 Guards and McCarthy's conditional 43
- 2.15 Gluing functions which do not compose — exponentials 46
- 2.16 Finitary products and coproducts 53
- 2.17 Initial and terminal datatypes 54
- 2.18 Sums and products in HASKELL 56
- 2.19 Exercises 59
- 2.20 Bibliography notes 62

3 RECURSION IN THE POINTFREE STYLE 64
- 3.1 Motivation 64
- 3.2 From natural numbers to finite sequences 69
- 3.3 Introducing inductive datatypes 74
- 3.4 Observing an inductive datatype 79
- 3.5 Synthesizing an inductive datatype 82
- 3.6 Introducing (list) catas, anas and hylos 83
- 3.7 Inductive types more generally 88
- 3.8 Functors 89
- 3.9 Polynomial functors 90
- 3.10 Polynomial inductive types 92
- 3.11 F-algebras and F-homomorphisms 93

3.12 F-catamorphisms 94
3.13 Parameterization and type functors 96
3.14 A catalogue of standard polynomial inductive types 101
3.15 Hylo-factorization 103
3.16 Functors and type functors in HASKELL 107
3.17 The mutual-recursion law 108
3.18 “Banana-split”: a corollary of the mutual-recursion law 117
3.19 Inductive datatype isomorphism 119
3.20 Bibliography notes 121
4 WHY MONADS MATTER 123
4.1 Partial functions 123
4.2 Putting partial functions together 124
4.3 Lists 126
4.4 Monads 127
4.5 Monadic application (binding) 130
4.6 Sequencing and the **do**-notation 130
4.7 Generators and comprehensions 131
4.8 Monads in HASKELL 133
4.9 The state monad 136
4.10 ‘Monadification’ of Haskell code made easy 142
4.11 Monadic recursion 146
4.12 Where do monads come from? 147
4.13 Adjoint catamorphisms 151
4.14 Bibliography notes 158
II CALCULATING WITH RELATIONS 159
5 WHEN EVERYTHING BECOMES A RELATION 160
5.1 Functions are not enough 160
5.2 From functions to relations 162
5.3 Pre/post conditions 164
5.4 Relational composition and converse 166
5.5 Relational equality 169
5.6 Diagrams 171
5.7 Taxonomy of binary relations 173
5.8 Functions, relationally 178
5.9 Meet and join 181
5.10 Relational thinking 184
5.11 Monotonicity 186
5.12 Rules of thumb 188
5.13 Endo-relations 190
5.14 Relational pairing 193
5.15 Relational coproducts 196
5.16 On key-value data models 199
5.17 What about relational “currying”? 200
5.18 Galois connections 203
5.19 Relation division 209

5.20 Predicates also become relations 218
5.21 Guards, coreflexives and the McCarthy conditional 220
5.22 Difunctionals 224
5.23 Other orderings on relations 226
5.24 Back to functions 232
5.25 Bibliography notes 233
6 THEOREMS FOR FREE BY CALCULATION 235
6.1 Introduction 235
6.2 Polymorphic type signatures 236
6.3 Relators 237
6.4 A relation on functions 238
6.5 Free theorem of type *t* 240
6.6 Examples 240
6.7 Catamorphism laws as free theorems 244
6.8 Bibliography notes 246
7 CONTRACT-ORIENTED PROGRAMMING 247
7.1 Contracts 248
7.2 Library loan example 251
7.3 Mobile phone example 256
7.4 A calculus of functional contracts 258
7.5 Relational Hoare Logic 261
7.6 Abstract interpretation 262
7.7 Safety and liveness properties 265
7.8 Examples 267
7.9 “Free contracts” 273
7.10 Reasoning by approximation 274
7.11 Bibliography notes 277
8 PROGRAMMING FROM RELATIONAL HYLOMORPHISMS 278
8.1 Relational catamorphisms 278
8.2 Relational hylomorphisms 283
8.3 Inductive predicates 285
8.4 Inductive equivalence relations 285
8.5 Inductive datatype isomorphism revisited 287
9 CALCULATIONAL PROGRAM REFINEMENT 289

III CALCULATING WITH MATRICES 290
10 TOWARDS A LINEAR ALGEBRA OF PROGRAMMING 291
A BACKGROUND — EINDHOVEN QUANTIFIER CALCULUS 292
A.1 Notation 292
A.2 Rules 292
B HASKELL SUPPORT LIBRARY 294

LIST OF EXERCISES

Exercise 2.1 22
Exercise 2.2 22
Exercise 2.3 24
Exercise 2.4 31
Exercise 2.5 34
Exercise 2.6 34
Exercise 2.7 35
Exercise 2.8 36
Exercise 2.9 36
Exercise 2.10 36
Exercise 2.11 38
Exercise 2.12 38
Exercise 2.13 38
Exercise 2.14 39
Exercise 2.15 40
Exercise 2.16 40
Exercise 2.17 42
Exercise 2.18 42
Exercise 2.19 42
Exercise 2.20 42
Exercise 2.21 42
Exercise 2.22 45
Exercise 2.23 46
Exercise 2.24 46
Exercise 2.25 46
Exercise 2.26 52
Exercise 2.27 52
Exercise 2.28 53
Exercise 2.29 53
Exercise 2.30 54
Exercise 2.31 55
Exercise 2.32 59
Exercise 2.33 59
Exercise 2.34 59
Exercise 2.35 59
Exercise 2.36 59
Exercise 2.37 60
Exercise 2.38 60
Exercise 2.39 60
Exercise 2.40 61
Exercise 2.41 61

Exercise 2.42 61
Exercise 2.43 61
Exercise 2.44 61
Exercise 2.45 61
Exercise 3.1 68
Exercise 3.2 68
Exercise 3.3 68
Exercise 3.4 69
Exercise 3.5 69
Exercise 3.6 79
Exercise 3.7 87
Exercise 3.8 87
Exercise 3.9 87
Exercise 3.10 92
Exercise 3.11 100
Exercise 3.12 101
Exercise 3.13 102
Exercise 3.14 102
Exercise 3.15 102
Exercise 3.16 102
Exercise 3.17 103
Exercise 3.18 105
Exercise 3.19 106
Exercise 3.20 106
Exercise 3.21 106
Exercise 3.22 107
Exercise 3.23 107
Exercise 3.24 108
Exercise 3.25 108
Exercise 3.26 112
Exercise 3.27 113
Exercise 3.28 113
Exercise 3.29 114
Exercise 3.30 114
Exercise 3.31 114
Exercise 3.32 114
Exercise 3.33 115
Exercise 3.34 116
Exercise 3.35 118
Exercise 3.36 118
Exercise 3.37 118
Exercise 3.38 121
Exercise 4.1 126
Exercise 4.2 126
Exercise 4.3 127
Exercise 4.4 129

Exercise 4.5 132
Exercise 4.6 133
Exercise 4.7 133
Exercise 4.8 134
Exercise 4.9 135
Exercise 4.10 136
Exercise 4.11 146
Exercise 4.12 146
Exercise 4.13 151
Exercise 5.1 168
Exercise 5.2 168
Exercise 5.3 173
Exercise 5.4 176
Exercise 5.5 177
Exercise 5.6 177
Exercise 5.7 177
Exercise 5.8 177
Exercise 5.9 178
Exercise 5.10 179
Exercise 5.11 180
Exercise 5.12 181
Exercise 5.13 181
Exercise 5.14 183
Exercise 5.15 183
Exercise 5.16 183
Exercise 5.17 183
Exercise 5.18 183
Exercise 5.19 186
Exercise 5.20 186
Exercise 5.21 187
Exercise 5.22 187
Exercise 5.23 189
Exercise 5.24 189
Exercise 5.25 191
Exercise 5.26 192
Exercise 5.27 192
Exercise 5.28 192
Exercise 5.29 192
Exercise 5.30 192
Exercise 5.31 192
Exercise 5.32 193
Exercise 5.33 195
Exercise 5.34 195
Exercise 5.35 196
Exercise 5.36 198
Exercise 5.37 198

Exercise 5.38 199
Exercise 5.39 199
Exercise 5.40 199
Exercise 5.41 202
Exercise 5.42 207
Exercise 5.43 207
Exercise 5.44 207
Exercise 5.45 207
Exercise 5.46 209
Exercise 5.47 209
Exercise 5.48 212
Exercise 5.49 212
Exercise 5.50 217
Exercise 5.51 217
Exercise 5.52 218
Exercise 5.53 220
Exercise 5.54 221
Exercise 5.55 224
Exercise 5.56 226
Exercise 5.57 229
Exercise 5.58 229
Exercise 5.59 229
Exercise 5.60 230
Exercise 5.61 230
Exercise 5.62 232
Exercise 6.1 237
Exercise 6.2 242
Exercise 6.3 242
Exercise 6.4 243
Exercise 6.5 243
Exercise 6.6 243
Exercise 6.7 243
Exercise 6.8 244
Exercise 6.9 244
Exercise 6.10 245
Exercise 6.11 245
Exercise 6.12 246
Exercise 7.1 250
Exercise 7.2 257
Exercise 7.3 259
Exercise 7.4 261
Exercise 7.5 263
Exercise 7.6 263
Exercise 7.7 264
Exercise 7.8 264
Exercise 7.9 264

Exercise 7.10 265
Exercise 7.11 273
Exercise 7.12 273
Exercise 7.13 274
Exercise 7.14 274
Exercise 8.1 283

PREAMBLE

This textbook, which has arisen from the author's research and teaching experience, has been in preparation for many years. Its main aim is to draw the attention of software practitioners to a calculational approach to the design of software artifacts ranging from simple algorithms and functions to the specification and realization of information systems.

Put in other words, the book invites software designers to raise standards and adopt mature development techniques found in other engineering disciplines, which (as a rule) are rooted on a sound mathematical basis. *Compositionality* and *parametricity* are central to the whole discipline, granting scalability from school desk exercises to large problems in an industry setting.

It is interesting to note that while coining the phrase *software engineering* in the 1960s, our colleagues of the time were already promising such high quality standards. In March, 1967, ACM President Anthony Oettinger delivered an address in which he said [67]:

> *"(...) the scientific, rigorous component of computing, is more like **mathematics** than it is like **physics**" (...) Whatever it is, on the one hand it has components of the purest of mathematics and on the other hand of the dirtiest of engineering.*

As a discipline, software engineering was announced at the Garmisch NATO conference in 1968, from whose report [63] the following excerpt is quoted:

> *In late 1967 the Study Group recommended the holding of a working conference on Software Engineering. The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.*

Provocative or not, the need for sound theoretical foundations has clearly been under concern since the very beginning of the discipline — exactly fifty years ago, at the time of writing. However, how "scientific" do such foundations turn out to be, now that five decades have since elapsed?[1]

Thirty years later (1997), Richard Bird and Oege de Moore published a textbook [12] in the preface of which C.A.R. Hoare writes:

> *Programming notation can be expressed by "**formulæ** and **equations** (...) which share the **elegance** of those which underlie **physics** and **chemistry** or any other branch of basic science".*

[1] The title of a communication of another ACM President, Vinton Cerf (2012), does not sound particularly optimistic [14].

The formulæ and equations mentioned in this quotation are those of a discipline known as the *Algebra of Programming*. Many others have contributed to this body of knowledge, notably Roland Backhouse and his colleagues at Eindhoven and Nottingham, see e.g. [1, 5], Jeremy Gibbons and Ralf Hinze at Oxford see e.g. [31], among many others. Unfortunately, references [1, 5] are still unpublished.

When the author of this draft textbook decided to teach *Algebra of Programming* to 2nd year students of the Minho degrees in computer science, back to 1998, he found textbook [12] too difficult for the students to follow, mainly because of its too explicit categorial (allegorical) flavour. So he decided to start writing slides and notes helping the students to read the book. Eventually, such notes became chapters 2 to 4 of the current version of the monograph. The same procedure was taken when teaching the relational approach of [12] to 4th year students (master level), see chapters 5 to 7.

This draft book is incomplete, all subsequent chapters being still in *slide form*[2]. Such half-finished chapters are omitted from the current print-out. Altogether, the idea is to show that software engineering and, in particular, computer programming can adopt the *scientific method* as other branches of engineering do. Somehow, it's like following in the footsteps of those who marveled at the power of algebraic reasoning in the remote past, in different contexts and disciplines:

> *"(...) De manera, que quien sabe por Algebra, sabe scientificamente* [In this way, who knows by Algebra knows scientifically]. Pedro Nunes, 1567 [66]

University of Minho, Braga, October 2020

José N. Oliveira

[2] See e.g. see technical report [71]. The third part will address a linear algebra of programming intended for quantitative reasoning about software. This is even less stable, but a number of papers exist already about the topic, starting from [70].

1

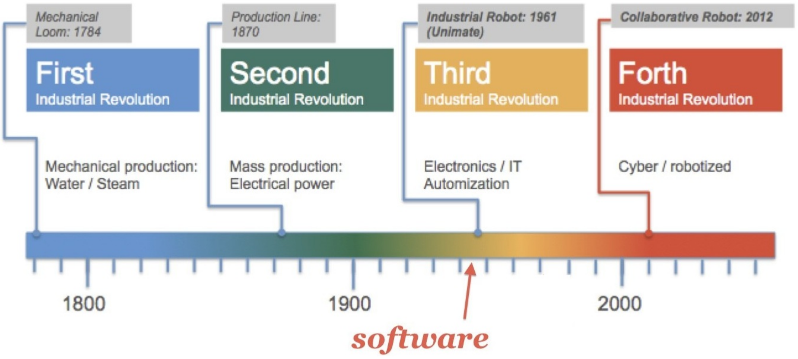
INTRODUCTION

This book is concerned with a major topic in software engineering: that of designing *correct* computer programs in the first place. Let us begin by inquiring into the phrase *software engineering* itself. Or even deeper into the word *software* itself.

Producing software is today an industrial activity which, according to Evans Data Corporation, involves more than 25 million developers. How did all this activity start? How productive is this industry? How relevant?

1.1 A BIT OF ARCHAEOLOGY

The following chart [1] gives the timeline of the four industrial revolutions that have changed human civilization in the last centuries:


Mechanical Loom: 1784
First Industrial Revolution
Mechanical production: Water / Steam
Production Line: 1870
Second Industrial Revolution
Mass production: Electrical power
Industrial Robot: 1961 (Unimate)
Third Industrial Revolution
Electronics / IT Automization
Collaborative Robot: 2012
Forth Industrial Revolution
Cyber / robotized
1800
1900
2000
software

This tells that, from the late 18c mechanical lines powered by steam and water streams, and through the late 19c mass production lines enabled by electrical power, one reaches the mid 20c third revolution in which electronics and IT automation starts to play a role. The role of software in industry started here, when dedicated electronics started to be replaced by devices that could be programmed and tailored to the specific tasks required.

Software had already proved its power during the second world war, in the military domain. However, it did not start there. It actually emerged from the theoretical work of two mathematicians, Alan Turing (1912-1954) and Alonzo Church (1903-1995). In fact, classical

[1] Credits: adapted from Nelmia Robotics Insight, 2015.

computing is rooted in mathematical abstractions that led in particular to the *Turing machine* [83] — which is still regarded as the canonical abstract notion of a programmable computer — and to the λ-calculus [15] — a mathematical system that provided the basis for the so-called functional programming paradigm.

The step from abstraction to reality was made possible by advances in physics, such as the invention of triodes (1912) and then of transistors (1948), a path that eventually lead to the integrated circuits that are the basis of the *in silico* technology of today [64, 79, 46].

Once such devices were first employed to store information in realistic situations, it became clear that further abstraction was required. This led to the explicit adoption of formal logic, a very important abstraction still in use today. As the aphorism says, *"logic is the language of computing"*.

The birth of software as an independent technology took place in the 1950s.[2] But it soon was faced with a crisis because an effective discipline of programming was lacking. Indeed, in the industrial setting software must have appeared as a kind of *"enfant terrible"*. *Hardware* and other "traditional" industrial *products* were fabricated according to well established laws of *physics* and principles of *engineering*. By contrast, *software* was not governed by the laws of physics: it did not weigh, did not smell, did not warm up/cool down, it was chemically neutral and it did not wear. In his 1967 address, ACM President Anthony Oettinger said [67]:

> *"(...) the scientific, rigorous component of computing, is more like* mathematics *than it is like physics".*

In spite of the tremendous progress in language design that took place throughout the 1960s — which witnessed the birth of what many regarded as the first *open source* project, the Algol family of languages — the crisis went on. Still Oettinger:

> *"It is a matter of* complexity*. Once you start putting thousands of these instructions together you create a* monster *which is* unintelligible *to anyone save its creator and, most of the time, unfortunately even to the creator."*

SOFTWARE ENGINEERING The term *software engineering* appeared in the late 1960s and was the subject of a conference supported by NATO that took place in Garmisch, Germany in 1968. The participants of this conference expressed concerns and called for stronger theoretical foundations [63]:

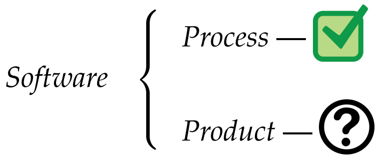
> *The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.*

[2] The first programming language, Fortran, appeared in 1953.

Many took the provocation seriously and embarked on researching so-called *formal methods* for developing code from formal specifications, notably Edsger Dijkstra (1930-2002), Tony Hoare (1934-), Niklaus Wirth (1934-) and Robert Floyd (1936-2001), all from the Algol development group and all Turing Awardees. This resulted in the birth of the principles of *structured programming* [18, 87] that became popular in the 1970s and reached the software industry in the form of a new, industry-strong language, Pascal, which evolved from the Algol culture.

In spite of these developments, the 1968 crisis was (and still is not...) over yet. The ever increasing complexity of the software that the IT sector requires programmers to build every day leads to unsafe code due to the widespread use of ad hoc methods, instead of the mathematically sound methods anticipated by its founding fathers. The main problem with such informal methods is that quality control is based on *testing* software artifacts *after* they have been built, and not on ensuring quality in a stepwise manner, as advocated by academia since the 1970s.

Oettinger's suggestion that software is, in a sense, *mathematics in motion*, was not accepted by a community that had been engaged in applying "traditional" engineering principles long before. As these are as applicable to the software *process* as in other disciplines, many researchers, teachers and practitioners were led into equating software engineering with the software development process itself. This is no doubt a very important aspect of the problem but tends to leave the study of the *software product* itself out of focus:



Fancy a chemical plant working in a perfect production line but in which nobody knows about the Lavoisier principle, the Mendeliev periodic table, the laws of chemical reactions and so on. By and large, this is still the kind of software industry that we have today, unfortunately. In a sense, managers and developers pretend that software production is not affected by its *special nature* and move on.

UNDERSTANDING COMPUTER PROGRAMS Today's widespread research on the mathematical meaning of software originated from the pioneering works of Floyd and Hoare [22, 34] (among others less known) in the late 1960s.[3] But these efforts had to wait for many years before the proposed techniques for program correctness proof were generally acclaimed and incorporated in tools that are more and more widespread today.

[3] But please note that Turing had himself already done similar but less known work back to 1949 [42].

Nearly a decade later, John Backus read, in his Turing Award Lecture, a revolutionary paper [8] about how he envisaged the future of computing software. This paper proclaimed conventional command-oriented programming languages obsolete because of their inefficiency arising from retaining, at a high-level, the so-called "memory access bottleneck" of the underlying computation model — the well-known *von Neumann* architecture. Alternatively, the (at the time already mature) *functional programming* style was put forward for two main reasons. Firstly, because of its potential for concurrent and parallel computation, which Backus envisaged as the future of computing. Secondly — and Backus emphasis was really put on this —, because of its strong mathematical basis.

Backus' *algebra of (functional) programs* was providential in alerting computer programmers that computer languages alone are insufficient, and that only languages which exhibit an *algebra* or *calculus* for reasoning about the objects they purport to describe will be useful in the long run.

The impact of Backus' first argument in the computing science and computer architecture communities was considerable, in particular if assessed in quality rather than quantity and in addition to the almost contemporary *structured programming* trend. By contrast, his second (theoretical) argument for changing computer programming was by and large ignored. Only the self-named *algebra of programming* research minority pursued in this direction.

TOWARDS AN ALGEBRA OF PROGRAMMING Even among those who defended mathematical approaches to software comprehension and development, consensus was rare about whether one should stay with imperative programming tamed by logical reasoning or move even further into the realm of algebraic functional programming. At the same time, the vast majority of programmers regarded all such efforts useless because the maths was too low level and got convoluted every time one tried to apply the available theories to practical case studies.

Indeed, in a relatively recent article [76], David Parnas questions such methods, which he regards still unfit for the software industry:

> *We must learn to use mathematics in software development, but we need to question, and be prepared to discard, most of the methods that we have been discussing and promoting for all these years.*

At the core of Parnas objections lies the contrast between the current ad-hoc (re)invention of burdening mathematical notation and less known elegant concepts which are neglected, often for cultural reasons or (lack of) background.

The question is: what is it that tells "good" and "bad" methods apart? As Parnas writes, *there is a disturbing gap between software development and traditional engineering disciplines*. In such disciplines one finds a successful, well-established mathematical background essentially made of calculus, linear algebra and probability theory.

Central to engineering mathematics is the construction of sets of simultaneous equations as models of physical systems (e.g. circuits, etc),

$$\left\{\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + a_{1m}x_m & = & b_1 \\ \vdots & \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{nm}x_m & = & b_n \end{array}\right. \tag{1.1}$$

that is, formulæ of the form

$$\langle \forall\ i\ :\ 1 \leqslant i \leqslant n : \sum_{j=1}^{m} a_{ij}x_j = b_i \rangle \tag{1.2}$$

The maturity of traditional engineering mathematics can be appreciated from the fact that such (often very large) sets of equations don't intimidate engineers, thanks to the *matrix* and *vector* concepts: grouping all coefficients a_{ij} of (1.1) in a matrix A, variables x_j in a vector X and values b_i in a vector B, (1.1) becomes

$$A \cdot X \quad = \quad B$$

where operator $(\cdot)$ denotes matrix multiplication. Backhouse [5] writes:

> *In this way a set of equations has been reduced to a single equation. This is a tremendous improvement in concision that does not incur any loss of precision!*

That is to say, such notation scales up and *quantity* does not disturb *quality*.

Another sign of maturity arises from the use of mathematical transformations, such as e.g. the Laplace transform [44] which changes the "mathematical space" so as to convert "difficult" sets of equations (e.g. differential) into "easy" ones (e.g. polynomial), whose solutions are mapped back to the original problem domain by the converse transform. Once again, complexity is controlled via effective mathematical techniques.

One may wonder about parallels to these techniques in mathematical methods for software design, in their use of formal logics. Do such logics scale up to very large sets of clauses issued by proof obligation generators, for instance? Is there a *linear algebra* for logic and set theory? Is there a logic equivalent to a *matrix*?

While the answer to the first question is *poorly!*, those to the other questions are affirmative: *yes, there are!* Quoting [76] once again:

> *There is an alternative. Some researchers have been studying the use of relational methods in computer science; (...) the well-known laws of relational algebra can serve as the axiomatic basis for programming. The axioms of relational algebra are simple and universal. This approach seems to have been neglected by most "mainline" researchers in the area of formal methods.*

Binary relations are Boolean matrices, thus providing a straight parallel with linear algebra. And relational composition of two relations R and S, usually denoted by a similar multiplicative term $R \cdot S$, provides another one. In set theory, this relational operator is defined indirectly as follows, assuming the *set-of-pairs* understanding of binary relations: pair (b, c) is in $R \cdot S$ iff there exist one or more mediating a such that $(b, a) \in R$ and $(a, c) \in S$.

If we look at one of the first definitions of this combinator, due to Charles Peirce (1839-1914) and explained in [50], we realize that it computes inner products like those of (1.2), where multiplication (restricted to 0s and 1s) captures logical conjunction and addition (resp. summation) captures disjunction (resp. existential quantification), if clipped at 1. Thereafter, relation union $R \cup S$ is nothing but index-wise Boolean matrix addition and the distributive laws

$$\begin{array}{rcl} R \cdot (S \cup T) & = & (R \cdot S) \cup (R \cdot T) \\ (S \cup T) \cdot R & = & (S \cdot R) \cup (T \cdot R) \end{array}$$

arise from the *bilinearity* of the underlying matrix algebra. Confirming the analogy, Gunther Schmidt's book on *relational mathematics* [80] makes extensive use of matrix notation, concepts and operations in relation algebra.

FURTHER DEVELOPMENTS Meanwhile, category theory [49] was born, emphasizing the description of mathematical concepts in terms of abstract *arrows* (morphisms) and *diagrams*, thus unveiling a compositional, abstract language of universal *combinators* that is inherently *generic* and *pointfree*.

The category of sets and functions immediately provided a basis for pointfree functional reasoning, but this was by and large ignored by Backus in his FP algebra of programs [8]. In any case, Backus's landmark FP paper was the first to show how relevant this reasoning style could be to programming. This happened four decades ago.

A bridge between the two pointfree schools — the relational and the categorical — was eventually established by Freyd and Scedrov [24] in their proposal of the concept of an *allegory*, which instantiates to *typed* relation algebra. The pointfree algebra of programming (AoP) as it is understood today [12] stems directly from [24].

In the early 1990s, the Groningen-Eindhoven MPC group led by Backhouse [1] contributed decisively to the AoP by structuring relation algebra in terms of easy-to-apprehend *rules* that make relational reasoning closer to school algebra. Think for instance of the following rule to reason about whole division of two natural numbers,

$$z \times y \leqslant x \equiv z \leqslant x \div y \qquad (y > 0) \tag{1.3}$$

assumed universally quantified in all its variables. Pragmatically, it expresses a "shunting" rule which enables one to exchange between a whole division at the upper side of an inequality and a multiplication at the lower side. Many properties of $(\times)$ and $(\div)$ can be inferred from (5.158), for instance $(x \div y) \times y \leqslant x$ — just replace z by $x \div y$ and simplify [4].

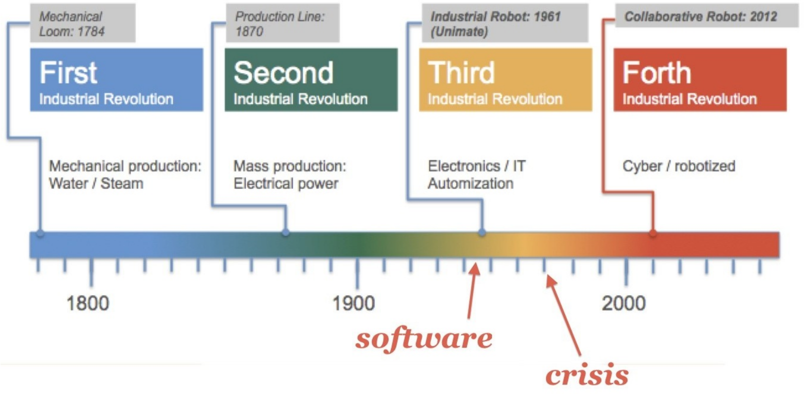
In 1997 — two decades after two landmark textbooks of the golden age of structured programming [18, 87] — Bird and Moor published a textbook [12] on how to use relational algebra to synthesise recursive programs from relational specifications, inaugurating a new discipline, called *Algebra of Programming*. In the preface of the book, computer science pioneer Tony Hoare writes:

> *Programming notation can be expressed by "**formulæ** and **equations** (...) which share the **elegance** of those which underlie **physics** and **chemistry** or any other branch of basic science".*

The five decade long aim of calculating software was achieved, but only partly. The book mostly covers calculating functional programs from relational specifications, or imperative programs with a purely functional semantics. Moreover, the techniques proposed were regarded as difficult and the book did not have the impact of, for instance, Dijkstra's [18] and Wirth's [87] textbooks two decades earlier.

1.2 THE FUTURE AHEAD

At the time of writing a new industrial revolution is under way — and in a rather singular way. It is depicted in the rightmost block in the picture we have already seen:



This trend has become known as *Industry 4.0* — the *fourth* industrial revolution — and is to rely on highly sophisticated software on an unprecedented scale.

Its main singularity resides in the fact that this is the first time a new industrial revolution happens solely (or mostly) relying on software

[4] Rule (5.158) *connects* division to multiplication, the latter helping to reason about the former. Functions connected in this way are said to be *adjoints*: multiplication is adjoint of division. Equivalences of this kind are scalable, powerful devices known in mathematics as Galois connections.

advances — in machine learning, robotics, security, reliability and so on. Billions of lines of code will be written every day and never was software *correctness* and *robustness* as essential as nowadays.

Concerning the challenges ahead, here is a quotation from a Robot Programming Tutorial:

> *"The fundamental challenge of all* robotics *is this: It is impossible to ever know the true state of the* environment. *A* robot *can* only guess *the state of the real world based on measurements returned by its* sensors.*"*

So the robot software programmer will need to live with the *abstract* view of the environment captured by the robot's sensors. Indeed, it would be impossible (and useless) for the robot software to cope with a 100% view of its environment.

ABSTRACTION Abstract view? Beware that software lives on *abstraction* by definition, even before the world became robot-dependent:

> *"The purpose of* abstraction *is not to be* vague, *but to create a new semantic level in which one can be* absolutely precise.*" (E. Dijkstra)*

Indeed, working with too concrete models of reality is one of the main defects of much software that has been written (and is still to be written, unfortunately).

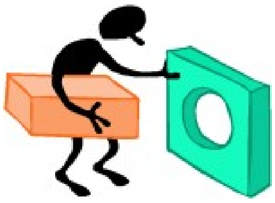
Is the colour of your students hair relevant for their assessment in your course? Surely not, no such column in the spreadsheet. But there you are: you've just lost one particular attribute of your students! This means that your spreadsheet already contains an *abstract* view of them...

The challenge is therefore to keep those attributes that are needed for the software to operate *and only those*. In some sense, programming is in this respect the *art of going abstract*!

COMPOSITIONALITY Many software *monsters* had been written already when, in 1967, Oettinger delivered his speech. The problem is that many, many more were created since, in spite of the availability of increasingly powerful languages with sophisticated abstraction mechanisms.

It is commonplace to say that today's programmers write poorly concurrent code. This is actually worse: they still write poorly structured *sequential* code because they were not trained in the *art of compositionality* early enough in their background. And so they find it hard to design a piece of software in terms of collaborative, small units, each doing its own job. Let alone other forms of composition in which such components operate concurrently, in a parallel way.

TYPE-ORIENTED PROGRAMMING How does one ensure that cooperating software components are compatible with each other and work without damaging one another? The obvious answer is: just in the same way as, for instance, one cannot physically connect a USB cable to a 220V outlet — the interfaces are physically different and do not allow it.



At the software level, such interfaces are known as *types* and the associated safety mechanism known as *type checking*. Type theory has developed immensely in recent years and is a fast evolving field of computer science nowadays. The morale is that every computation, piece of data etc. should have a *formal type*. Types permit (automatic) *checking* before *building*. Doing software without types is like doing *biology* without a post-Linnaean *taxonomy*. But beware: much software running today is still untyped or too weakly typed.

CONTRACT-ORIENTED PROGRAMMING In the meantime, static type checking has evolved from ensuring *static* types (at compile-time) to ensuring that desirable properties are maintained at run-time, leading to extended (or *dynamic*) type checking. Such dynamic types are known also as *contracts*. The advice is that, as in the regular functioning of any society, programming should be based on *formal contracts* [57, 41] validated by the underlying maths and supported by dedicated toolsets.

This discipline ensures safety and security and is essential to *safety-critical* equipment operation.

PARAMETRICITY AND SCALABILITY Adaptability is a much valued feature in classical engineering. Adaptable design is a paradigm that extends the intended utility of products and designs beyond the initial, intended setting. Adaptability aims at reusing the same "design" for the creation of different products.

The mathematical essence of software enables particularly interesting forms of adaptability. Software engineers faced with a "new" problem tend to reuse (by copying, pasting and modifying) previously developed code. Quite often, a component is copied, renamed and little more is needed. This means that the original code and the modified one are abstractly the same. For instance, determining the length of a list does not depend on the kind of elements kept in the list. Furthermore, length-of-a-list is but an instance of a more general problem, that of counting the number of items in a finite data structure, independently of its topology. This feature of software is termed *parameterization* and has been known for a long time [17].

Software developers should be trained to appreciate writing *parametric*, reusable code — that is, *generic* code which is automatically *instantiated* in particular situations. This is not only intellectually rewarding but also brings elegance and economy into programming. So-called *polymorphic types* do this automatically and there is a nice theory behind them called *parametric polymorphism* [47, 85]. So nice that one can derive *properties* of one's code even *before writing it* [55],

It is no wonder that parametricity promotes code reuse and makes software solutions more robust and truly scalable.

1.3 SUMMARY

The starting point of the section above was the realization that, with software taking over all fields of (what used to be solely) human activity, the challenges to be faced in the future by software designers are enormous. Insecurity, risk of malfunction/failure in increasingly complex systems will reach unprecedented levels.

This was followed by listing some technical ingredients for good software design that are regarded as useful to mitigate the complexity that lies ahead — considering software as a *product* and ignoring (intentionally) its development process, which is a distinct problem.

Even though the average software designer will agree on the relevance of such design principles, these tend to be overlooked or poorly adopted (if not at all ignored) by programmers lacking the required background knowledge.

Several decades of teaching experience of the author of this text suggest that there is a proper timing for acquiring such background knowledge: this should take place in the early phases of graduate education, at the same time students face other "theoretical subjects" in the fields of physics and maths, for instance. Doing it later is unfortunately too late. On the other hand, not every programming paradigm is suitable for such training. Again based on experience, it has become clear that *functional programming* (FP) provides an easier vehicle for students to understand and apply such basic principles. This calls for a *functional-first* academic syllabus.

A traditional obstacle to teaching FP in the past was the inefficiency of the functional languages available at the time. But, thanks to tremendous advances in the field in the last decades, this is no longer a (very serious) problem.

Another piece of resistance in the minds of students has been the feeling that FP is not main stream in industry. But this is changing steadily, with more and more software companies switching (at least in part) to FP in areas such as finance application design, data science or domains where high levels of safety and security are at target.

The main point of functional-first syllabi is that switching to *thinking functionally* can be a tremendous effort for students addicted to the atomicity of the one-instruction-at-a-time programming paradigm. By contrast, the converse switch from functional to imperative or object-oriented programming is not so demanding. FP calls for a good, overall perception of the whole picture and for a clear insight on how data flows throughout the designed artifact. Moreover, since functions can lose information, there is always a "proper place" in the pipeline for a particular function to be inserted.

Atomicity or structuralism also impacts the way programmers validate their software. FP promotes a "correct-by-construction" approach to programming that promises significant reduction in development costs by avoiding dependence on testing and debugging. Rather than *unit testing* focussing on particular components, FP tests are usually expressed in terms of desirable properties involving several functions at the same time. It is thus no wonder that tools such as e.g. QuickCheck started in the FP field [16].[5]

Structural design, data-flow awareness and less dependence on testing also promote a solid FP background towards new, emerging programming paradigms such as reversible and quantum computing [65]. For instance, while classical memory access does not harm the data, reading quantum data dramatically spoils the quantum effect. From this perspective, addiction to the edit-run-debug vicious development cycle can substantially reduce the proficiency of a "conventional" programming mind once faced with such new technologies.

Functional programming and FP calculi alone are, however, not enough. Problem specifications as a rule involve functions but *are* not functions in themselves. Thus, a strictly functional algebra of programs cannot fully bridge the specification-level to the implementation-level, compromising the "correct-by-construction" desideratum. There are two options there: either start from specifications expressed in first-order logic and somehow derive functional implementations from them; or extend the *pointfree* calculus of functions so that the specifications themselves can also be written in the same style.

[5] The shortcomings of testing are well-known. It is hopelessly inadequate in situations such as concurrency (where scenarios are often impossible to reproduce) and with very large input spaces (such as robots and autonomous cars with video sensors).

1.4 WHY THIS BOOK?

The well-known aphorism *"functions are special cases of relations"* is the main guideline for the second alternative — extend the calculus of functions towards a calculus of relations — and such is the principle behind texbook [12] and the current "book". This explains the two parts *Calculating with Functions* (part I) and *Calculating with Relations* (part II).

When the author of this draft textbook decided to teach the algebra of programming to 2nd year computer science students, back to 1998, he found textbook [12] too difficult for the students to follow, mainly because of its too explicit categorial (allegorical) flavour. So he decided to start writing slides and notes helping the students to read the book. Eventually, such notes became chapters 2 to 4 of the current version of this monograph. The same procedure was taken when teaching the relational approach of [12] at master level, see chapters 5 to 8.

Another motivation in complementing [12] is to give a wider perception of the usefulness of relational algebra in calculating software that is *data intensive*. Indeed, relations are as effective in describing algorithms as they are in describing data, which sanctions the "equation"

Algorithms + *Data Structures* = *Programs* [87]

with nice theoretical homogeneity. This homogeneity can be found in tools such as Alloy [37], for instance, which has been used in the lab part of the courses mentioned above along with Haskell.

There is a promised third part, *Calculating with Matrices* (part III) which will evolve naturally from the first two, heading towards *quantitative* aspects of software design. In the age of data mining and machine learning it is very important to bring type safety and calculational design to these subjects, and standard relational database theory "à la Codd" [51] is not enough, as is explained in [68] for instance. Typed linear algebra, which is also making a contribution to probabilistic programming [62] seems to be a promise in such direction, but some consolidation of this kowledge needs to happen before it can be brought into the classroom.

Part I

CALCULATING WITH FUNCTIONS

2

AN INTRODUCTION TO POINTFREE PROGRAMMING

Everybody is familiar with the concept of a *function* since the school desk. The functional intuition traverses mathematics from end to end because it has a solid semantics rooted on a well-known mathematical system — the class of "all" sets and set-theoretical functions.

Functional programming literally means "programming with functions". Programming languages such as LISP or HASKELL allow us to program with functions. However, the functional intuition is far more reaching than producing code which runs on a computer. Since the pioneering work of John McCarthy — the inventor of LISP — in the early 1960s, one knows that other branches of programming can be structured, or expressed functionally. The idea of producing programs by *calculation*, that is to say, that of calculating efficient programs out of abstract, inefficient ones has a long tradition in functional programming.

This book is structured around the idea that functional programming can be used as a basis for teaching programming as a whole, from the successor function $n \mapsto n+1$ to large information system design.[1]

This chapter provides a light-weight introduction to the theory of functional programming. The main emphasis is on *compositionality* — one of the main advantages of "thinking functionally" — by explaining how to construct new functions out of other functions using a minimal set of predefined functional *combinators*. This leads to a programming style that is *point free* in the sense that function descriptions dispense with variables (also known as *points*).

Several technical issues are deliberately ignored and deferred to later chapters. Most programming examples will be provided in the HASKELL functional programming language. Appendix B includes the listings of some HASKELL modules that complement the HASKELL *Standard Prelude* and help to "animate" the main concepts introduced in this chapter.

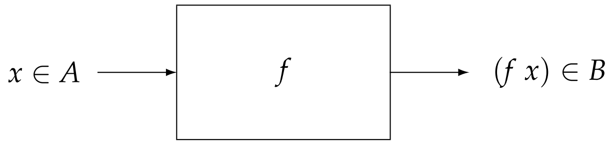
[1] This idea addresses programming in a broad sense, including for instance *reversible* and *quantum programming*, where functional programming already plays leading roles [61, 59, 30].

2.1 INTRODUCING FUNCTIONS AND TYPES

The definition of a function

$$f : A \to B \tag{2.1}$$

can be regarded as a kind of "process" abstraction: it is a "black box" which produces an output once it is supplied with an input:



The box isn't really necessary to convey the abstraction and a single labelled arrow suffices:

$$A \xrightarrow{\quad f \quad} B$$

This simplified notation focusses on what is indeed relevant about f — that it can be regarded as a kind of "contract":

> f *commits itself* to producing a B-value provided it is supplied with an A-value.

How is such a value produced? In many situations one wishes to ignore this because one is just *using* function f. In others, however, one may want to inspect the internals of the "black box" in order to know the function's *computation rule*. For instance,

$$\begin{array}{rcl} \mathsf{succ} & : & \mathbb{N} \to \mathbb{N} \\ \mathsf{succ}\ n & \overset{\mathrm{def}}{=} & n+1 \end{array}$$

expresses the computation rule of the *successor* function — the function succ which finds "the next natural number" — in terms of natural number addition and of natural number 1. What we above meant by a "contract" corresponds to the *signature* of the function, which is expressed by arrow $\mathbb{N} \to \mathbb{N}$ in the case of succ and which, by the way, can be shared by other functions, *e.g.* $sq\ n \overset{\mathrm{def}}{=} n^2$.

In programming terminology one says that succ and sq have the same "type". Types play a prominent rôle in functional programming (as they do in other programming paradigms). Informally, they provide the "glue", or interfacing material, for putting functions together to obtain more complex functions. Formally, a "type checking" discipline can be expressed in terms of compositional rules which check for functional expression wellformedness.

It has become standard to use arrows to denote function signatures or function types, recall (2.1). To denote the fact that function f accepts arguments of type A and produces results of type B, we will use the following interchangeable notations: $f\ :\ B \leftarrow A$, $f\ :\ A \to B$, $B \xleftarrow{\ f\ } A$

or $A \xrightarrow{f} B$. This corresponds to writing $f :: a \to b$ in the HASKELL functional programming language, where type variables are denoted by lowercase letters. A will be referred to as the *domain* of f and B will be referred to as the *codomain* of f. Both A and B are symbols or expressions which denote sets of values, most often called *types*.

2.2 FUNCTIONAL APPLICATION

What do we want functions for? If we ask this question to a physician or engineer the answer is very likely to be: one wants functions for modelling and reasoning about the behaviour of real things.

For instance, function $distance\ t = 60 \times t$ could be written by a school physics student to model the distance (in, say, kilometers) a car will drive (per hour) at average speed $60km/hour$. When questioned about how far the car has gone in 2.5 hours, such a model provides an immediate answer: just evaluate $distance\ 2.5$ to obtain $150km$.

So we get a naïve purpose of functions: we want them to be *applied* to arguments in order to obtain results. Functional *application* is denoted by juxtaposition, *e.g.* $f\ a$ for $B \xleftarrow{f} A$ and $a \in A$, and associates to the left: $f\ x\ y$ denotes $(f\ x)\ y$ rather than $f\ (x\ y)$.

2.3 FUNCTIONAL EQUALITY AND COMPOSITION

Application is not everything we want to do with functions. Very soon our physics student will be able to talk about properties of the *distance* model, for instance that (linear) property

$$distance\ (2 \times t) = 2 \times (distance\ t) \qquad (2.2)$$

holds. Later on, we could learn from her or him that the same property can be restated as $distance\ (twice\ t) = twice\ (distance\ t)$, by introducing function $twice\ x \stackrel{\rm def}{=} 2 \times x$. Or even simply as

$$distance \cdot twice = twice \cdot distance \qquad (2.3)$$

where "$\cdot$" denotes function-arrow chaining, as suggested by drawing

$$\begin{tikzcd}
\mathbb{R} \arrow[d, "distance"'] & \mathbb{R} \arrow[l, "twice"'] \arrow[d, "distance"] \\
\mathbb{R} & \mathbb{R} \arrow[l, "twice"]
\end{tikzcd} \qquad (2.4)$$

where both space and time are modelled by real numbers in $\mathbb{R}$.

This trivial example illustrates some relevant facets of the functional programming paradigm. Which version of the property presented above is "better"? the version explicitly mentioning variable t and requiring parentheses (2.2)? the version hiding variable t but resorting to function *twice* (2.3)? or even diagram (2.4) alone?

Expression (2.3) is clearly more compact than (2.2). The trend for notation economy and compactness is well-known throughout the history of mathematics. In the 16th century, for instance, algebrists would write *12.cu.p̃.18.ce.p̃.27.co.p̃.17* for what is nowadays written as $12x^3 + 18x^2 + 27x + 17$. We may find such *syncopated* notation odd, but we should not forget that at its time it was replacing even more obscure and lengthy expression denotations.

Why do people look for compact notations? A compact notation leads to shorter documents (less lines of code in programming) in which patterns are easier to identify and to reason about. Properties can be stated in clear-cut, one-line long equations which are easy to memorize. And diagrams such as (2.4) can be easily drawn which enable us to visualize maths in a graphical format.

Some people will argue that such a compact "pointfree" notation (that is, the notation that hides variables, or function "definition points") is too cryptic to be useful as a practical programming medium. In fact, pointfree programming languages such as Iverson's APL or Backus' FP have been more respected than loved by the programmers community. Virtually all commercial programming languages require variables and so implement the more traditional "pointwise" notation.

Throughout this book we will adopt both, depending upon the context. Our chosen programming medium — HASKELL — blends the pointwise and pointfree programming styles in a quite successful way. In order to switch from one to the other, we need two "bridges": one lifting equality to the functional level and the other lifting function application.

Concerning equality, note that the "=" sign in (2.2) differs from that in (2.3): while the former states that two real numbers are the same number, the latter states that two $\mathbb{R} \leftarrow \mathbb{R}$ functions are the same function. Formally, we will say that two functions $f, g : B \leftarrow A$ are equal if they agree at pointwise-level, that is[2]

$$f = g \quad \textit{iff} \quad \langle \forall\, a \;:\; a \in A : \; f\, a \;=_B\; g\, a \rangle \tag{2.5}$$

where $=_B$ denotes equality at B-level. Rule (2.5) is known as *extensional equality*.

Concerning application, the pointfree style replaces it by the more generic concept of functional *composition* suggested by function-arrow chaining: wherever two functions are such that the target type of one of them, say $B \xleftarrow{g} A$ is the same as the source type of the other, say $C \xleftarrow{f} B$, then another function can be defined, $C \xleftarrow{f \cdot g} A$, called the *composition* of f and g, or "f after g", which "glues" f and g together:

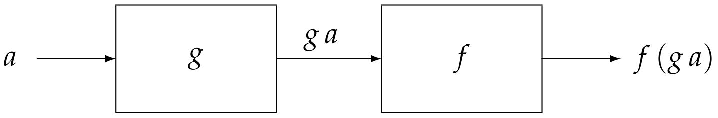
$$(f \cdot g)\, a \;=\; f\,(g\, a) \tag{2.6}$$

[2] Quantified notation $\langle \forall\, x \;:\; P \;:\; Q \rangle$ means: *"for all x in the range P, Q holds"*, where P and Q are logical expressions involving x. (See appendix A for more details.) This notation will be used sporadically in the first part of this book.

This situation is pictured by the following arrow-diagram

$$\begin{tikzcd}
B \arrow[d, "f"'] & A \arrow[l, "g"'] \arrow[dl, "f\cdot g"] \\
C &
\end{tikzcd} \tag{2.7}$$

or by block-diagram



Therefore, the type-rule associated to functional composition can be expressed as follows:[3]

$$\frac{\begin{array}{c} C \xleftarrow{f} B \\ B \xleftarrow{g} A \end{array}}{C \xleftarrow{f\cdot g} A}$$

Composition is certainly the most basic of all functional combinators. It is the first kind of "glue" that comes to mind when programmers need to combine, or chain functions (or processes) to obtain more elaborate functions (or processes).[4] This is because of one of its most relevant properties,

$$(f \cdot g) \cdot h \quad = \quad f \cdot (g \cdot h) \tag{2.8}$$

depicted by diagram

$$\begin{tikzcd}
& D \arrow[dl, "g\cdot h"'] \arrow[d, "h"] \\
B \arrow[d, "f"'] & A \arrow[l, "g"] \arrow[dl, "f\cdot g"] \\
C &
\end{tikzcd}$$

which shares the pattern of, for instance

$$(a + b) + c \quad = \quad a + (b + c)$$

and so is called the *associative* property of composition. This enables us to move parentheses around in pointfree expressions involving functional compositions, or even to omit them altogether, for instance by writing $f \cdot g \cdot h \cdot i$ as an abbreviation of $((f \cdot g) \cdot h) \cdot i$, or of $(f \cdot (g \cdot h)) \cdot i$, or of $f \cdot ((g \cdot h) \cdot i)$, *etc.* For a chain of n-many function compositions the notation $\bigcirc_{i=1}^{n} f_i$ will be acceptable as abbreviation of $f_1 \cdot \cdots \cdot f_n$.

[3] This and other type-rules to come adopt the usual "fractional" layout, reminiscent of that used in school arithmetics for addition, subtraction, etc.

[4] It even has a place in scripting languages such as UNIX's shell, where `f | g` is the shell counterpart of $g \cdot f$, for appropriate "processes" f and g.

2.4 IDENTITY FUNCTIONS

How free are we to fulfill the "give me an A and I will give you a B" contract of equation (2.1)? In general, the choice of f is not unique. Some fs will do as little as possible while others will laboriously compute non-trivial outputs. At one of the extremes, we find functions which "do nothing" for us, that is, the added-value of their output when compared to their input amounts to very little: $f\,a = a$. In this case $B = A$, of course, and f is said to be the *identity* function on A:

$$\begin{array}{rcl} id_A & : & A \leftarrow A \\ id_A\,a & \stackrel{\rm def}{=} & a \end{array} \tag{2.9}$$

Note that every type X "has" its identity id_X. Subscripts will be omitted wherever implicit in the context. For instance, the arrow notation $\mathbb{N} \xleftarrow{id} \mathbb{N}$ saves us from writing $id_{\mathbb{N}}$. So, we will often refer to "the" identity function rather than to "an" identity function.

How useful are identity functions? At first sight, they look fairly uninteresting. But the interplay between composition and identity, captured by the following equation,

$$f \cdot id = id \cdot f = f \tag{2.10}$$

will be appreciated later on. This property shares the pattern of, for instance,

$$a + 0 = 0 + a = a$$

This is why we say that *id* is the *unit* (*identity*) of composition. In a diagram, (2.10) looks like this:

$$\begin{tikzcd} A \arrow[d, "f"'] & A \arrow[l, "id"'] \arrow[d, "f"] \\ B & B \arrow[l, "id"] \end{tikzcd} \tag{2.11}$$

Note the graphical analogy of diagrams (2.4) and (2.11). The latter is interesting in the sense that it is *generic*, holding for every f. Diagrams of this kind are very common and express important (and rather 'natural') properties of functions, as we shall see further on.

2.5 CONSTANT FUNCTIONS

Opposite to the identity functions, which do not lose any information, we find functions which lose all (or almost all) information. Regardless of their input, the output of these functions is always the same value.

Let C be a nonempty data domain and let and $c \in C$. Then we define the *everywhere* c function as follows, for arbitrary A:

$$\begin{array}{rcl} \underline{c} & : & A \rightarrow C \\ \underline{c}\,a & \stackrel{\mathrm{def}}{=} & c \end{array} \tag{2.12}$$

The following property defines constant functions at pointfree level,

$$\underline{c} \cdot f \quad = \quad \underline{c} \tag{2.13}$$

and is depicted by a diagram similar to (2.11):

$$\begin{tikzcd} C \arrow[d, "id"'] & A \arrow[l, "\underline{c}"'] \arrow[d, "f"] \\ C & B \arrow[l, "\underline{c}"] \end{tikzcd} \tag{2.14}$$

Clearly, $\underline{c} \cdot f = \underline{c} \cdot g$, for any f, g, meaning that any difference that may exist in behaviour between such functions is lost.

Note that, strictly speaking, symbol $\underline{c}$ denotes two different functions in diagram (2.14): one, which we should have written $\underline{c}_A$, accepts inputs from A while the other, which we should have written $\underline{c}_B$, accepts inputs from B:

$$\underline{c}_B \cdot f \quad = \quad \underline{c}_A \tag{2.15}$$

This property will be referred to as the constant-*fusion* property.

As with identity functions, subscripts will be omitted wherever implicit in the context.

Exercise *2.1.Use (2.5) to show that $f \cdot h = h \cdot f = f$ has the unique solution $h = id$, cf. (2.10).*[5]

□

Exercise *2.2. The* HASKELL Prelude *provides for constant functions: you write* const c *for* $\underline{c}$. *Check that* HASKELL *assigns the same type to expressions* $f \cdot (\mathsf{const}\ c)$ *and* $\mathsf{const}\ (f\ c)$, *for every f and c. What else can you say about these functional expressions? Justify.*[6]

□

---

[5] This textbook follows the convention that free variables such as f above are always assumed universally quantified. Thus what is to be shown above is: $\langle \forall\, h,f\ ::\ f \cdot h = h \cdot f = f\ \Leftrightarrow\ h = id \rangle$.

[6] The properties of Haskell's higher order function const will be dealt with in more detail later in exercise 6.2.

2.6 MONICS AND EPICS

Identity functions and constant functions are limit points of the functional spectrum with respect to information preservation. All the other functions are in between: they lose "some" information, which is regarded as uninteresting for some reason. This remark supports the following aphorism about a facet of functional programming: it is the *art* of transforming or losing information in a controlled and precise way. That is to say, the art of constructing the exact observation of data which fits in a particular context or requirement.

How do functions lose information? Basically in two different ways: they may be "blind" enough to confuse different inputs, by mapping them onto the same output, or they may ignore values of their codomain. For instance, $\underline{c}$ confuses *all* inputs by mapping them all onto c. Moreover, it ignores all values of its codomain apart from c.

Functions which do not confuse inputs are called *monics* (or *injective* functions) and obey the following property: $B \xleftarrow{f} A$ is *monic* if, for every pair of functions $A \xleftarrow{h,k} C$, if $f \cdot h = f \cdot k$ then $h = k$, *cf.* diagram

$$\begin{tikzcd} B & A \arrow[l, "f"'] & C \arrow[l, shift right, "h"'] \arrow[l, shift left, "k"] \end{tikzcd}$$

(we say that f is "post-cancellable"). It is easy to check that "the" identity function is monic, since

$$\begin{array}{cl} & id \cdot h = id \cdot k \\ \equiv & \qquad \{ \ id \text{ is unit of composition (2.10)} \ \} \\ & h = k \end{array}$$

which trivializes $id \cdot h = id \cdot k \Rightarrow h = k$. By contrast, any constant function $\underline{c}$ is not monic:

$$\begin{array}{cl} & \underline{c} \cdot h = \underline{c} \cdot k \Rightarrow h = k \\ \equiv & \qquad \{ \text{ by (2.15) } \} \\ & \underline{c} = \underline{c} \Rightarrow h = k \\ \equiv & \qquad \{ \text{ function equality is reflexive } \} \\ & \text{TRUE} \Rightarrow h = k \\ \equiv & \qquad \{ \text{ predicate logic } \} \\ & h = k \end{array}$$

So the implication does not hold in general (only if $h = k$).

Functions which do not ignore values of their codomain are called *epics* (or *surjective* functions) and obey the following property: $A \xleftarrow{f} B$

is *epic* if, for every pair of functions $C \xleftarrow{h,k} A$, if $h \cdot f = k \cdot f$ then $h = k$, *cf.* diagram

$$\begin{tikzcd} C & A \arrow[l, shift right, "k"'] \arrow[l, shift left, "h"] & B \arrow[l, "f"] \end{tikzcd}$$

(we say that f is "pre-cancellable"). As expected, identity functions are epic:

$$\begin{array}{ll} & h \cdot id = k \cdot id \Rightarrow h = k \\ \equiv & \qquad \{ \text{ by (2.10) } \} \\ & h = k \Rightarrow h = k \\ \equiv & \qquad \{ \text{ predicate logic } \} \\ & \text{TRUE} \end{array}$$

Exercise *2.3. Under what circumstances is a constant function epic? Justify.*
□

2.7 ISOS

A function $B \xleftarrow{f} A$ which is both monic and epic is said to be *iso* (an isomorphism, or a bijective function). In this situation, f always has a *converse* (or *inverse*) $B \xrightarrow{f^\circ} A$, which is such that

$$f \cdot f^\circ = id_B \quad \wedge \quad f^\circ \cdot f = id_A \tag{2.16}$$

(*i.e.* f is *invertible*).

Isomorphisms are very important functions because they convert data from one "format", say A, to another format, say B, without losing information. So f and and f° are faithful protocols between the two formats A and B. Of course, these formats contain the same "amount" of information, although the same data adopts a different "shape" in each of them. In mathematics, one says that A is *isomorphic* to B and one writes $A \cong B$ to express this fact.

Isomorphic data domains are regarded as "abstractly" the same. Note that, in general, there is a wide range of isos between two isomorphic data domains. For instance, let *Weekday* be the set of weekdays,

$$\begin{array}{l} Weekday = \\ \quad \{ Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday \} \end{array}$$

and let symbol 7 denote the set $\{1, 2, 3, 4, 5, 6, 7\}$, which is the *initial segment* of $\mathbb{N}$ containing exactly seven elements. The following function f, which associates each weekday with its "ordinal" number,

$$\begin{array}{l}
f : \mathit{Weekday} \rightarrow 7 \\
f\ \mathit{Monday} = 1 \\
f\ \mathit{Tuesday} = 2 \\
f\ \mathit{Wednesday} = 3 \\
f\ \mathit{Thursday} = 4 \\
f\ \mathit{Friday} = 5 \\
f\ \mathit{Saturday} = 6 \\
f\ \mathit{Sunday} = 7
\end{array}$$

is iso (guess f°). Clearly, $f\ d = i$ means "d is the i-th day of the week". But note that function $g\ d \overset{\mathrm{def}}{=} \mathit{rem}(f\ d, 7) + 1$ is also an iso between Weekday and 7. While f regards *Monday* the first day of the week, g places *Sunday* in that position. Both f and g are witnesses of isomorphism

$$\mathit{Weekday} \;\cong\; 7 \tag{2.17}$$

Isomorphisms are quite flexible in pointfree reasoning. If, for some reason, f° is found handier than isomorphism f in the reasoning, then the shunting rules

$$f \cdot g = h \;\equiv\; g = f^\circ \cdot h \tag{2.18}$$

$$g \cdot f = h \;\equiv\; g = h \cdot f^\circ \tag{2.19}$$

can be of help.

Finally, note that all classes of functions referred to so far — constants, identities, epics, monics and isos — are closed under composition, that is, the composition of two constants is a constant, the composition of two epics is epic, *etc.*

2.8 GLUING FUNCTIONS WHICH DO NOT COMPOSE — PRODUCTS

Function composition has been presented above as a basis for gluing functions together in order to build more complex functions. However, not every two functions can be glued together by composition. For instance, functions $f : A \leftarrow C$ and $g : B \leftarrow C$ do not compose with each other because the domain of one of them is not the codomain of the other. However, both f and g share the same domain C. So, something we can do about gluing f and g together is to draw a diagram expressing this fact, something like

$$\begin{tikzcd}
A & & B \\
& C \arrow[ul, "f"] \arrow[ur, "g"'] &
\end{tikzcd}$$

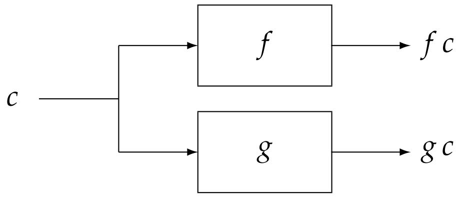
Because f and g share the same domain, their outputs can be paired, that is, we may write ordered pair $(f\ c, g\ c)$ for each $c \in C$. Such pairs belong to the Cartesian product of A and B, that is, to the set

$$A \times B \;\overset{\mathrm{def}}{=}\; \{(a, b) \mid a \in A \wedge b \in B\}$$

So we may think of the operation which pairs the outputs of f and g as a new function combinator $\langle f, g\rangle$ defined as follows:

$$\begin{array}{lcl} \langle f,g\rangle & : & C \to A \times B \\ \langle f,g\rangle\, c & \overset{\mathrm{def}}{=} & (f\,c, g\,c) \end{array} \tag{2.20}$$

Traditionally, the pairing combinator $\langle f, g\rangle$ is pronounced "*f split g*" (or "pair f and g") and can be depicted by the following "block", or "data flow" diagram:



Function $\langle f, g\rangle$ keeps the information of both f and g in the same way Cartesian product $A \times B$ keeps the information of A and B. So, in the same way A data or B data can be retrieved from $A \times B$ data via the implicit *projections* π_1 or π_2,

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B \tag{2.21}$$

defined by

$$\pi_1(a,b) = a \quad \text{and} \quad \pi_2(a,b) = b$$

f and g can be retrieved from $\langle f, g\rangle$ via the same projections:

$$\pi_1 \cdot \langle f,g\rangle = f \quad \text{and} \quad \pi_2 \cdot \langle f,g\rangle = g \tag{2.22}$$

This fact (or pair of facts) will be referred to as the $\times$-*cancellation* property and is illustrated in the following diagram which puts everything together:

$$\begin{tikzcd} A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\ & C \arrow[ul, "f"] \arrow[u, "\langle f,g\rangle"'] \arrow[ur, "g"'] & \end{tikzcd} \tag{2.23}$$

In summary, the type-rule associated to the "split" combinator is expressed by

$$\frac{\begin{array}{c} A \xleftarrow{f} C \\ B \xleftarrow{g} C \end{array}}{A \times B \xleftarrow{\langle f,g\rangle} C}$$

A *split* arises wherever two functions do not compose but share the same domain. What about gluing two functions which fail such a requisite, *e.g.*

$$\begin{array}{c} A \xleftarrow{f} C \\ B \xleftarrow{g} D \\ \hline \ldots ? \end{array}$$

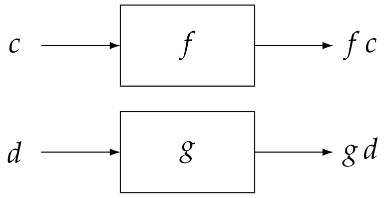
The $\langle f, g \rangle$ *split* combination does not work any more. Nevertheless, a way to "bridge" the domains of f and g, C and D respectively, is to regard them as targets of the projections π_1 and π_2 of $C \times D$:

$$\begin{tikzcd}
A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
C \arrow[u, "f"] & C \times D \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & D \arrow[u, "g"']
\end{tikzcd}$$

From this diagram $\langle f \cdot \pi_1, g \cdot \pi_2 \rangle$ arises

$$\begin{tikzcd}
A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
& C \times D \arrow[ul, "f \cdot \pi_1"] \arrow[u, "\langle f \cdot \pi_1, g \cdot \pi_2 \rangle"] \arrow[ur, "g \cdot \pi_2"'] &
\end{tikzcd}$$

mapping $C \times D$ to $A \times B$. It corresponds to the "parallel" application of f and g which is suggested by the following data-flow diagram:



Functional combination $\langle f \cdot \pi_1, g \cdot \pi_2 \rangle$ appears so often that it deserves special notation — it will be expressed by $f \times g$. So, by definition, we have

$$f \times g \overset{\mathrm{def}}{=} \langle f \cdot \pi_1, g \cdot \pi_2 \rangle \tag{2.24}$$

which is pronounced "product of f and g" and has typing-rule

$$\begin{array}{c} A \xleftarrow{f} C \\ B \xleftarrow{g} D \\ \hline A \times B \xleftarrow{f \times g} C \times D \end{array} \tag{2.25}$$

Note the overloading of symbol "×", which is used to denote both Cartesian product and functional product. This choice of notation will be fully justified later on.

What is the interplay among functional combinators $f \cdot g$ (composition), $\langle f, g\rangle$ (*split*) and $f \times g$ (product) ? Composition and *split* relate to each other via the following property, known as $\times$-*fusion*:[7]

$$\begin{tikzcd}
A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
& C \arrow[ul, "g"] \arrow[u, "\langle g,h \rangle"'] \arrow[ur, "h"'] & \\
& D \arrow[u, "f"] \arrow[uul, "g \cdot f"] \arrow[uur, "h \cdot f"'] &
\end{tikzcd}$$

$$\langle g, h\rangle \cdot f = \langle g \cdot f, h \cdot f\rangle \tag{2.26}$$

This shows that *split* is right-distributive with respect to composition. Left-distributivity does not hold but there is something we can say about $f \cdot \langle g, h\rangle$ in case $f = i \times j$:

$$\begin{array}{ll}
& (i \times j) \cdot \langle g, h\rangle \\
= & \qquad \{ \text{ by (2.24) } \} \\
& \langle i \cdot \pi_1, j \cdot \pi_2\rangle \cdot \langle g, h\rangle \\
= & \qquad \{ \text{ by } \times\text{-fusion (2.26) } \} \\
& \langle (i \cdot \pi_1) \cdot \langle g, h\rangle, (j \cdot \pi_2) \cdot \langle g, h\rangle\rangle \\
= & \qquad \{ \text{ by (2.8) } \} \\
& \langle i \cdot (\pi_1 \cdot \langle g, h\rangle), j \cdot (\pi_2 \cdot \langle g, h\rangle)\rangle \\
= & \qquad \{ \text{ by } \times\text{-cancellation (2.22) } \} \\
& \langle i \cdot g, j \cdot h\rangle
\end{array}$$

The law we have just derived is known as $\times$-*absorption*. (The intuition behind this terminology is that "*split* absorbs $\times$", as a special kind of fusion.) It is a consequence of $\times$-fusion and $\times$-cancellation and is depicted as follows:

$$\begin{tikzcd}
A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
D \arrow[u, "i"] & D \times E \arrow[u, "i \times j"] \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & E \arrow[u, "j"'] \\
& C \arrow[ul, "g"] \arrow[u, "\langle g,h \rangle"'] \arrow[ur, "h"'] &
\end{tikzcd}$$

$$(i \times j) \cdot \langle g, h\rangle = \langle i \cdot g, j \cdot h\rangle \tag{2.27}$$

This diagram provides us with two further results about products and projections which can be easily justified:

$$i \cdot \pi_1 = \pi_1 \cdot (i \times j) \tag{2.28}$$

$$j \cdot \pi_2 = \pi_2 \cdot (i \times j) \tag{2.29}$$

Two special properties of $f \times g$ are presented next. The first one expresses a kind of "bi-distribution" of $\times$ with respect to composition:

$$(g \cdot h) \times (i \cdot j) = (g \times i) \cdot (h \times j) \tag{2.30}$$

[7] Note how this law can be regarded as a pointfree rendering of (2.20).

We will refer to this property as the $\times$*-functor property*. The other property, which we will refer to as the $\times$*-functor-id property*, has to do with identity functions:

$$id_A \times id_B \;=\; id_{A\times B} \tag{2.31}$$

These two properties will be identified as the *functorial properties* of product. Once again, this choice of terminology will be explained later on.

Let us finally analyse the particular situation in which a *split* is built involving projections π_1 and π_2 only. These exhibit interesting properties, for instance $\langle \pi_1, \pi_2 \rangle = id$. This property is known as $\times$*-reflection* and is depicted as follows:[8]

$$\begin{tikzcd}
A & A\times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
& A\times B \arrow[ul, "\pi_1"] \arrow[u, "id_{A\times B}"'] \arrow[ur, "\pi_2"'] &
\end{tikzcd}
\qquad \langle \pi_1, \pi_2 \rangle = id_{A\times B} \tag{2.32}$$

What about $\langle \pi_2, \pi_1 \rangle$? This corresponds to a diagram

$$\begin{tikzcd}
B & B\times A \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & A \\
& A\times B \arrow[ul, "\pi_2"] \arrow[u, "\langle \pi_2, \pi_1 \rangle"'] \arrow[ur, "\pi_1"'] &
\end{tikzcd}$$

which looks very much the same if submitted to a 180^o clockwise rotation (thus A and B swap with each other). This suggests that swap — the name we adopt for $\langle \pi_2, \pi_1 \rangle$ — is its own inverse; this is checked easily as follows:

$$\begin{array}{cl}
& \mathsf{swap} \cdot \mathsf{swap} \\
= & \qquad \{ \text{ by definition } \mathsf{swap} \overset{\text{def}}{=} \langle \pi_2, \pi_1 \rangle \ \} \\
& \langle \pi_2, \pi_1 \rangle \cdot \mathsf{swap} \\
= & \qquad \{ \text{ by } \times\text{-fusion (2.26) } \} \\
& \langle \pi_2 \cdot \mathsf{swap}, \pi_1 \cdot \mathsf{swap} \rangle \\
= & \qquad \{ \text{ definition of } \mathsf{swap} \text{ twice } \} \\
& \langle \pi_2 \cdot \langle \pi_2, \pi_1 \rangle, \pi_1 \cdot \langle \pi_2, \pi_1 \rangle \rangle \\
= & \qquad \{ \text{ by } \times\text{-cancellation (2.22) } \} \\
& \langle \pi_1, \pi_2 \rangle \\
= & \qquad \{ \text{ by } \times\text{-reflection (2.32) } \} \\
& id
\end{array}$$

Therefore, swap is iso and establishes the following isomorphism

$$A \times B \;\cong\; B \times A \tag{2.33}$$

[8] For an explanation of the word "*reflection*" in the name chosen for this law (and for others to come) see section 2.13 later on.

which is known as the *commutative property* of product.

The "product datatype" $A \times B$ is essential to information processing and is available in virtually every programming language. In HASKELL one writes (A, B) to denote $A \times B$, for A and B two predefined datatypes, fst to denote π_1 and snd to denote π_2. In the C programming language this datatype is called the "struct datatype",

```
struct {
  A first;
  B second;
};
```

while in PASCAL it is called the "record datatype":

```
record
  first : A;
  second : B
end
```

Isomorphism (2.33) can be re-interpreted in this context as a guarantee that *one does not lose (or gain) anything in swapping fields in record datatypes*. C or PASCAL programmers know also that record-field nesting has the same status, that is to say that, for instance, datatype

```
record
  f : A;
  s : record
    f : B;
    s : C;
    end
end;
```

is abstractly the same as

```
record
  f : record
    f : A;
    s : B
    end;
  s : C;
end;
```

In fact, this is another well-known isomorphism, known as the *associative property* of product:

$$A \times (B \times C) \quad \cong \quad (A \times B) \times C \tag{2.34}$$

This is established by $A \times (B \times C) \xleftarrow{assocr} (A \times B) \times C$, which is pronounced "associate to the right" and is defined by

$$\mathsf{assocr} \quad \stackrel{\mathrm{def}}{=} \quad \langle \pi_1 \cdot \pi_1, \pi_2 \times id \rangle \tag{2.35}$$

Appendix B lists an extension to the HASKELL *Standard Prelude* that makes isomorphisms such as swap and assocr available. In this module, the concrete syntax chosen for $\langle f, g \rangle$ is `split f g` and the one chosen for $f \times g$ is `f >< g`.

***Exercise** 2.4. Rely on (2.24) to prove properties (2.30) and (2.31).*
□

2.9 GLUING FUNCTIONS WHICH DO NOT COMPOSE — COPRODUCTS

The *split* functional combinator arose in the previous section as a kind of glue for combining two functions which do not compose but share the same domain. The "dual" situation of two non-composable functions $f : C \leftarrow A$ and $g : C \leftarrow B$ which however share the same codomain is depicted in

$$\begin{tikzcd}
A \arrow[dr, "f"'] & & B \arrow[dl, "g"] \\
& C &
\end{tikzcd}$$

It is clear that the kind of glue we need in this case should make it possible to apply f in case we are on the "A-side" or to apply g in case we are on the "B-side" of the diagram. Let us write $[f, g]$ to denote the new kind of combinator. Its codomain will be C. What about its domain?

We need to describe the datatype which is "either an A or a B". Since A and B are sets, we may think of $A \cup B$ as such a datatype. This works in case A and B are disjoint sets, but wherever the intersection $A \cap B$ is non-empty it is undecidable whether a value $x \in A \cap B$ is an "A-value" or a "B-value". In the limit, if $A = B$ then $A \cup B = A = B$, that is to say, we have not invented a new datatype at all. These difficulties can be circumvented by resorting to *disjoint union*,

$$A + B \overset{\text{def}}{=} \{ i_1\, a \mid a \in A \} \cup \{ i_2\, b \mid b \in B \}$$

assuming the "tagging" functions

$$i_1\, a = (t_1, a) \quad , \quad i_2\, b = (t_2, b) \tag{2.36}$$

with types[9] $A \xrightarrow{i_1} A + B \xleftarrow{i_2} B$. Knowing the exact values of tags t_1 and t_2 is not essential to understanding the concept of a disjoint union. It suffices to know that i_1 and i_2 tag differently ($t_1 \neq t_2$) and consistently.

[9] The tagging functions i_1 and i_2 are usually referred to as the *injections* of the disjoint union.

The values of $A + B$ can be thought of as "copies" of A or B values which are "stamped" with different tags in order to guarantee that values which are simultaneously in A and B do not get mixed up. For instance, the following realizations of $A + B$ in the C programming language,

```
struct {
  int tag; /*1,2 */
  union {
    A ifA;
    B ifB;
  } data;
};
```

or in PASCAL,

```
record
  case tag : integer
  of x =
    1 : (P : A);
    2 : (S : B)
 end;
```

adopt integer tags. In the HASKELL *Standard Prelude*, the $A + B$ datatype is realized by

```
data Either a b = Left a | Right b
```

So, `Left` and `Right` can be thought of as the injections i_1 and i_2 in this realization.

At this level of abstraction, disjoint union $A + B$ is called the *coproduct* of A and B, on top of which we define the new combinator $[f, g]$ (pronounced "either f or g") as follows:

$$\begin{array}{lcl} [f,g] & : & A + B \longrightarrow C \\ [f,g]\,x & \overset{\text{def}}{=} & \left\{ \begin{array}{lcl} x = i_1\,a & \Rightarrow & f\,a \\ x = i_2\,b & \Rightarrow & g\,b \end{array} \right. \end{array} \tag{2.37}$$

As we did for products, we can express all this in a diagram:

$$\begin{tikzcd} A \arrow[r, "i_1"] \arrow[dr, "f"'] & A + B \arrow[d, "{[f,g]}"] & B \arrow[l, "i_2"'] \arrow[dl, "g"] \\ & C & \end{tikzcd} \tag{2.38}$$

It is interesting to note how similar this diagram is to the one drawn for products — one just has to reverse the arrows, replace projections by injections and the *split* arrow by the *either* one. This expresses the fact that *product* and *coproduct* are *dual* mathematical constructs (compare with *sine* and *cosine* in trigonometry). This duality is of great conceptual economy because everything we can say about product $A \times B$

can be rephrased to coproduct $A + B$. For instance, we may introduce the sum of two functions $f + g$ as the notion dual to product $f \times g$:

$$f + g \stackrel{\mathrm{def}}{=} [i_1 \cdot f , i_2 \cdot g] \tag{2.39}$$

The following list of $+$-laws provides eloquent evidence of this duality:

$+$-cancellation :

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "g"'] & A+B \arrow[d, "{[g,h]}"] & B \arrow[l, "i_2"'] \arrow[dl, "h"] \\
& C &
\end{tikzcd}
\qquad [g,h] \cdot i_1 = g \ , \ [g,h] \cdot i_2 = h \tag{2.40}$$

$+$-reflection :

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "i_1"'] & A+B \arrow[d, "id_{A+B}"] & B \arrow[l, "i_2"'] \arrow[dl, "i_2"] \\
& A+B &
\end{tikzcd}
\qquad [i_1 , i_2] = id_{A+B} \tag{2.41}$$

$+$-fusion :

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "g"] \arrow[ddr, "f \cdot g"'] & A+B \arrow[d, "{[g,h]}"] & B \arrow[l, "i_2"'] \arrow[dl, "h"'] \arrow[ddl, "f \cdot h"] \\
& C \arrow[d, "f"] & \\
& D &
\end{tikzcd}
\qquad f \cdot [g,h] = [f \cdot g , f \cdot h] \tag{2.42}$$

$+$-absorption :

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[d, "i"'] & A+B \arrow[d, "i+j"'] & B \arrow[l, "i_2"'] \arrow[d, "j"] \\
D \arrow[r, "i_1"] \arrow[dr, "g"'] & D+E \arrow[d, "{[g,h]}"] & E \arrow[l, "i_2"] \arrow[dl, "h"] \\
& C &
\end{tikzcd}
\qquad [g,h] \cdot (i + j) = [g \cdot i , h \cdot j] \tag{2.43}$$

$+$-functor :

$$(g \cdot h) + (i \cdot j) = (g + i) \cdot (h + j) \tag{2.44}$$

$+$-functor-id :

$$id_A + id_B = id_{A+B} \tag{2.45}$$

In summary, the typing-rules of the *either* and *sum* combinators are as follows:

$$\frac{C \xleftarrow{f} A \qquad C \xleftarrow{g} B}{C \xleftarrow{[f,g]} A+B} \qquad \frac{C \xleftarrow{f} A \qquad D \xleftarrow{g} B}{C+D \xleftarrow{f+g} A+B} \tag{2.46}$$

Exercise *2.5. By analogy (duality) with* swap, *show that* $[i_2, i_1]$ *is its own inverse and so that fact*

$$A+B \quad \cong \quad B+A \tag{2.47}$$

holds.

□

Exercise *2.6. Dualize (2.35), that is, write the iso which witnesses fact*

$$A+(B+C) \quad \cong \quad (A+B)+C \tag{2.48}$$

from right to left. Use the `either` *syntax available from the* HASKELL Standard Prelude *to encode this iso in* HASKELL.

□

2.10 MIXING PRODUCTS AND COPRODUCTS

Datatype constructions $A \times B$ and $A + B$ have been introduced above as devices required for expressing the codomain of *splits* ($A \times B$) or the domain of *eithers* ($A + B$). Therefore, a function mapping values of a coproduct (say $A + B$) to values of a product (say $A' \times B'$) can be expressed alternatively as an *either* or as a *split*. In the first case, both components of the *either* combinator are *splits*. In the latter, both components of the *split* combinator are *eithers*.

This exchange of format in defining such functions is known as the *exchange law*. It states the functional equality which follows:

$$[\langle f,g\rangle, \langle h,k\rangle] \quad = \quad \langle [f,h], [g,k]\rangle \tag{2.49}$$

It can be checked by type-inference that both the left-hand side and the right-hand side expressions of this equality have type $B \times D \leftarrow A + C$, for $B \xleftarrow{f} A$, $D \xleftarrow{g} A$, $B \xleftarrow{h} C$ and $D \xleftarrow{k} C$.

An example of a function which is in the exchange-law format is isomorphism

$$A \times (B+C) \xleftarrow{\mathsf{undistr}} (A \times B) + (A \times C) \tag{2.50}$$

(pronounce $\mathsf{undistr}$ as "un-distribute-right") which is defined by

$$\mathsf{undistr} \stackrel{\mathrm{def}}{=} [id \times i_1 , id \times i_2] \tag{2.51}$$

and witnesses the fact that product distributes through coproduct:

$$A \times (B + C) \;\cong\; (A \times B) + (A \times C) \tag{2.52}$$

In this context, suppose that we know of three functions $D \xleftarrow{f} A$, $E \xleftarrow{g} B$ and $F \xleftarrow{h} C$. By (2.46) we infer $E + F \xleftarrow{g+h} B + C$. Then, by (2.25) we infer

$$D \times (E + F) \xleftarrow{f \times (g+h)} A \times (B + C) \tag{2.53}$$

So, it makes sense to combine products and sums of functions and the expressions which denote such combinations have the same "shape" (or symbolic pattern) as the expressions which denote their domain and range — the $\ldots \times (\cdots + \cdots)$ "shape" in this example. In fact, if we *abstract* such a pattern via some symbol, say F — that is, if we define

$$\mathsf{F}(\alpha, \beta, \gamma) \stackrel{\mathrm{def}}{=} \alpha \times (\beta + \gamma)$$

— then we can write $\mathsf{F}(D, E, F) \xleftarrow{\mathsf{F}(f,g,h)} \mathsf{F}(A, B, C)$ for (2.53).

This kind of abstraction works for every combination of products and coproducts. For instance, if we now abstract the right-hand side of (2.50) via pattern

$$\mathsf{G}(\alpha, \beta, \gamma) \stackrel{\mathrm{def}}{=} (\alpha \times \beta) + (\alpha \times \gamma)$$

we have $\mathsf{G}(f, g, h) = (f \times g) + (f \times h)$, a function which maps $\mathsf{G}(A, B, C) = (A \times B) + (A \times C)$ onto $\mathsf{G}(D, E, F) = (D \times E) + (D \times F)$. All this can be put in a diagram

$$\begin{tikzcd}
\mathsf{F}(A,B,C) \arrow[d, "\mathsf{F}(f,g,h)"'] & \mathsf{G}(A,B,C) \arrow[l, "\mathsf{undistr}"'] \arrow[d, "\mathsf{G}(f,g,h)"] \\
\mathsf{F}(D,E,F) & \mathsf{G}(D,E,F)
\end{tikzcd}$$

which unfolds to

$$\begin{tikzcd}
A \times (B + C) \arrow[d, "f \times (g+h)"'] & (A \times B) + (A \times C) \arrow[l, "\mathsf{undistr}"'] \arrow[d, "(f \times g) + (f \times h)"] \\
D \times (E + F) & (D \times E) + (D \times F)
\end{tikzcd} \tag{2.54}$$

once the F and G patterns are instantiated. An interesting topic which stems from (completing) this diagram will be discussed in the next section.

***Exercise** 2.7. Apply the* exchange law *to* undistr.

□

***Exercise** 2.8. Complete the "?"s in diagram*

$$\begin{tikzcd}
 & ? \arrow[dl, "{[x,y]}"'] \arrow[d, "id+id\times f"] \\
? & ? \arrow[l, "{[k,g]}"]
\end{tikzcd}$$

and then solve the implicit equation for x and y.
□

***Exercise** 2.9. Repeat exercise 2.8 with respect to diagram*

$$\begin{tikzcd}
? \arrow[r, "h+\langle i,j\rangle"] \arrow[dr, "x+y"'] & ? \arrow[d, "id+id\times f"] \\
 & ?
\end{tikzcd}$$

□

***Exercise** 2.10. Show that $\langle [f,h]\cdot(\pi_1+\pi_1), [g,k]\cdot(\pi_2+\pi_2)\rangle$ reduces to $[f\times g, h\times k]$.*
□

2.11 ELEMENTARY DATATYPES

So far we have talked mostly about arbitrary datatypes represented by capital letters A, B, *etc.* (lowercase `a`, `b`, *etc.* in the HASKELL illustrations). We also mentioned $\mathbb{R}$, $\mathbb{B}$ and $\mathbb{N}$ and, in particular, the fact that we can associate to each natural number n its *initial segment* $\mathsf{n} = \{1, 2, \ldots, n\}$. We extend this to $\mathbb{N}_0$ by stating $\mathsf{0} = \{\}$ and, for $n > 0$, $\mathsf{n+1} = \{n+1\} \cup \mathsf{n}$.

Initial segments can be identified with enumerated types and are regarded as primitive datatypes in our notation. We adopt the convention that primitive datatypes are written in the *sans serif* font and so, strictly speaking, n is distinct from n: the latter denotes a natural number while the former denotes a datatype.

Datatype 0

Among such enumerated types, $\mathsf{0}$ is the smallest because it is empty. This is the `Void` datatype in HASKELL, which has no constructor at all.

Datatype 0 (which we tend to write simply as 0) may not seem very "useful" in practice but it is of theoretical interest. For instance, it is easy to check that the following "obvious" properties hold,

$$A + 0 \cong A \tag{2.55}$$

$$A \times 0 \cong 0 \tag{2.56}$$

where the second is actually an equality: $A \times 0 = 0$.

Datatype 1

Next in the sequence of initial segments we find 1, which is singleton set $\{1\}$. How useful is this datatype? Note that every datatype A containing exactly one element is isomorphic to $\{1\}$, *e.g.* $A = \{\text{NIL}\}$, $A = \{0\}$, $A = \{1\}$, $A = \{\text{FALSE}\}$, *etc.*. We represent this class of singleton types by 1.

Recall that isomorphic datatypes have the same expressive power and so are "abstractly identical". So, the actual choice of inhabitant for datatype 1 is irrelevant, and we can replace any particular singleton set by another without losing information. This is evident from the following, observing isomorphism,

$$A \times 1 \cong A \tag{2.57}$$

which can be read informally as follows: if the second component of a record ("struct") cannot change, then it is useless and can be ignored. Selector π_1 is, in this context, an iso mapping the left-hand side of (2.57) to its right-hand side. Its inverse is $\langle id, \underline{c} \rangle$ where c is a particular choice of inhabitant for datatype 1.

In summary, when referring to datatype 1 we will mean an arbitrary singleton type, and there is a unique iso (and its inverse) between two such singleton types. The HASKELL representative of 1 is datatype `()`, called the *unit type*, which contains exactly constructor `()`. It may seem confusing to denote the type and its unique inhabitant by the same symbol but it is not, since HASKELL keeps track of types and constructors in separate symbol sets.

Note that any function of type $A \rightarrow 1$ is bound to be a constant function. This function, usually called the "bang", or "sink" function, is denoted by an exclamation mark:

$$A \xrightarrow{!} 1 \tag{2.58}$$

Clearly, it is *the unique* function of its type. (Can you write a different one, of the same type?)

Finally, what can we say about $1 + A$? Every function $B \xleftarrow{f} 1 + A$ observing this type is bound to be an *either* $[\underline{b_0}, g]$ for $b_0 \in B$ and $B \xleftarrow{g} A$. This is very similar to the handling of a pointer in C or PASCAL: we "pull a rope" and either we get nothing (1) or we get something useful of type B. In such a programming context "nothing" above means a predefined value NIL. This analogy supports our preference in the sequel for NIL as canonical inhabitant of datatype 1. In fact, we will refer to $1 + A$ (or $A + 1$) as the "pointer to A" datatype. This corresponds to the `Maybe` type constructor of the HASKELL *Standard Prelude*.

Datatype 2

Let us inspect the $1+1$ instance of the "pointer" construction just mentioned above. Any observation $B \xleftarrow{f} 1+1$ can be decomposed in two constant functions: $f = [\underline{b_1}, \underline{b_2}]$. Now suppose that $B = \{b_1, b_2\}$ (for $b_1 \neq b_2$). Then $1 + 1 \cong B$ will hold, for whatever choice of inhabitants b_1 and b_2. So we are in a situation similar to 1: we will use symbol 2 to represent the abstract class of all such Bs containing exactly two elements. Therefore, we can write:

$$1+1 \quad \cong \quad 2$$

Of course, $\mathbb{B} = \{\text{TRUE}, \text{FALSE}\}$ and initial segment $2 = \{1,2\}$ are in this abstract class. In the sequel we will show some preference for the particular choice of inhabitants $b_1 = \text{TRUE}$ and $b_2 = \text{FALSE}$, which enables us to use symbol 2 in places where Bool is expected. Clearly,

$$2 \times A \quad \cong \quad A + A \tag{2.59}$$

Exercise *2.11. Derive the isomorphism*

$$(B+C) \times A \xleftarrow{\mathsf{undistl}} (B \times A) + (C \times A) \tag{2.60}$$

from undistr *(2.50) and other isomorphisms studied thus far.*
□

Exercise *2.12.Use the exchange law to infer* undistl $= \langle \pi_1 + \pi_1, [\pi_2, \pi_2] \rangle$ *from the outcome of exercise 2.11.*
□

Exercise *2.13. Furthermore, show that (2.59) follows from (2.60) and, on the practical side, relate* HASKELL *expression*

```
either (split (const True) id) (split (const False) id)
```

to the same isomorphism (2.59).
□

2.12 NATURAL PROPERTIES

Let us resume discussion about $\mathsf{undistr}$ and the two other functions in diagram (2.54). What about using $\mathsf{undistr}$ itself to close this diagram, at the bottom? Note that definition (2.51) works for D, E and F in the same way it does for A, B and C. (Indeed, the particular choice of symbols A, B and C in (2.50) was rather arbitrary.) Therefore, we get:

$$\begin{tikzcd}
A \times (B + C) \arrow[d, "f \times (g+h)"'] & (A \times B) + (A \times C) \arrow[l, "\mathsf{undistr}"'] \arrow[d, "(f \times g) + (f \times h)"] \\
D \times (E + F) & (D \times E) + (D \times F) \arrow[l, "\mathsf{undistr}"]
\end{tikzcd}$$

which expresses a very important property of $\mathsf{undistr}$:

$$(f \times (g + h)) \cdot \mathsf{undistr} \quad = \quad \mathsf{undistr} \cdot ((f \times g) + (f \times h)) \qquad (2.61)$$

This is called the *natural* property of $\mathsf{undistr}$. This kind of property (often called *"free"* instead of *"natural"*) is not a privilege of $\mathsf{undistr}$. As a matter of fact, every function interfacing patterns such as F or G above will exhibit its own *natural* property. Furthermore, we have already quoted *natural* properties without mentioning it. Recall (2.10), for instance. This property (establishing *id* as the *unit* of composition) is, after all, the *natural* property of *id*. In this case we have $\mathsf{F}\,\alpha = \mathsf{G}\,\alpha = \alpha$, as can be easily observed in diagram (2.11).

In general, *natural* properties are described by diagrams in which two "copies" of the operator of interest are drawn as horizontal arrows:

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{F}\,A \arrow[d, "\mathsf{F}\,f"'] & \mathsf{G}\,A \arrow[l, "\phi"'] \arrow[d, "\mathsf{G}\,f"] \\
B & \mathsf{F}\,B & \mathsf{G}\,B \arrow[l, "\phi"]
\end{tikzcd} \qquad (\mathsf{F}\,f) \cdot \phi = \phi \cdot (\mathsf{G}\,f) \qquad (2.62)$$

Note that f is universally quantified, that is to say, the *natural* property holds for every $f : B \leftarrow A$.

Diagram (2.62) corresponds to unary patterns F and G. As we have seen with $\mathsf{undistr}$, other functions (g,h *etc.*) come into play for multiary patterns. A very important rôle will be assigned throughout this book to these F, G, *etc.* "shapes" or patterns which are shared by pointfree functional expressions and by their domain and codomain expressions. From chapter 3 onwards we will refer to them by their proper name — "functor" — which is standard in mathematics and computer science. Then we will also explain the names assigned to properties such as, for instance, (2.30) or (2.44).

***Exercise** 2.14. Show that (2.28) and (2.29) are* natural *properties. Dualize these properties. **Hint**: recall diagram (2.43).*

□

***Exercise** 2.15. Establish the* natural *properties of the* swap *(2.33) and* assocr *(2.35) isomorphisms.*

□

***Exercise** 2.16.Draw the natural property of function* $\phi = \mathsf{swap} \cdot (id \times \mathsf{swap})$ *as a diagram, that is, identify* F *and* G *in (2.62) for this case.*

Do the same for $\phi = \mathsf{coswap} \cdot (\mathsf{swap} + \mathsf{swap})$ *where* $\mathsf{coswap} = [i_2, i_1]$.

□

2.13 UNIVERSAL PROPERTIES

Functional constructs $\langle f, g \rangle$ and $[f, g]$ — and their derivatives $f \times g$ and $f + g$ — provide good illustration about what is meant by a *program combinator* in a compositional approach to programming: the combinator is put forward equipped with a concise *set of properties* which enable programmers to transform programs, reason about them and perform useful calculations. This leads to scientific *programming methodology*.

Such properties bear standard names such as *cancellation*, *reflection*, *fusion*, *absorption etc.*. Where do these names come from? As a rule, for each combinator to be defined one has to define suitable constructions at "interface"-level [10], *e.g.* $A \times B$ and $A + B$. These are not chosen or invented at random: each is defined in a way such that the associated combinator is uniquely defined. This is assured by a so-called *universal property* from which the others can be derived.

Take product $A \times B$, for instance. Its universal property states that, for each pair of arrows $A \xleftarrow{f} C$ and $B \xleftarrow{g} C$, there exists an arrow $A \times B \xleftarrow{\langle f,g \rangle} C$ such that

$$k = \langle f, g \rangle \quad \Leftrightarrow \quad \begin{cases} \pi_1 \cdot k = f \\ \pi_2 \cdot k = g \end{cases} \tag{2.63}$$

holds — recall diagram (2.23) — for all $A \times B \xleftarrow{k} C$.

Note that (2.63) is an *equivalence*, implicitly stating that $\langle f, g \rangle$ is the *unique* arrow satisfying the property on the right. In fact, read (2.63) in the $\Rightarrow$ direction and let k be $\langle f, g \rangle$. Then $\pi_1 \cdot \langle f, g \rangle = f$ and $\pi_2 \cdot \langle f, g \rangle = g$ will hold, meaning that $\langle f, g \rangle$ effectively obeys the property

[10] In the current context, *programs* "are" functions and program-*interfaces* "are" the datatypes involved in functional signatures.

on the right. In other words, we have derived ×-cancellation (2.22). Reading (2.63) in the $\Leftarrow$ direction we understand that, if some k satisfies such properties, then it "has to be" the same arrow as $\langle f,g\rangle$.

The relevance of universal property (2.63) is that it offers a way of *solving equations* of the form $k = \langle f,g\rangle$. Take for instance the following exercise: can the identity be expressed, or "reflected", using this combinator? We just solve the equation $id = \langle f,g\rangle$ for f and g:

$$\begin{array}{ll} & id = \langle f,g\rangle \\ \equiv & \quad \{ \text{ universal property (2.63) } \} \\ & \left\{ \begin{array}{l} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{array} \right. \\ \equiv & \quad \{ \text{ by (2.10) } \} \\ & \left\{ \begin{array}{l} \pi_1 = f \\ \pi_2 = g \end{array} \right. \end{array}$$

The equation has the unique solutions $f = \pi_1$ and $g = \pi_2$ which, once substituted in the equation itself, yield

$$id = \langle \pi_1, \pi_2\rangle$$

i.e., nothing but the ×-reflection law (2.32).

All other laws can be calculated from the universal property in a similar way. For instance, the ×-fusion (2.26) law is obtained by solving the equation $k = \langle i,j\rangle$ again for f and g, but this time fixing $k = \langle i,j\rangle \cdot h$, assuming i, j and h given:[11]

$$\begin{array}{ll} & \langle i,j\rangle \cdot h = \langle f,g\rangle \\ \equiv & \quad \{ \text{ universal property (2.63) } \} \\ & \left\{ \begin{array}{l} \pi_1 \cdot (\langle i,j\rangle \cdot h) = f \\ \pi_2 \cdot (\langle i,j\rangle \cdot h) = g \end{array} \right. \\ \equiv & \quad \{ \text{ composition is associative (2.8) } \} \\ & \left\{ \begin{array}{l} (\pi_1 \cdot \langle i,j\rangle) \cdot h = f \\ (\pi_2 \cdot \langle i,j\rangle) \cdot h = g \end{array} \right. \\ \equiv & \quad \{ \text{ by } \times\text{-cancellation (derived above) } \} \\ & \left\{ \begin{array}{l} i \cdot h = f \\ j \cdot h = g \end{array} \right. \end{array}$$

Substituting the solutions $f = i \cdot h$ and $g = j \cdot h$ in the equation, we get the ×-fusion law: $\langle i,j\rangle \cdot h = \langle i \cdot h, j \cdot h\rangle$.

[11] Solving equations of this kind is reminiscent of many similar calculations carried out in school maths and physics courses. The spirit is the same. The difference is that this time one is not calculating water pump debits, accelerations, velocities, or other physical entities: the solutions of our equations are (just) functional *programs*.

It will take about the same effort to derive *split* structural equality

$$\langle i,j\rangle = \langle f,g\rangle \quad\Leftrightarrow\quad \left\{\begin{array}{l} i = f \\ j = g \end{array}\right. \tag{2.64}$$

from universal property (2.63) — just let $k = \langle i,j\rangle$ and solve.

Similar arguments can be built around coproduct's universal property,

$$k = [f,g] \quad\Leftrightarrow\quad \left\{\begin{array}{l} k \cdot i_1 = f \\ k \cdot i_2 = g \end{array}\right. \tag{2.65}$$

from which structural equality of *either*s can be inferred,

$$[i,j] = [f,g] \quad\Leftrightarrow\quad \left\{\begin{array}{l} i = f \\ j = g \end{array}\right. \tag{2.66}$$

as well as the other properties we know about this combinator.

Exercise *2.17. Show that* assocr *(2.35) is iso by solving the equation* assocr $\cdot$ assocl $=$ *id for* assocl. ***Hint****: don't ignore the role of universal property (2.63) in the calculation.*

□

Exercise *2.18. Prove the equality:* $[\langle f,\underline{k}\rangle, \langle g,\underline{k}\rangle] = \langle [f,g],\underline{k}\rangle$

□

Exercise *2.19. Derive* $+$*-cancellation (2.40),* $+$*-reflection (2.41) and* $+$*-fusion (2.42) from universal property (2.65). Then derive the* exchange law *(2.49) from the universal property of product (2.63) or coproduct (2.65).*

□

Exercise *2.20. Function coassocr* $= [id + i_1, i_2 \cdot i_2]$ *is a witness of isomorphism* $(A+B)+C \cong A+(B+C)$*, from left to right. Calculate its converse coassocl by solving the equation*

$$\underbrace{[x,[y,z]]}_{coassocl} \cdot coassocr = id \tag{2.67}$$

for x, y and z.

□

Exercise *2.21. Let* δ *be a function of which you know that* $\pi_1 \cdot \delta = id$ *e* $\pi_2 \cdot \delta = id$ *hold. Show that necessarily* δ *satisfies the natural property* $(f \times f) \cdot \delta = \delta \cdot f$*.*

□

2.14 GUARDS AND MCCARTHY'S CONDITIONAL

Most functional programming languages and notations cater for pointwise conditional expressions of the form

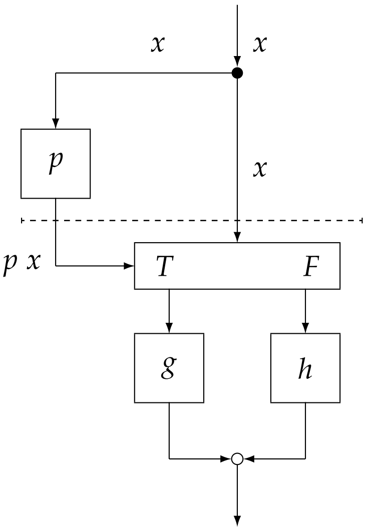
$$\textbf{if } p\ x \textbf{ then } g\ x \textbf{ else } h\ x \quad (2.68)$$

which evaluates to $g\ x$ in case $p\ x$ holds and to $h\ x$ otherwise, that is

$$\left\{ \begin{array}{lcl} p\,x & \Rightarrow & g\,x \\ \neg(p\,x) & \Rightarrow & h\,x \end{array} \right.$$

given some predicate $\mathbb{B} \xleftarrow{p} A$, some "then"-function $B \xleftarrow{g} A$ and some "else"-function $B \xleftarrow{h} A$.

Can (2.68) be written in the pointfree style?



The drawing above is an attempt to express such a conditional expression as a "block"-diagram. Firstly, the input x is copied, the left copy being passed to predicate p yielding the Boolean $p\ x$. One can easily define this part using $copy = \langle id, id \rangle$.

The informal part of the diagram is the T-F "switch": it should channel x to g in case $p\ x$ switches the T-output, or channel x to h otherwise.

At first sight, this T-F gate should be of type $\mathbb{B} \times A \to A \times A$. But the output cannot be $A \times A$, as f or g act in *alternation*, not in *parallel* — it should rather be $A + A$, in which case the last step is achieved just by running $[g, h]$. How does one switch from our starting product-based model of conditionals to a coproduct-based one?

The key observation is that the type $\mathbb{B} \times A$ marked by the dashed line in the block-diagram is isomorphic to $A + A$, recall (2.59). That is, the information captured by the pair $(p\ x, x) \in \mathbb{B} \times A$ can be converted into a unique $y \in A + A$ without loss of information. Let us define a new combinator for this, denoted $p?$:

$$(p?)a \quad = \quad \left\{ \begin{array}{lcl} p\,a & \Rightarrow & i_1\,a \\ \neg(p\,a) & \Rightarrow & i_2\,a \end{array} \right. \qquad (2.69)$$

We call $A + A \xleftarrow{p?} A$ a *guard*, or better, the guard associated to a given predicate $\mathbb{B} \xleftarrow{p} A$. In a sense, guard $p?$ is more "informative" than p alone: it provides information about the outcome of testing p on some input a, encoded in terms of the coproduct injections (i_1 for a *true* outcome and i_2 for a *false* outcome, respectively) without losing the input a itself.

Finally, the composition $[g,h] \cdot p?$, depicted in the following diagram

$$\begin{tikzcd}
& A \arrow[d, "p?"'] & \\
A \arrow[r, "i_1"] \arrow[dr, "g"'] & A+A \arrow[d, "{[g,h]}"'] & A \arrow[l, "i_2"'] \arrow[dl, "h"] \\
& B &
\end{tikzcd}$$

has (2.68) as pointwise meaning. It is a well-known functional combinator termed "McCarthy conditional"[12] and usually denoted by the expression $p \rightarrow g, h$. Altogether, we have the definition

$$p \rightarrow g, h \overset{\text{def}}{=} [g,h] \cdot p? \tag{2.70}$$

which suggests that, to reason about conditionals, one may seek help in the algebra of coproducts. Indeed, the following fact,

$$f \cdot (p \rightarrow g, h) = p \rightarrow f \cdot g, f \cdot h \tag{2.71}$$

which we shall refer to as the *first McCarthy's conditional fusion law*[13], is nothing but an immediate consequence of $+$-fusion (2.42).

We shall introduce and define instances of predicate p as long as they are needed. A particularly important assumption of our notation should, however, be mentioned at this point: we assume that, for every datatype A, the equality predicate $\mathbb{B} \xleftarrow{=_A} A \times A$ is defined in a way which guarantees three basic properties: reflexivity ($a =_A a$ for every a), transitivity ($a =_A b$ and $b =_A c$ implies $a =_A c$) and symmetry ($a =_A b$ iff $b =_A a$). Subscript A in $=_A$ will be dropped wherever implicit in the context.

In HASKELL programming, the equality predicate for a type becomes available by declaring the type as an instance of class `Eq`, which exports equality predicate `(==)`. This does not, however, guarantee the reflexive, transitive and symmetry properties, which need to be proved by dedicated mathematical arguments.

We close this section with an illustration of how *smart* pointfree algebra can be in reasoning about functions that *one does not actually define explicitly*. It also shows how relevant the *natural properties* studied in section 2.12 are. The issue is that our definition of a guard (2.69) is pointwise and most likely unsuitable to prove facts such as, for instance,

$$p? \cdot f = (f+f) \cdot (p \cdot f)? \tag{2.72}$$

[12] After John McCarthy, the computer scientist who first defined it.

[13] For the second one go to exercise 2.23.

Thinking better, instead of "inventing" (2.69), we might (and perhaps should!) have defined

$$\begin{tikzcd} A \arrow[r, "\langle p,id \rangle"] \arrow[rr, bend right, "p?"'] & 2 \times A \arrow[r, "\alpha"] & A + A \end{tikzcd} \tag{2.73}$$

which actually expresses rather closely our strategy of switching from products to coproducts in the definition of $(p?)$. Isomorphism α (2.59) is the subject of exercise 2.13. Do we need to define it explicitly? Perhaps not: from its type, $2 \times A \rightarrow A + A$, we immediately infer its natural (or "free") property:

$$\alpha \cdot (id \times f) = (f + f) \cdot \alpha \tag{2.74}$$

It turns out that this is the *knowledge* we need about α in order to prove (2.72), as the following calculation shows:

$$\begin{array}{cl}
& p\,?\cdot f \\
= & \quad \{\ (2.73)\,;\ \langle p, id \rangle \cdot f = \langle p \cdot f, f \rangle\ \} \\
& \alpha \cdot \langle p \cdot f, f \rangle \\
= & \quad \{\ \times\text{-absorption (2.27)}\ \} \\
& \alpha \cdot (id \times f) \cdot \langle p \cdot f, id \rangle \\
= & \quad \{\ \text{free theorem of } \alpha \text{ (2.74)}\ \} \\
& (f + f) \cdot \alpha \cdot \langle p \cdot f, id \rangle \\
= & \quad \{\ \text{(2.73) again}\ \} \\
& (f + f) \cdot (p \cdot f)?
\end{array}$$

□

Other examples of this kind of reasoning, based on natural (free) properties of isomorphisms — and often on "shunting" them around using laws (2.18,2.19) — will be given later in this book.

The less one has to write to solve a problem, the better. One saves time and one's brain, adding to productivity. This is often called *elegance* when applying a scientific method. (Unfortunately, be prepared for much lack of it in the software engineering field!)

***Exercise** 2.22. Prove that the following equality between two conditional expressions*

$$\begin{array}{cl}
& k\ (\mathbf{if}\ p\ x\ \mathbf{then}\ f\ x\ \mathbf{else}\ h\ x, \mathbf{if}\ p\ x\ \mathbf{then}\ g\ x\ \mathbf{else}\ i\ x) \\
= & \mathbf{if}\ p\ x\ \mathbf{then}\ k\ (\lambda ap\ f\ x, \lambda ap\ g\ x)\ \mathbf{else}\ k\ (h\ x, i\ x)
\end{array}$$

holds by rewriting it in the pointfree style (using the McCarthy's conditional combinator) and applying the exchange law *(2.49), among others.*

□

***Exercise** 2.23. Prove the* first McCarthy's conditional fusion law *(2.71). Then, from (2.70) and property (2.72), infer the second such law:*

$$(p \rightarrow f, g) \cdot h \quad = \quad (p \cdot h) \rightarrow (f \cdot h), (g \cdot h) \tag{2.75}$$

□

***Exercise** 2.24. Prove that property*

$$\langle f, (p \rightarrow q\ ,\ h) \rangle \quad = \quad p \rightarrow \langle f, q \rangle\ ,\ \langle f, h \rangle \tag{2.76}$$

and its corollary

$$(p \rightarrow g\ ,\ h) \times f \quad = \quad p \cdot \pi_1 \rightarrow g \times f\ ,\ h \times f \tag{2.77}$$

hold, assuming the basic fact:

$$p \rightarrow f\ ,\ f \quad = \quad f \tag{2.78}$$

□

***Exercise** 2.25. Define $p\ (x, y) = x > y$ and the maximum of two integers, $m\ (x, y)$, by:*

$$m \quad = \quad p \rightarrow \pi_1\ ,\ \pi_2$$

Then show that

$$\mathsf{succ} \cdot m = m \cdot (\mathsf{succ} \times \mathsf{succ})$$

holds, by using the McCarthy conditional fusion-laws and basic arithmetics.
□

2.15 GLUING FUNCTIONS WHICH DO NOT COMPOSE — EXPONENTIALS

Now that we have made the distinction between the pointfree and pointwise functional notations reasonably clear, it is instructive to revisit section 2.2 and identify *functional application* as the "bridge" between the pointfree and pointwise worlds. However, we should say "a bridge" rather than "the bridge", for in this section we enrich such an interface with another "bridge" which is very relevant to programming.

Suppose we are given the task to combine two functions, one binary $B \xleftarrow{f} C \times A$ and the other unary: $D \xleftarrow{g} A$. It is clear that none of the combinations $f \cdot g$, $\langle f, g \rangle$ or $[f, g]$ is well-typed. So, f and g cannot be put together directly — they require some extra interfacing.

Note that $\langle f, g \rangle$ would be well-defined in case the C component of f's domain could be somehow "ignored". Suppose, in fact, that in some particular context the first argument of f happens to be "irrelevant", or to be frozen to some $c \in C$. It is easy to derive a new function

$$\begin{array}{lcl} f_c & : & A \rightarrow B \\ f_c\, a & \stackrel{\mathrm{def}}{=} & f(c, a) \end{array}$$

from f which combines nicely with g via the *split* combinator: $\langle f_c, g \rangle$ is well-defined and bears type $B \times D \leftarrow A$. For instance, suppose that $C = A$ and f is the equality predicate $=$ on A. Then $\mathsf{Bool} \xleftarrow{=_c} A$ is the "equal to c" predicate on A values:

$$=_c\ a \ \stackrel{\mathrm{def}}{=}\ a = c \tag{2.79}$$

As another example, recall function *twice* (2.3) which could be defined as $\times_2$ using the new notation.

However, we need to be more careful about what is meant by f_c. Such as functional application, expression f_c interfaces the pointfree and the pointwise levels — it involves a function (f) and a value (c). But, for $B \xleftarrow{f} C \times A$, there is a major distinction between $f\, c$ and f_c — while the former denotes a value of type B, *i.e.* $f\, c \in B$, f_c denotes a function of type $B \leftarrow A$. We will say that $f_c \in B^A$ by introducing a new datatype construct which we will refer to as the *exponential*:

$$B^A \ \stackrel{\mathrm{def}}{=}\ \{g \mid g : B \leftarrow A\} \tag{2.80}$$

There are strong reasons to adopt the B^A notation to the detriment of the more obvious $B \leftarrow A$ or $A \rightarrow B$ alternatives, as we shall see shortly.

The B^A exponential datatype is therefore inhabited by functions from A to B, that is to say, functional declaration $g : B \leftarrow A$ means the same as $g \in B^A$. And what do we want functions for? We want to apply them. So it is natural to introduce the *apply* operator

$$\begin{array}{l} ap : \ B \xleftarrow{ap} B^A \times A \\ ap(f, a) \stackrel{\mathrm{def}}{=} f\, a \end{array} \tag{2.81}$$

which applies a function f to an argument a.

Back to generic binary function $B \xleftarrow{f} C \times A$, let us now think of the operation which, for every $c \in C$, produces $f_c \in B^A$. This can be regarded as a function of signature $B^A \leftarrow C$ which expresses f as a kind of C-indexed family of functions of signature $B \leftarrow A$. We will denote such a function by $\overline{f}$ (read $\overline{f}$ as "f transposed"). Intuitively, we want f and $\overline{f}$ to be related to each other by the following property:

$$f(c,a) \;=\; (\overline{f}\,c)a \tag{2.82}$$

Given c and a, both expressions denote the same value. But, in a sense, $\overline{f}$ is more tolerant than f: while the latter is binary and requires *both* arguments (c,a) to become available before application, the former is happy to be provided with c first and with a later on, if actually required by the evaluation process.

Similarly to $A \times B$ and $A + B$, exponential B^A involves a universal property,

$$k = \overline{f} \;\Leftrightarrow\; f = ap \cdot (k \times id) \tag{2.83}$$

from which laws for cancellation, reflection and fusion can be derived:

Exponentials cancellation :

$$\begin{tikzcd}
B^A & B^A \times A \arrow[r, "ap"] & B \\
C \arrow[u, "\overline{f}"] & C \times A \arrow[u, "\overline{f}\times id"] \arrow[ur, "f"'] &
\end{tikzcd}$$

$$f = ap \cdot (\overline{f} \times id) \tag{2.84}$$

Exponentials reflection :

$$\begin{tikzcd}
B^A & B^A \times A \arrow[r, "ap"] & B \\
B^A \arrow[u, "id_{B^A}"] & B^A \times A \arrow[u, "id_{B^A}\times id_A"] \arrow[ur, "ap"'] &
\end{tikzcd}$$

$$\overline{ap} = id_{B^A} \tag{2.85}$$

Exponentials fusion :

$$\begin{tikzcd}
B^A & B^A \times A \arrow[r, "ap"] & B \\
C \arrow[u, "\overline{g}"] & C \times A \arrow[u, "\overline{g}\times id"] \arrow[ur, "g"] & \\
D \arrow[u, "f"] & D \times A \arrow[u, "f\times id"] \arrow[uur, "g\cdot(f\times id)"'] &
\end{tikzcd}$$

$$\overline{g \cdot (f \times id)} = \overline{g} \cdot f \tag{2.86}$$

Note that the cancellation law is nothing but fact (2.82) written in the pointfree style.

Is there an absorption law for exponentials? The answer is affirmative but first we need to introduce a new functional combinator which arises as the transpose of $f \cdot ap$ in the following diagram:

$$\begin{tikzcd}
D^A \times A \arrow[r, "ap"] & D \\
B^A \times A \arrow[u, "\overline{f\cdot ap}\times id"] \arrow[r, "ap"] & B \arrow[u, "f"']
\end{tikzcd}$$

We shall denote this by f^A and its type-rule is as follows:

$$\frac{C \xleftarrow{f} B}{C^A \xleftarrow{f^A} B^A}$$

It can be shown that, once A and $C \xleftarrow{f} B$ are fixed, f^A is the function which accepts some input function $B \xleftarrow{g} A$ as argument and produces function $f \cdot g$ as result (see exercise 2.41). So f^A is the "compose with f" functional combinator:

$$(f^A)g \stackrel{\text{def}}{=} f \cdot g \tag{2.87}$$

Now we are ready to understand the laws which follow:

Exponentials absorption :

$$\begin{tikzcd}
D^A & D^A \times A \arrow[r, "ap"] & D \\
B^A \arrow[u, "f^A"] & B^A \times A \arrow[u, "f^A \times id"] \arrow[r, "ap"] & B \arrow[u, "f"] \\
C \arrow[u, "\overline{g}"] & C \times A \arrow[u, "\overline{g} \times id"] \arrow[ur, "g"] &
\end{tikzcd}$$

$$\overline{f \cdot g} = f^A \cdot \overline{g} \tag{2.88}$$

Note how, from this, we also get

$$f^A = \overline{f \cdot ap} \tag{2.89}$$

Thus (2.88) can also be written

$$\overline{f \cdot g} = \overline{f \cdot ap} \cdot \overline{g} \tag{2.90}$$

Exponentials-functor :

$$(g \cdot h)^A = g^A \cdot h^A \tag{2.91}$$

Exponentials-functor-id :

$$id^A = id \tag{2.92}$$

WHY THE EXPONENTIAL NOTATION. To conclude this section we need to explain why we have adopted the apparently esoteric B^A notation for the "function from A to B" data type. This is the opportunity to relate what we have seen above with two (higher order) functions which are very familiar to functional programmers. In the HASKELL Prelude they are defined thus:

```
curry :: ((a,b) → c) → (a → b → c)
curry f a b = f (a,b)
```

$$
\begin{array}{l}
\mathsf{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow (a,b) \rightarrow c \\
\mathsf{uncurry}\, f\ (a,b) = f\ a\ b
\end{array}
$$

In our notation for types, curry maps functions in function space $C^{A \times B}$ to functions in $(C^B)^A$; and uncurry maps functions from the latter function space to the former.

Let us calculate the meaning of curry by removing variables from its definition:

$$\overbrace{(\underbrace{\mathsf{curry}\, f}_{\overline{f}}\ a)}^{g}\, b = f\ (a,b)$$

$$\equiv \qquad \{\ \text{introduce } g\ \}$$

$$g\, b \ = \ f(a,b)$$

$$\equiv \qquad \{\ \text{since } g\, b = ap(g,b) \text{ (2.81) }\ \}$$

$$ap(g,b) \ = \ f(a,b)$$

$$\equiv \qquad \{\ g = \overline{f}\ a\ ; \text{natural-}id\ \}$$

$$ap(\overline{f}\ a, id\ b) \ = \ f(a,b)$$

$$\equiv \qquad \{\ \text{product of functions: } (f \times g)(x,y) = (f\ x, g\ y)\ \}$$

$$ap((\overline{f} \times id)(a,b)) \ = \ f(a,b)$$

$$\equiv \qquad \{\ \text{composition}\ \}$$

$$(ap \cdot (\overline{f} \times id))(a,b) \ = \ f(a,b)$$

$$\equiv \qquad \{\ \text{extensionality (2.5), i.e. removing points } a \text{ and } b\ \}$$

$$ap \cdot (\overline{f} \times id) \ = \ f$$

From the above we infer that the definition of curry is a re-statement of the cancellation law (2.84). That is,

$$\mathsf{curry}\, f \ \stackrel{\mathrm{def}}{=}\ \overline{f} \tag{2.93}$$

and curry is transposition in HASKELL-speak.[14]

Next we do the same for the definition of uncurry :

$$\underbrace{\mathsf{uncurry}\, f}_{k}\ (a,b) = f\ a\ b$$

$$\equiv \qquad \{\ \text{introduce } k\ ; \text{lefthand side as calculated above}\ \}$$

$$k\ (a,b) = (ap \cdot (f \times id))\ (a,b)$$

$$\equiv \qquad \{\ \text{extensionality (2.5)}\ \}$$

$$k = ap \cdot (f \times id)$$

$$\equiv \qquad \{\ \text{universal property (2.83)}\ \}$$

[14] This terminology widely adopted in other functional languages.

$$
\begin{array}{ll}
 & f = \overline{k} \\
\equiv & \quad \{ \text{ expand } k \ \} \\
 & f = \overline{\mathsf{uncurry}\, f}
\end{array}
$$

We conclude that uncurry is the inverse of transposition, that is, of curry . We shall use the abbreviation $\widehat{f}$ for uncurry f, whereby the above equality is written $f = \overline{\widehat{f}}$. It can also be checked that $f = \widehat{\overline{f}}$ also holds, instantiating k above by $\widehat{\overline{f}}$:

$$
\begin{array}{ll}
 & \widehat{\overline{f}} = ap \cdot (\overline{f} \times id) \\
\equiv & \quad \{ \text{ cancellation (2.84) } \} \\
 & \widehat{\overline{f}} = f
\end{array}
$$

□

So uncurry — i.e. $\widehat{(\_)}$ — and curry — i.e. $\overline{(\_)}$ — are inverses of each other,

$$
g = \overline{f} \quad \Leftrightarrow \quad \hat{g} = f \tag{2.94}
$$

leading to isomorphism

$$
A \to C^B \quad \cong \quad A \times B \to C
$$

which can also be written as

$$
\begin{tikzcd}
(C^B)^A \arrow[rr, bend left, "\mathsf{uncurry}"] \arrow[rr, phantom, "\cong"] & & C^{A \times B} \arrow[ll, bend left, "\mathsf{curry}"]
\end{tikzcd} \tag{2.95}
$$

decorated with the corresponding witnesses.[15]

Isomorphism (2.95) is at the core of the theory and practice of functional programming. It clearly resembles a well known equality concerning numeric exponentials, $b^{c \times a} = (b^a)^c$. Moreover, other known facts about numeric exponentials, *e.g.* $a^{b+c} = a^b \times a^c$ or $(b \times c)^a = b^a \times c^a$ also find their counterpart in functional exponentials. The counterpart of the former,

$$
A^{B+C} \quad \cong \quad A^B \times A^C \tag{2.96}
$$

arises from the uniqueness of the *either* combination: every pair of functions $(f, g) \in A^B \times A^C$ leads to a unique function $[f, g] \in A^{B+C}$ and vice-versa, every function in A^{B+C} is the *either* of some function in A^B and of another in A^C.

The function exponentials counterpart of the second fact about numeric exponentials above is

$$
(B \times C)^A \cong B^A \times C^A \tag{2.97}
$$

[15] Writing $\overline{f}$ (resp. $\widehat{f}$) or curry f (resp. uncurry f) is a matter of taste: the latter are more in the tradition of functional programming and help when the functions have to be named; the former save ink in algebraic expressions and calculations.

This can be justified by a similar argument concerning the uniqueness of the *split* combinator $\langle f, g \rangle$.

What about other facts valid for numeric exponentials such as $a^0 = 1$ and $1^a = 1$? The reader is invited to go back to section 2.11 and recall what 0 and 1 mean as datatypes: the empty (void) and singleton datatypes, respectively. Our counterpart to $a^0 = 1$ then is

$$A^0 \cong 1 \tag{2.98}$$

where A^0 denotes the set of all functions from the empty set to some A. What does (2.98) mean? It simply tells us that there is only one function in such a set — the empty function mapping "no" value at all. This fact confirms our choice of notation once again (compare with $a^0 = 1$ in a numeric context).

Next, we may wonder about facts

$$1^A \cong 1 \tag{2.99}$$

$$A^1 \cong A \tag{2.100}$$

which are the functional exponentiation counterparts of $1^a = 1$ and $a^1 = a$. Fact (2.99) is valid: it means that there is only one function mapping A to some singleton set $\{c\}$ — the constant function $\underline{c}$. There is no room for another function in 1^A because only c is available as output value. Our standard denotation for such a unique function is given by (2.58).

Fact (2.100) is also valid: all functions in A^1 are (single valued) constant functions and there are as many constant functions in such a set as there are elements in A. These functions are often called (abstract) "points" because of the 1-to-1 mapping between A^1 and the elements (points) in A.

***Exercise** 2.26. Relate the isomorphism involving generic elementary type* 2

$$A \times A \cong A^2 \tag{2.101}$$

to the expression $\lambda f \rightarrow (f\ \mathsf{True}, f\ \mathsf{False})$ *written in* HASKELL *syntax.*
□

***Exercise** 2.27. Consider the witnesses of isomorphism (2.97)*

$$\begin{tikzcd}(B \times C)^A \arrow[r, bend left, "\mathsf{unpair}"] \arrow[r, phantom, "\cong"] & B^A \times C^A \arrow[l, bend left, "\mathsf{pair}"]\end{tikzcd}$$

defined by:

$\mathsf{pair}\ (f, g) = \langle f, g \rangle$
$\mathsf{unpair}\ k = (\pi_1 \cdot k, \pi_2 \cdot k)$

Show that $\mathsf{pair} \cdot \mathsf{unpair} = id$ *and* $\mathsf{unpair} \cdot \mathsf{pair} = id$ *hold.*
□

***Exercise** 2.28. Show that the following equality*

$$\overline{f}\ a = f \cdot \langle \underline{a}, id \rangle \tag{2.102}$$

holds.
□

***Exercise** 2.29. Prove the equality* $\underline{g} = \overline{g \cdot \pi_2}$ *knowing that*

$$\overline{\pi_2} = \underline{id} \tag{2.103}$$

holds.
□

2.16 FINITARY PRODUCTS AND COPRODUCTS

In section 2.8 it was suggested that product could be regarded as the abstraction behind data-structuring primitives such as `struct` in C or `record` in Pascal. Similarly, coproducts were suggested in section 2.9 as abstract counterparts of C unions or Pascal variant records. For a finite A, exponential B^A could be realized as an *array* in any of these languages. These analogies are captured in table 1.

In the same way C `struct`s and `union`s may contain finitely many entries, as may Pascal (variant) records, product $A \times B$ extends to finitary product $A_1 \times \ldots \times A_n$, for $n \in \mathbb{N}$, also denoted by $\Pi_{i=1}^{n} A_i$, to which as many projections π_i are associated as the number n of factors involved. Of course, *splits* become n-ary as well

$$\langle f_1, \ldots, f_n \rangle : A_1 \times \ldots \times A_n \leftarrow B$$

for $f_i : A_i \leftarrow B$, $i = 1, n$.

Dually, coproduct $A + B$ is extensible to the finitary sum $A_1 + \cdots + A_n$, for $n \in \mathbb{N}$, also denoted by $\sum_{j=1}^{n} A_j$, to which as many injections i_j are assigned as the number n of terms involved. Similarly, *eithers* become n-ary

$$[\, f_1, \ldots, f_n \,] : A_1 + \ldots + A_n \rightarrow B$$

for $f_i : B \leftarrow A_i$, $i = 1, n$.

Abstract notation	PASCAL	C/C++	Description

$A \times B$	`record`
	`P: A;`
	`S: B`
	`end;`
		`struct {`
		`A first;`
		`B second;`
		`};`
			Records

$A + B$	`record`
	`case`
	`tag:  integer`
	`of x =`
	`1:  (P:A);`
	`2:  (S:B)`
	`end;`
		`struct {`
		`int tag; /* 1,2 */`
		`union {`
		`A ifA;`
		`B ifB;`
		`} data;`
		`};`
			Variant records

B^A	`array[A] of B`	`B ...[A]`	Arrays
$1 + A$	`^A`	`A *...`	Pointers

Table 1.: Abstract notation versus programming language datastructures.

Datatype n

Next after 2, we may think of 3 as representing the abstract class of all datatypes containing exactly three elements. Generalizing, we may think of n as representing the abstract class of all datatypes containing exactly n elements. Of course, initial segment n will be in this abstract class. (Recall (2.17), for instance: both Weekday and 7 are abstractly represented by 7.) Therefore,

$$n \cong \underbrace{1 + \cdots + 1}_{n}$$

and

$$\underbrace{A \times \ldots \times A}_{n} \cong A^n \tag{2.104}$$

$$\underbrace{A + \ldots + A}_{n} \cong n \times A \tag{2.105}$$

hold.

***Exercise** 2.30. On the basis of table 1, encode* undistr *(2.51) in C or* PASCAL. *Compare your code with the* HASKELL *pointfree and pointwise equivalents.*
□

2.17 INITIAL AND TERMINAL DATATYPES

All properties studied for binary *splits* and binary *eithers* extend to the finitary case. For the particular situation $n = 1$, we will have $\langle f \rangle = [\, f\,] = f$ and $\pi_1 = i_1 = id$, of course. For the particular situation

$n = 0$, finitary products "degenerate" to 1 and finitary coproducts "degenerate" to 0. So diagrams (2.23) and (2.38) are reduced to

$$\begin{tikzcd}
1 & & 0 \arrow[d, "{[\ ]}"] \\
C \arrow[u, "\langle\rangle"] & & C
\end{tikzcd}$$

The standard notation for the empty *split* $\langle\rangle$ is $!_C$, where subscript C can be omitted if implicit in the context. By the way, this is precisely the only function in 1^C, recall (2.99) and (2.58). Dually, the standard notation for the empty *either* $[\ ]$ is $?_C$, where subscript C can also be omitted. By the way, this is precisely the only function in C^0, recall (2.98).

In summary, we may think of 0 and 1 as, in a sense, the "extremes" of the whole datatype spectrum. For this reason they are called *initial* and *terminal*, respectively. We conclude this subject with the presentation of their main properties which, as we have said, are instances of properties we have stated for products and coproducts.

Initial datatype reflection :

$$\begin{tikzcd}
0 \arrow[loop above, "?_0=id_0"]
\end{tikzcd}$$

$$?_0 = id_0 \tag{2.106}$$

Initial datatype fusion :

$$\begin{tikzcd}
0 \arrow[d, "?_A"'] \arrow[dr, "?_B"] & \\
A \arrow[r, "f"'] & B
\end{tikzcd}$$

$$f \cdot ?_A = ?_B \tag{2.107}$$

Terminal datatype reflection :

$$\begin{tikzcd}
1 \arrow[loop above, "!_1=id_1"]
\end{tikzcd}$$

$$!_1 = id_1 \tag{2.108}$$

Terminal datatype fusion :

$$\begin{tikzcd}
1 & \\
A \arrow[u, "!_A"] & B \arrow[l, "f"] \arrow[ul, "!_B"']
\end{tikzcd}$$

$$!_A \cdot f = !_B \tag{2.109}$$

***Exercise** 2.31. Particularize the* exchange law *(2.49) to empty products and empty coproducts,* i.e. 1 *and* 0.

□

2.18 SUMS AND PRODUCTS IN HASKELL

We conclude this chapter with an analysis of the main primitive available in HASKELL for creating datatypes: the `data` declaration. Suppose we declare

$$\textbf{data}\ \mathit{CostumerId} = P\ \mathbb{Z} \mid C\ \mathbb{Z}$$

meaning to say that, for some company, a client is identified either by its passport number or by its credit card number, if any. What does this piece of syntax precisely mean?

If we enquire the HASKELL *interpreter* about what it knows about `CostumerId`, the reply will contain the following information:

```
Main> :i CostumerId
-- type constructor
data CostumerId

-- constructors:
P :: Int -> CostumerId
C :: Int -> CostumerId
```

In general, let A and B be two known datatypes. Via declaration

```
data C = C1 A | C2 B
```
(2.110)

one obtains from HASKELL a new datatype C equipped with constructors $C \xleftarrow{C1} A$ and $C \xleftarrow{C2} B$, in fact the only ones available for constructing values of C:

$$\begin{tikzcd}
A \arrow[dr, "C1"'] & & B \arrow[dl, "C2"] \\
& C &
\end{tikzcd}$$

This diagram leads to an obvious instance of coproduct diagram (2.38),

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "C1"'] & A + B \arrow[d, "{[C1,C2]}"] & B \arrow[l, "i_2"'] \arrow[dl, "C2"] \\
& C &
\end{tikzcd}$$

describing that a `data` declaration in HASKELL means the *either* of its constructors.

Because there are no other means to build C data, it follows that C is isomorphic to $A + B$. So $[C1, C2]$ has an inverse, say inv, which is such that $inv \cdot [C1, C2] = id$. How do we calculate inv? Let us first think of the generic situation of a function $D \xleftarrow{f} C$ which observes datatype C:

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "C1"'] & A+B \arrow[d, "{[C1,C2]}"] & B \arrow[l, "i_2"'] \arrow[dl, "C2"] \\
& C \arrow[d, "f"] & \\
& D &
\end{tikzcd}$$

This is an opportunity for *+-fusion* (2.42), whereby we obtain

$$f \cdot [C1, C2] \quad = \quad [f \cdot C1, f \cdot C2]$$

Therefore, the observation will be fully described provided we explain how f behaves with respect to $C1$ — *cf.* $f \cdot C1$ — and with respect to $C2$ — *cf.* $f \cdot C2$. This is what is behind the typical *inductive* structure of pointwise f, which will be made of two and only two clauses:

$$\begin{array}{l}
f : C \rightarrow D \\
f(C1\,a) = \ldots \\
f(C2\,b) = \ldots
\end{array}$$

Let us use this in calculating the inverse *inv* of $[C1, C2]$:

$$\begin{array}{cl}
 & inv \cdot [C1, C2] = id \\
\equiv & \qquad \{ \text{ by } +\text{-}\mathit{fusion} \text{ (2.42) } \} \\
 & [inv \cdot C1, inv \cdot C2] = id \\
\equiv & \qquad \{ \text{ by } +\text{-}\mathit{reflection} \text{ (2.41) } \} \\
 & [inv \cdot C1, inv \cdot C2] = [i_1, i_2] \\
\equiv & \qquad \{ \text{ } \mathit{either} \text{ structural equality (2.66) } \} \\
 & inv \cdot C1 = i_1 \wedge inv \cdot C2 = i_2
\end{array}$$

Therefore:

$$\begin{array}{l}
inv : C \rightarrow A + B \\
inv(C1\,a) = i_1\,a \\
inv(C2\,b) = i_2\,b
\end{array}$$

In summary, $C1$ is a “renaming” of injection i_1, $C2$ is a “renaming” of injection i_2 and C is a “renamed” replica of $A + B$:

$$C \xleftarrow{[C1,C2]} A + B \tag{2.111}$$

$[C1, C2]$ is called the *algebra* of datatype C and its inverse *inv* is called the *coalgebra* of C. The algebra contains the constructors $C1$ and $C2$ of type C, that is, it is used to "build" C-values. In the opposite direction, co-algebra *inv* enables us to "destroy" or observe values of C:

$$\begin{tikzcd}
C \arrow[rr, bend left, "inv"] \arrow[rr, phantom, "\cong"] & & A+B \arrow[ll, bend left, "{[C1,C2]}"]
\end{tikzcd}$$

Algebra/coalgebras also arise about product datatypes. For instance, suppose that one wishes to describe datatype *Point* inhabited by pairs (x_0, y_0), (x_1, y_1) (*etc.*) of Cartesian coordinates of a given type, say A. Although $A \times A$ equipped with projections π_1, π_2 "is" such a datatype, one may be interested in a suitably named replica of $A \times A$ in which points are built explicitly by some constructor (say *Point*) and observed by dedicated selectors (say x and y):

$$\begin{tikzcd}
A & A \times A \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "Point"] & A \\
& Point \arrow[ul, "x"] \arrow[ur, "y"'] &
\end{tikzcd} \tag{2.112}$$

This gives birth to the algebra *Point* and the coalgebra $\langle x, y \rangle$ of datatype *Point*:

$$\begin{tikzcd}
Point \arrow[rr, bend left, "\langle x{,}y\rangle"] \arrow[rr, phantom, "\cong"] & & A \times A \arrow[ll, bend left, "Point"]
\end{tikzcd}$$

In HASKELL one writes

$$\mathbf{data}\ Point\ a = Point\ \{x :: a, y :: a\}$$

but be warned that HASKELL delivers *Point* in curried form:

$$Point :: a \to a \to Point\ a$$

Finally, what is the "HASKELL-equivalent" to handling a *pointer* in (say) C? This corresponds to $A = 1$ in (2.111),

$$C \xleftarrow{[C1,C2]} 1 + B$$

and to the following HASKELL declaration:

$$\mathbf{data}\ C = C1\ ()\ |\ C2\ B$$

Note that HASKELL allows for a more programming-oriented alternative in this case, in which the unit type $()$ is eliminated:

$$\mathbf{data}\ C = C1\ |\ C2\ B$$

The difference is that here *C1* denotes an inhabitant of C (and so a clause $f(C1\,a) = \ldots$ is rewritten to $f\,C1 = \ldots$) while above *C1* denotes a (constant) function $C \xleftarrow{C1} 1$. Isomorphism (2.100) helps in comparing these two alternative situations.

2.19 EXERCISES

Exercise 2.32. *Let A and B be two disjoint datatypes, that is, $A \cap B = \emptyset$ holds. Show that isomorphism*

$$A \cup B \quad \cong \quad A + B \tag{2.113}$$

holds. ***Hint***: *define* $A \cup B \xleftarrow{i} A + B$ *as* $i = [emb_A\,, emb_B]$ *for* $emb_A\; a = a$ *and* $emb_B\; b = b$*, and find its inverse. By the way, why didn't we define i as simply as* $i \stackrel{\text{def}}{=} [id_A\,, id_B]$*?*

□

Exercise 2.33. *Knowing that a given function* xr *satisfies property*

$$\mathsf{xr} \cdot \langle\langle f, g\rangle, h\rangle = \langle\langle f, h\rangle, g\rangle \tag{2.114}$$

for all f, g and h, derive from (2.114) the definition of xr*:*

$$\mathsf{xr} = \langle \pi_1 \times id, \pi_2 \cdot \pi_1 \rangle \tag{2.115}$$

□

Exercise 2.34. *Let* distr *(read:* ‘distribute right’*) be the bijection which witnesses isomorphism* $A \times (B + C) \cong A \times B + A \times C$*. Fill in the “. . . ”in the diagram which follows so that it describes bijection* distl *(red:* ‘distribute left’*) which witnesses isomorphism* $(B + C) \times A \cong B \times A + C \times A$*:*

$$\begin{tikzcd}
(B+C)\times A \arrow[r, "\mathsf{swap}"] \arrow[rrr, bend right, "\mathsf{distl}"'] & \cdots \arrow[r, "\mathsf{distr}"] & \cdots \arrow[r, "\cdots"] & B\times A + C\times A
\end{tikzcd}$$

□

Exercise 2.35. *In the context of exercise 2.34, prove*

$$[g, h] \times f \quad = \quad [g \times f\,, h \times f] \cdot \mathsf{distl} \tag{2.116}$$

knowing that

$$f \times [g, h] \quad = \quad [f \times g, f \times h] \cdot \mathsf{distr} \tag{2.117}$$

holds.

□

Exercise 2.36. *Noting the following consequence of (2.116),*

$$ap \cdot ([g, h] \times id) = ap \cdot [g \times id\,, h \times id] \cdot \mathsf{distl}$$

find g and h such that $ap \cdot [g \times id, h \times id] = id$. Conclude that

$$\overline{\mathsf{distl}} = [\overline{i_1}, \overline{i_2}] \tag{2.118}$$

Draw the type diagram of $\overline{\mathsf{distl}}$.

□

***Exercise** 2.37. The arithmetic law $(a+b)(c+d) = (ac+ad)+(bc+bd)$ corresponds to the isomorphism*

$$\begin{tikzcd}
(A+B)\times(C+D) \arrow[r, phantom, "\cong"] & (A\times C + A\times D)+(B\times C+B\times D) \arrow[l, bend left, "h=[[i_1\times i_1, i_1\times i_2], [i_2\times i_1, i_2\times i_2]]"]
\end{tikzcd}$$

From universal property (2.65) infer the following definition of function h, written in Haskell syntax:

```
h(Left(Left(a,c))) = (Left a,Left c)
h(Left(Right(a,d))) = (Left a,Right d)
h(Right(Left(b,c))) = (Right b,Left c)
h(Right(Right(b,d))) = (Right b,Right d)
```

□

***Exercise** 2.38. Every C programmer knows that a struct of pointers*

$$(A+1)\times(B+1)$$

offers a data type which represents both product $A \times B$ (struct) and coproduct $A + B$ (union), alternatively. Express in pointfree notation the isomorphisms i_1 to i_5 of

$$\begin{tikzcd}
(A+1)\times(B+1) & ((A+1)\times B)+((A+1)\times 1) \arrow[l, "i_1"'] \\
 & (A\times B+1\times B)+(A\times 1+1\times 1) \arrow[u, "i_2"'] \\
 & (A\times B+B)+(A+1) \arrow[u, "i_3"'] \\
(A\times B+(B+A))+1 \arrow[r, "i_5"'] & A\times B+(B+(A+1)) \arrow[u, "i_4"']
\end{tikzcd}$$

which witness the observation above.

□

***Exercise** 2.39. Prove the following property of McCarthy conditionals:*

$$p \rightarrow f \cdot g \;,\; h \cdot k \quad = \quad [f, h] \cdot (p \rightarrow i_1 \cdot g \;,\; i_2 \cdot k) \tag{2.119}$$

□

***Exercise** 2.40. Assuming the fact*

$$(p? + p?) \cdot p? \quad = \quad (i_1 + i_2) \cdot p? \tag{2.120}$$

show that nested conditionals can be simplified:

$$p \rightarrow (p \rightarrow f \ , \ g) \ , \ (p \rightarrow h \ , \ k) \quad = \quad p \rightarrow f \ , \ k \tag{2.121}$$

□

***Exercise** 2.41. Show that $(\overline{f \cdot ap})\, g = f \cdot g$ holds,* cf. *(2.87).*
□

***Exercise** 2.42. Consider the higher-order isomorphism $flip$ defined as follows:*

$$\begin{array}{ccccccc} (C^B)^A & \cong & C^{A \times B} & \cong & C^{B \times A} & \cong & (C^A)^B \\ f & \mapsto & \widehat{f} & \mapsto & \widehat{f}.\mathsf{swap} & \mapsto & \overline{\widehat{f} \cdot \mathsf{swap}} = flip\ f \end{array}$$

Show that $flip\ f\ x\ y = f\ y\ x$.
□

***Exercise** 2.43. Let $C \xrightarrow{\mathsf{const}} C^A$ be the function of exercise 2.2, that is, $\mathsf{const}\, c = \underline{c}_A$. Which fact is expressed by the following diagram featuring* const*?*

$$\begin{tikzcd} C \arrow[r, "\mathsf{const}"] \arrow[d, "f"'] & C^A \arrow[d, "f^A"] \\ B \arrow[r, "\mathsf{const}"'] & B^A \end{tikzcd} \tag{2.122}$$

Write it at point-level and describe it by your own words.
□

***Exercise** 2.44. Show that $\overline{\pi_2} \cdot f = \overline{\pi_2}$ holds for every f. Thus $\overline{\pi_2}$ is a constant function — which one?*
□

***Exercise** 2.45. Establish the difference between the following two declarations in* HASKELL*,*

$$\begin{array}{l}\textbf{data}\ D = D1\ A \mid D2\ B\ C\\ \textbf{data}\ E = E1\ A \mid E2\ (B,C)\end{array}$$

for A, B and C any three predefined types. Are D and E isomorphic? If so, can you specify and encode the corresponding isomorphism?
□

2.20 BIBLIOGRAPHY NOTES

A few decades ago John Backus read, in his Turing Award Lecture, a revolutionary paper [8]. This paper proclaimed conventional command-oriented programming languages obsolete because of their inefficiency arising from retaining, at a high-level, the so-called "memory access bottleneck" of the underlying computation model — the well-known *von Neumann* architecture. Alternatively, the (at the time already mature) *functional programming* style was put forward for two main reasons. Firstly, because of its potential for concurrent and parallel computation. Secondly — and Backus emphasis was really put on this —, because of its strong algebraic basis.

Backus *algebra of (functional) programs* was providential in alerting computer programmers that computer languages alone are insufficient, and that only languages which exhibit an *algebra* for reasoning about the objects they purport to describe will be useful in the long run.

The impact of Backus first argument in the computing science and computer architecture communities was considerable, in particular if assessed in quality rather than quantity and in addition to the almost contemporary *structured programming* trend [16]. By contrast, his second argument for changing computer programming was by and large ignored, and only the so-called *algebra of programming* research minorities pursued in this direction. However, the advances in this area throughout the last two decades are impressive and can be fully appreciated by reading a textbook written relatively recently by Bird and de Moor [12]. A comprehensive review of the voluminous literature available in this area can also be found in this book.

Although the need for a pointfree algebra of programming was first identified by Backus, perhaps influenced by Iverson's APL growing popularity in the USA at that time, the idea of reasoning and using mathematics to transform programs is much older and can be traced to the times of McCarthy's work on the foundations of computer programming [54], of Floyd's work on program meaning [22] and of Paterson and Hewitt's *comparative schematology* [77]. Work of the so-called

[16] Even the C programming language and the UNIX operating system, with their implicit functional flavour, may be regarded as subtle outcomes of the "going functional" trend.

program transformation school was already very expressive in the mid 1970s, see for instance references [13].

The mathematics adequate for the effective integration of these related but independent lines of thought was provided by the categorial approach of Manes and Arbib compiled in a textbook [53] which has very strongly influenced the last decade of 20th century theoretical computer science.

A so-called MPC ("Mathematics of Program Construction") community has been among the most active in producing an integrated body of knowledge on the algebra of programming which has found in functional programming an eloquent and paradigmatic medium. Functional programming has a tradition of absorbing fresh results from theoretical computer science, algebra and category theory. Languages such as HASKELL [11] have been competing to integrate the most recent developments and therefore are excellent *prototyping* vehicles in courses on program calculation, as happens with this book.

For fairly recent work on this topic see e.g. [27, 32, 33, 26].

3

RECURSION IN THE POINTFREE STYLE

How useful from a programmer's point of view are the abstract concepts presented in the previous chapter? Recall that a table was presented — table 1 — which records an analogy between abstract type notation and the corresponding data-structures available in common, imperative languages.

This analogy will help in showing how to extend the abstract notation studied thus far towards a most important field of programming: *recursion*. This, however, will be preceeded by a simpler introduction to the subject rooted on very basic and intuitive notions of mathematics.

3.1 MOTIVATION

Where do algorithms come from? From human imagination only? Surely not — they actually emerge from mathematics. In a sense, in the same way one may say that hardware follows the laws of physics (e.g. semiconductor electronics) one might say that software is governed by the laws of mathematics.

This section provides a naive introduction to algorithm analysis and synthesis by showing how a quite elementary class of algorithms — equivalent to for-loops in C or any other imperative language — arise from elementary properties of the underlying maths domain.

We start by showing how the arithmetic operation of multiplying two natural numbers (in $\mathbb{N}_0$) is a for-loop which emerges solely from the algebraic properties of multiplication:

$$\left\{\begin{array}{l} a \times 0 = 0 \\ a \times 1 = a \\ a \times (b + c) = a \times b + a \times c \end{array}\right. \tag{3.1}$$

These properties are known as the *absorption*, *unit* and *distributive* properties of multiplication, respectively.

Start by making $c := 1$ in the third (distributive) property, obtaining $a \times (b+1) \;=\; a \times b + a \times 1$, and then simplify. The second clause is useful in this simplification but it is not required in the final system of two equations,

$$\begin{cases} a \times 0 = 0 \\ a \times (b+1) = a \times b + a \end{cases} \tag{3.2}$$

since it is derivable from the remaining two, for $b := 0$ and property $0 + a = a$ of addition.

System (3.2) *is already* a runnable program in a functional language such as Haskell (among others). The moral of this trivial exercise is that programs *arise* from the underlying maths, instead of being *invented* or coming out of the blue. Novices in functional programming do this kind of reasoning all the time without even noticing it, when writing their first programs. For instance, the function which computes discrete exponentials will scale up the same procedure, thanks to the properties

$$\begin{cases} a^0 = 1 \\ a^1 = a \\ a^{b+c} = a^b \times a^c \end{cases}$$

where the program just developed for multiplication can be re-used, and so and so on.

Type-wise, the multiplication algorithm just derived for natural numbers is not immediate to generalize. Intuitively, it will diverge for b a negative integer and for b a real number less than 1, at least. Argument a, however, does not seem to be constrained.

Indeed, the two arguments a and b will have different types in general. Let us see why and how. Starting by looking at infix operators $(\times)$ and $(+)$ as *curried* operators — recall section 2.15 — we can resort to the corresponding *sections* and write:

$$\begin{cases} (a\times)0 = 0 \\ (a\times)(b+1) = (a+)((a\times)b) \end{cases} \tag{3.3}$$

It can be easily checked that

$$(a\times) \quad = \quad \mathsf{for}\ (a+)\ 0 \tag{3.4}$$

by introducing a **for-loop** combinator given by

$$\begin{cases} \mathsf{for}\ f\ i\ 0 = i \\ \mathsf{for}\ f\ i\ (n+1) = f\ (\mathsf{for}\ f\ i\ n) \end{cases} \tag{3.5}$$

where f is the loop-body and i is the initialization value. In fact, $(\mathsf{for}\ f\ i)n = f^n\ i$, that is, f is iterated n times over the initial value i.

For-loops are a primitive construct available in many programming languages. In C, for instance, one will write something like

```
int mul(int a, int n)
{
int s=0; int i;
for (i=1;i<n+1;i++) {s += a;}
return s;
};
```

for (the uncurried version of) loop for $(a+)\ 0$.

To better understand this construct let us remove variables from both equations in (3.3) by lifting function application to function composition and lifting 0 to the "everywhere 0" (constant) function:

$$\left\{ \begin{array}{l} (a\times) \cdot \underline{0} = \underline{0} \\ (a\times) \cdot (+1) = (+a) \cdot (a\times) \end{array} \right.$$

Using the *junc* ("either") pointfree combinator we merge the two equations into a single one,

$$[(a\times) \cdot \underline{0}\,, (a\times) \cdot (+1)] \quad = \quad [\underline{0}\,, (+a) \cdot (a\times)]$$

— thanks to the Eq-+ rule (2.66) — then single out the common factor $(a\times)$ in the left hand side,

$$(a\times) \cdot [\underline{0}\,, (+1)] \quad = \quad [\underline{0}\,, (+a) \cdot (a\times)]$$

— thanks to +-fusion (2.42) — and finally do a similar *fission* operation on the other side,

$$(a\times) \cdot [\underline{0}\,, (+1)] \quad = \quad [\underline{0}\,, (+a)] \cdot (id + (a\times)) \qquad (3.6)$$

— thanks to +-absorption (2.43).

As we already know, equalities of compositions are nicely drawn as diagrams. That of (3.6) is as follows:

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "(a\times)"'] & A + \mathbb{N}_0 \arrow[l, "{[\underline{0}\,,(+1)]}"'] \arrow[d, "id+(a\times)"] \\
\mathbb{N}_0 & A + \mathbb{N}_0 \arrow[l, "{[\underline{0}\,,(+a)]}"]
\end{tikzcd}$$

Function $(+1)$ is the successor function succ on natural numbers. Type A is any (non-empty) type. For the particular case of $A = 1$, the diagram is more interesting, as $[\underline{0}\,, \mathsf{succ}]$ becomes an isomorphism, telling a *unique* way of building natural numbers:[1]

> *Every natural number in* $\mathbb{N}_0$ *either is* 0 *or the successor of another natural number.*

[1] This is nothing but a re-statement of the well-known *Peano* axioms for the natural numbers. Giuseppe Peano (1858-1932) was a famous Italian mathematician.

We will denote such an isomorphism by *in* and its converse by *out* in the following version of the same diagram

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[r, bend left, "{out=in^\circ}"] \arrow[r, phantom, "\cong"] \arrow[d, "(a\times)"'] & 1+\mathbb{N}_0 \arrow[l, bend left, "{in=[\underline{0}\,,\mathsf{succ}]}"] \arrow[d, "id+(a\times)"] \\
\mathbb{N}_0 & 1+\mathbb{N}_0 \arrow[l, bend left, "{[\underline{0}\,,(+a)]}"]
\end{tikzcd}$$

capturing both the isomorphism and the $(a\times)$ recursive function. By solving the isomorphism equation $out \cdot in = id$ we easily obtain the definition of *out*, the converse of *in* [2]:

$$\begin{aligned}
&out\ 0 = i_1()\\
&out(n+1) = i_2\ n
\end{aligned}$$

Finally, we generalize the target $\mathbb{N}_0$ to any non-empty type B, $(+a)$ to any function $B \xrightarrow{g} B$ and 0 to any constant k in B (this is why B has to be non-empty). The corresponding generalization of $(a\times)$ is denoted by f below:

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[r, bend left, "{out=in^\circ}"] \arrow[r, phantom, "\cong"] \arrow[d, "f"'] & 1+\mathbb{N}_0 \arrow[l, bend left, "{in=[\underline{0}\,,\mathsf{succ}]}"] \arrow[d, "id+f"] \\
B & 1+B \arrow[l, bend left, "{[\underline{k}\,,g]}"]
\end{tikzcd}$$

It turns out that, given k and g, there is a unique solution to the equation (in f) captured by the diagram: $f \cdot in = [\underline{k}\,,g] \cdot (id+f)$. We know this solution already, recall (3.5):

$$f \quad = \quad \mathsf{for}\ g\ k$$

As we have seen earlier on, solution uniqueness is captured by universal properties. In this case we have the following property, which we will refer to by writing "for-loop-universal":

$$f = \mathsf{for}\ g\ k \quad \equiv \quad f \cdot in = [\underline{k}\,,g] \cdot (id+f) \tag{3.7}$$

From this property it is possible to infer a basic theory of for-loops. For instance, by making $f = id$ and solving the for-loop-universal equation (3.7) for g and k we obtain the reflection law:

$$\mathsf{for\ succ\ 0} \quad = \quad id \tag{3.8}$$

This can be compared with the following (useless) program in C:

[2] Note how the singularity of type 1 ensures *out* a function: what would the outcome of *out* 0 be should A be arbitrary?

```
int id(int n)
{
  int s=0; int i;
  for (i=1;i<n+1;i++) {s += 1;}
  return s;
};
```

(Clearly, the value returned in `s` is that of input `n`.)

More knowledge about for-loops can be extracted from (3.7). Later on we will show that these constructs are special cases of a more general concept termed *catamorphism*.[3] In the usual *"banana-bracket"* notation of catamorphisms, to be introduced later, the for-combinator will be written:

$$\mathsf{for}\ g\ k = (\!|\ [\underline{k}, g]\ |\!) \tag{3.9}$$

In the sequel, we shall study the (more general) theory of catamorphisms and come back to for-loops as an instantiation. Then we will understand how more interesting for-loops can be synthesized, for instance those handling more than one "global variable", thanks to catamorphism theory (for instance, the mutual recursion laws).

As a generalization of what we have just seen happening between for-loops and natural numbers, it will be shown that a catamorphism is intimately connected to the data-structure it processes, for instance a finite list (sequence) or a binary tree. A good understanding of such structures is therefore required. We proceed to studying the list data structure first, wherefrom trees stem as natural extensions.

Exercise *3.1. Addition is known to be associative ($a + (b + c) = (a + b) + c$) and have unit 0 ($a + 0 = a$). Following the same strategy that was adopted above for $(a\times)$, show that*

$$(a+) \quad = \quad \mathsf{for}\ \mathsf{succ}\ a \tag{3.10}$$

□

Exercise *3.2. The following* fusion-*law*

$$h \cdot (\mathsf{for}\ g\ k) = \mathsf{for}\ j\ (h\ k) \quad \Leftarrow \quad h \cdot g = j \cdot h \tag{3.11}$$

can be derived from universal-property (3.7) [4]*. Since $(a+) \cdot id = (a+)$, provide an alternative derivation of (3.10) using the fusion-law above.*

□

Exercise *3.3. From (3.4) and fusion-law (3.11) infer: $(a*) \cdot \mathsf{succ} = \mathsf{for}\ a\ (a+)$.*

□

[3] See e.g. section 3.6.

[4] A generalization of this property will be derived in section 3.12.

***Exercise** 3.4. Show that* $f = \mathsf{for}\ \underline{k}\ k$ *and* $g = \mathsf{for}\ id\ k$ *are the same program (function).*

□

***Exercise** 3.5. Generic function* $k = \mathsf{for}\ f\ i$ *can be encoded in the syntax of C by writing*

```
int k(int n) {
  int r=i;
  int x;
  for (x=1;x<n+1;x++) {r=f(r);}
  return r;
};
```

for some predefined f*. Encode the functions* f *and* g *of exercise 3.4 in C and compare them.*

□

3.2 FROM NATURAL NUMBERS TO FINITE SEQUENCES

Let us consider a very common data-structure in programming: "linked-lists". In PASCAL one will write

```
L = ^N;
N = record
  first: A;
  next: ^N
end;
```

to specify such a data-structure `L`. This consists of a pointer to a *node* (`N`), where a node is a record structure which puts some predefined type A together with a pointer to another node, and so on. In the C programming language, every $x \in L$ will be declared as `L x` in the context of datatype definition

```
typedef struct N {
     A first;
     struct N *next;
} *L;
```

and so on.

What interests us in such "first year programming course" datatype declarations? Records and pointers have already been dealt with in table 1. So we can use this table to find the abstract version of datatype

L, by replacing pointers by the "$1 + \cdots$" notation and records (*structs*) by the "$\ldots \times \ldots$" notation:

$$\left\{ \begin{array}{lcl} L & = & 1 + N \\ N & = & A \times (1 + N) \end{array} \right. \tag{3.12}$$

We obtain a system of two equations on unknowns L and N, in which L's dependence on N can be removed by substitution:

$$\left\{ \begin{array}{lcl} L & = & 1 + N \\ N & = & A \times (1 + N) \end{array} \right.$$

$\equiv$ { substituting L for $1 + N$ in the second equation }

$$\left\{ \begin{array}{lcl} L & = & 1 + N \\ N & = & A \times L \end{array} \right.$$

$\equiv$ { substituting $A \times L$ for N in the first equation }

$$\left\{ \begin{array}{lcl} L & = & 1 + A \times L \\ N & = & A \times L \end{array} \right.$$

System (3.12) is thus equivalent to:

$$\left\{ \begin{array}{lcl} L & = & 1 + A \times L \\ N & = & A \times (1 + N) \end{array} \right. \tag{3.13}$$

Intuitively, L abstracts the "possibly empty" linked-list of elements of type A, while N abstracts the "non-empty" linked-list of elements of type A. Note that L and N are independent of each other, but also that each depends on itself. Can we solve these equations in a way such that we obtain "solutions" for L and N, in the same way we do with school equations such as, for instance,

$$x = 1 + \frac{x}{2} \qquad ? \tag{3.14}$$

Concerning this equation, let us recall how we would go about it in school mathematics:

$$x = 1 + \frac{x}{2}$$

$\equiv$ { adding $-\frac{x}{2}$ to both sides of the equation }

$$x - \frac{x}{2} = 1 + \frac{x}{2} - \frac{x}{2}$$

$\equiv$ { $-\frac{x}{2}$ cancels $\frac{x}{2}$ }

$$x - \frac{x}{2} = 1$$

$\equiv$ { multiplying both sides of the equation by 2 *etc.* }

$$2 \times x - x = 2$$

$\equiv$ { subtraction }

$$x = 2$$

We very quickly get solution $x = 2$. However, many steps were omitted from the actual calculation. This unfolds into the longer sequence of more elementary steps which follows, in which notation $a - b$ abbreviates $a + (-b)$ and $\frac{a}{b}$ abbreviates $a \times \frac{1}{b}$, for $b \neq 0$:

$$
\begin{array}{ll}
& x = 1 + \dfrac{x}{2} \\
\equiv & \qquad \{ \text{ adding } -\frac{x}{2} \text{ to both sides of the equation } \} \\
& x - \dfrac{x}{2} = (1 + \dfrac{x}{2}) - \dfrac{x}{2} \\
\equiv & \qquad \{ + \text{ is associative } \} \\
& x - \dfrac{x}{2} = 1 + (\dfrac{x}{2} - \dfrac{x}{2}) \\
\equiv & \qquad \{ -\frac{x}{2} \text{ is the additive inverse of } \frac{x}{2} \} \\
& x - \dfrac{x}{2} = 1 + 0 \\
\equiv & \qquad \{ \text{ 0 is the unit of addition } \} \\
& x - \dfrac{x}{2} = 1 \\
\equiv & \qquad \{ \text{ multiplying both sides of the equation by 2 } \} \\
& 2 \times (x - \dfrac{x}{2}) = 2 \times 1 \\
\equiv & \qquad \{ \text{ 1 is the unit of multiplication } \} \\
& 2 \times (x - \dfrac{x}{2}) = 2 \\
\equiv & \qquad \{ \text{ multiplication distributes over addition } \} \\
& 2 \times x - 2 \times \dfrac{x}{2} = 2 \\
\equiv & \qquad \{ \text{ 2 cancels its inverse } \frac{1}{2} \} \\
& 2 \times x - 1 \times x = 2 \\
\equiv & \qquad \{ \text{ multiplication distributes over addition } \} \\
& (2 - 1) \times x = 2 \\
\equiv & \qquad \{ 2 - 1 = 1 \text{ and 1 is the unit of multiplication } \} \\
& x = 2
\end{array}
$$

Back to (3.13), we would like to submit each of the equations, *e.g.*

$$L \quad = \quad 1 + A \times L \tag{3.15}$$

to a similar reasoning. Can we do it? The analogy which can be found between this equation and (3.14) goes beyond pattern similarity. From chapter 2 we know that many properties required in the reasoning above hold in the context of (3.15), provided the "=" sign is replaced by the "≅" sign, that of set-theoretical isomorphism. Recall that, for instance, $+$ is associative (2.48), 0 is the unit of addition (2.55), 1 is the unit of multiplication (2.57), multiplication distributes over addition (2.52) *etc.* Moreover, the first step above assumed that addition is compatible (monotonic) with respect to equality,

$$\frac{\begin{array}{ccc} a & = & b \\ c & = & d \end{array}}{\begin{array}{ccc} a+c & = & b+d \end{array}}$$

a fact which still holds when numeric equality gives place to isomorphism and numeric addition gives place to coproduct:

$$\frac{\begin{array}{ccc} A & \cong & B \\ C & \cong & D \end{array}}{\begin{array}{ccc} A+C & \cong & B+D \end{array}}$$

— recall (2.46) for isos f and g.

Unfortunately, the main steps in the reasoning above are concerned with two basic *cancellation properties*

$$\begin{array}{lcl} x+b=c & \equiv & x=c-b \\ x \times b = c & \equiv & x=\frac{c}{b} \quad (b \neq 0) \end{array}$$

which hold about numbers but do not hold about datatypes. In fact, neither products nor coproducts have arbitrary inverses [5], and so we cannot "calculate by cancellation". How do we circumvent this limitation?

Just think of how we would have gone about (3.14) in case we didn't know about the *cancellation properties*: we would be bound to the x by $1+\frac{x}{2}$ substitution plus the other properties. By performing such a substitution over and over again we would obtain...

$$\begin{array}{ll}
& x = 1+\frac{x}{2} \\
\equiv & \quad \{ \ x \text{ by } 1+\frac{x}{2} \text{ substitution followed by simplification} \ \} \\
& x = 1+\frac{1+\frac{x}{2}}{2} = 1+\frac{1}{2}+\frac{x}{4} \\
\equiv & \quad \{ \text{ the same as above } \} \\
& x = 1+\frac{1}{2}+\frac{1+\frac{x}{2}}{4} = 1+\frac{1}{2}+\frac{1}{4}+\frac{x}{8} \\
\equiv & \quad \{ \text{ over and over again, } n\text{-times } \} \\
& \ldots \\
\equiv & \quad \{ \text{ simplification } \} \\
& x = \sum_{i=0}^{n} \frac{1}{2^i} + \frac{x}{2^{n+1}}
\end{array}$$

[5] The initial and terminal datatypes do have inverses — 0 is its own "additive inverse" and 1 is its own "multiplicative inverse" — but not all the others.

$$
\begin{array}{ll}
\equiv & \quad \{ \text{ sum of } n \text{ first terms of a geometric progression } \} \\
& x = (2 - \frac{1}{2^n}) + \frac{x}{2^{n+1}} \\
\equiv & \quad \{ \text{ let } n \to \infty \} \\
& x = (2 - 0) + 0 \\
\equiv & \quad \{ \text{ simplification } \} \\
& x = 2
\end{array}
$$

Clearly, this is a much more complicated way of finding solution $x = 2$ for equation (3.14). But we would have loved it in case it were the only known way, and this is precisely what happens with respect to (3.15). In this case we have:

$$
\begin{array}{ll}
& L = 1 + A \times L \\
\equiv & \quad \{ \text{ substitution of } 1 + A \times L \text{ for } L \} \\
& L = 1 + A \times (1 + A \times L) \\
\equiv & \quad \{ \text{ distributive property (2.52) } \} \\
& L \cong 1 + A \times 1 + A \times (A \times L) \\
\equiv & \quad \{ \text{ unit of product (2.57) and associativity of product (2.34) } \} \\
& L \cong 1 + A + (A \times A) \times L \\
\equiv & \quad \{ \text{ by (2.98), (2.100) and (2.104) } \} \\
& L \cong A^0 + A^1 + A^2 \times L \\
\equiv & \quad \{ \text{ another substitution as above and similar simplifications } \} \\
& L \cong A^0 + A^1 + A^2 + A^3 \times L \\
\equiv & \quad \{ \text{ after } (n+1)\text{-many similar steps } \} \\
& L \cong \sum_{i=0}^{n} A^i + A^{n+1} \times L
\end{array}
$$

Bearing a large n in mind, let us deliberately (but temporarily) ignore term $A^{n+1} \times L$. Then L will be isomorphic to the sum of n-many contributions A^i,

$$
L \cong \sum_{i=0}^{n} A^i
$$

each of them consisting of i-long tuples, or *sequences*, of values of A. (Number i is said to be the *length* of any sequence in A^i.) Such sequences will be denoted by enumerating their elements between square brackets, for instance the *empty sequence* $[\,]$ which is the only inhabitant in A^0, the two element sequence $[a_1, a_2]$ which belongs to A^2 provided $a_1, a_2 \in A$, and so on. Note that all such contributions are mutually disjoint, that is, $A^i \cap A^j = \emptyset$ wherever $i \neq j$. (In other words, a sequence of length i is never a sequence of length j, for $i \neq j$.)

If we join all contributions A^i into a single set, we obtain the set of all *finite sequences* on A, denoted by $A^\star$ and defined as follows:

$$A^\star \stackrel{\mathrm{def}}{=} \bigcup_{i \geqslant 0} A^i \tag{3.16}$$

The intuition behind taking the limit in the numeric calculation above was that term $\frac{x}{2^{n+1}}$ was getting smaller and smaller as n went larger and larger and, "in the limit", it could be ignored. By analogy, taking a similar limit in the calculation just sketched above will mean that, for a "sufficiently large" n, the sequences in A^n are so long that it is very unlikely that we will ever use them! So, for $n \to \infty$ we obtain

$$L \cong \sum_{i=0}^{\infty} A^i$$

Because $\sum_{i=0}^{\infty} A^i$ is isomorphic to $\bigcup_{i=0}^{\infty} A^i$ (see exercise 2.32), we finally have:

$$L \cong A^\star$$

All in all, we have obtained $A^\star$ as a solution to equation (3.15). In other words, datatype L is isomorphic to the datatype which contains all finite sequences of some predefined datatype A. This corresponds to the HASKELL `[a]` datatype, in general. Recall that we started from the "linked-list datatype" expressed in PASCAL or C. In fact, wherever the C programmer thinks of linked-lists, the HASKELL programmer will think of finite sequences.

But, what does equation (3.15) mean in fact? Is $A^\star$ the only solution to this equation? Back to the numeric field, we know of equations which have more than one solution — for instance $x = \frac{x^2+3}{4}$, which admits two solutions 1 and 3 —, which have no solution at all — for instance $x = x + 1$ —, or which admit an infinite number of — for instance $x = x$.

We will address these topics in the next section about *inductive* datatypes and — more generally — in chapter 8, where the formal semantics of recursion will be made explicit. This is where the "limit" constructions used informally in this section will be shown to make sense.

3.3 INTRODUCING INDUCTIVE DATATYPES

Datatype L as defined by (3.15) is said to be *recursive* because L "recurs" in the definition of L itself [6]. From the discussion above, it is clear that set-theoretical equality "$=$" in this equation should give place to set-theoretical isomorphism ("$\cong$"):

$$L \cong 1 + A \times L \tag{3.17}$$

[6] By analogy, we may regard (3.14) as a "recursive definition" of number 2.

Which isomorphism $L \xleftarrow{in} 1 + A \times L$ do we expect to witness (3.15)? This will depend on which particular solution to (3.15) we are thinking of. So far we have seen only one, $A^\star$. By recalling the notion of *algebra* of a datatype (section 2.18), so we may rephrase the question as: which algebra

$$A^\star \xleftarrow{in} 1 + A \times A^\star$$

do we expect to witness the tautology which arises from (3.15) by replacing unknown L with solution $A^\star$, that is

$$A^\star \quad \cong \quad 1 + A \times A^\star \qquad\qquad ?$$

It will have to be of the form $in = [in_1, in_2]$ as depicted by the following diagram:

$$\begin{tikzcd}
1 \arrow[r, "i_1"] \arrow[dr, "in_1"'] & 1 + A \times A^\star \arrow[d, "in"] & A \times A^\star \arrow[l, "i_2"'] \arrow[dl, "in_2"] \\
& A^\star &
\end{tikzcd} \tag{3.18}$$

Arrows in_1 and in_2 can be guessed rather intuitively: $in_1 = \underline{[\,]}$, which will express the "NIL pointer" by the empty sequence, at $A^\star$ level, and $in_2 = cons$, where $cons$ is the standard "left append" sequence constructor, which we for the moment introduce rather informally as follows:

$$\begin{array}{l} cons : A \times A^\star \to A^\star \\ cons(a, [a_1, \ldots, a_n]) = [a, a_1, \ldots, a_n] \end{array} \tag{3.19}$$

In a diagram:

$$\begin{tikzcd}
1 \arrow[r, "i_1"] \arrow[dr, "\underline{[\,]}"'] & 1 + A \times A^\star \arrow[d, "{[\underline{[\,]}, cons]}"] & A \times A^\star \arrow[l, "i_2"'] \arrow[dl, "cons"] \\
& A^\star &
\end{tikzcd} \tag{3.20}$$

Of course, for in to be iso it needs to have an inverse, which is not hard to guess,

$$out \overset{\mathrm{def}}{=} (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?) \tag{3.21}$$

where sequence operators hd (*head of a nonempty sequence*) and tl (*tail of a nonempty sequence*) are (again informally) described as follows:

$$\begin{array}{l} hd : A^\star \to A \\ hd\,[a_1, a_2, \ldots, a_n] = a_1 \end{array} \tag{3.22}$$

$$\begin{array}{l} tl : A^\star \to A^\star \\ tl\,[a_1, a_2, \ldots, a_n] = [a_2, \ldots, a_n] \end{array} \tag{3.23}$$

Showing that *in* and *out* are each other inverses is not a hard task either:

$$
\begin{array}{ll}
 & in \cdot out = id \\
\equiv & \qquad \{ \text{ definitions of } in \text{ and } out \ \} \\
 & [\underline{[\,]}, cons] \cdot (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?) = id \\
\equiv & \qquad \{ \text{ +-absorption (2.43) and (2.15) } \} \\
 & [\underline{[\,]}, cons \cdot \langle hd, tl \rangle] \cdot (=_{[\,]}?) = id \\
\equiv & \qquad \{ \text{ property of sequences: } cons(hd\,s, tl\,s) = s \ \} \\
 & [\underline{[\,]}, id] \cdot (=_{[\,]}?) = id \\
\equiv & \qquad \{ \text{ going pointwise (2.69) } \} \\
 & \left\{ \begin{array}{rcl} =_{[\,]}\ a & \Rightarrow & [\underline{[\,]}, id]\,(i_1\,a) \\ \neg(=_{[\,]}\ a) & \Rightarrow & [\underline{[\,]}, id]\,(i_2\,a) \end{array} \right. = a \\
\equiv & \qquad \{ \text{ +-cancellation (2.40) } \} \\
 & \left\{ \begin{array}{rcl} =_{[\,]}\ a & \Rightarrow & \underline{[\,]}\,a \\ \neg(=_{[\,]}\ a) & \Rightarrow & id\,a \end{array} \right. = a \\
\equiv & \qquad \{ \ a = [\,] \text{ in one case and identity function (2.9) in the other } \} \\
 & \left\{ \begin{array}{rcl} a = [\,] & \Rightarrow & a \\ \neg(a = [\,]) & \Rightarrow & a \end{array} \right. = a \\
\equiv & \qquad \{ \text{ property } (p \rightarrow f, f) = f \text{ holds } \} \\
 & a = a
\end{array}
$$

A comment on the particular choice of terminology above: symbol *in* suggests that we are going inside, or constructing (synthesizing) values of $A^\star$; symbol *out* suggests that we are going out, or destructing (analyzing) values of $A^\star$. We shall often resort to this duality in the sequel.

Are there more solutions to equation (3.17)? In trying to implement this equation, a HASKELL programmer could have written, after the declaration of type A, the following datatype declaration:

data $L = Nil\ () \mid Cons\ (A, L)$

which, as we have seen in section 2.18, can be written simply as

data $L = Nil \mid Cons\ (A, L)$

and generates diagram

$$\begin{tikzcd}
1 \arrow[r, "i_1"] \arrow[dr, "\underline{Nil}"'] & 1 + A \times L \arrow[d, "in'"] & A \times L \arrow[l, "i_2"'] \arrow[dl, "Cons"] \\
 & L &
\end{tikzcd} \qquad (3.24)$$

leading to algebra $in' = [\underline{Nil}, Cons]$.

HASKELL seems to have generated another solution for the equation, which it calls L. To avoid the inevitable confusion between this symbol denoting the newly created datatype and symbol L in equation (3.17), which denotes a mathematical variable, let us use symbol T to denote the former (T stands for "type"). This can be coped with very simply by writing `T` instead of `L` above:

$$\textbf{data}\ \mathsf{T} = Nil \mid Cons\ (A, \mathsf{T}) \qquad (3.25)$$

In order to make T more explicit, we will write in_{T} instead of in'.

Some questions are on demand at this point. First of all, what is datatype T? What are its inhabitants? Next, is $\mathsf{T} \xleftarrow{in_{\mathsf{T}}} 1 + A \times \mathsf{T}$ an iso or not?

HASKELL will help us to answer these questions. Suppose that A is a primitive numeric datatype, and that we add `deriving Show` to (3.25) so that we can "see" the inhabitants of the T datatype. The information associated to T is thus:

```
Main> :i T
-- type constructor
data T

-- constructors:
Nil :: T
Cons :: (A,T) -> T

-- instances:
instance Show T
instance Eval T
```

By typing `Nil`

```
Main> Nil
Nil :: T
```

we confirm that Nil is itself an inhabitant of T, and by typing `Cons`

```
Main> Cons
<<function>> :: (A,T) -> T
```

we realize that $Cons$ is not so (as expected), but it can be used to build such inhabitants, for instance:

```
Main> Cons(1,Nil)
Cons (1,Nil) :: T
```

or

```
Main> Cons(2,Cons(1,Nil))
Cons (2,Cons (1,Nil)) :: T
```

etc. We conclude that *expressions* involving *Nil* and *Cons* are inhabitants of type T. Are these the *only* ones? The answer is *yes* because, by design of the HASKELL language, the constructors of type T will remain fixed once its declaration is interpreted, that is, no further constructor can be added to T. Does in_{T} have an inverse? Yes, its inverse is coalgebra

$$\begin{array}{l} out_{\mathsf{T}} : \mathsf{T} \to 1 + A \times \mathsf{T} \\ out_{\mathsf{T}}\ Nil = i_1\ \text{NIL} \\ out_{\mathsf{T}}(Cons(a,l)) = i_2(a,l) \end{array} \tag{3.26}$$

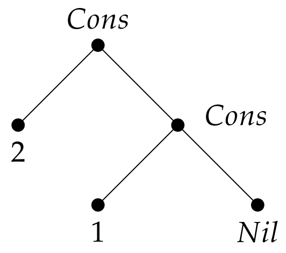
which can be straightforwardly encoded in HASKELL using the `Either` realization of $+$ (recall sections 2.9 and 2.18):

$$\begin{array}{l} \mathsf{out}_{\mathsf{T}} :: \mathsf{T} \to Either\ ()\ (A, \mathsf{T}) \\ \mathsf{out}_{\mathsf{T}}\ Nil = i_1\ () \\ \mathsf{out}_{\mathsf{T}}\ (Cons\ (a,l)) = i_2\ (a,l) \end{array}$$

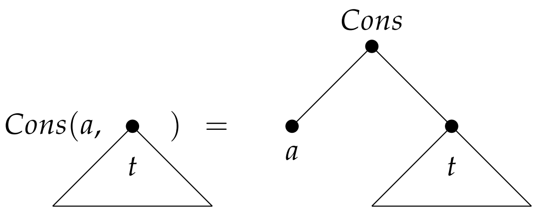
In summary, isomorphism

$$\begin{tikzcd} \mathsf{T} \arrow[rr, bend left, "out_{\mathsf{T}}"] \arrow[rr, phantom, "\cong"] & & 1 + A \times \mathsf{T} \arrow[ll, bend left, "in_{\mathsf{T}}"] \end{tikzcd} \tag{3.27}$$

holds, where datatype T is inhabited by symbolic expressions which we may visualize very conveniently as trees, for instance



picturing expression $Cons(2, Cons(1, Nil))$. *Nil* is the empty tree and *Cons* may be regarded as the operation which adds a new root and a new branch, say a, to a tree t:



The choice of symbols `T`, `Nil` and `Cons` was rather arbitrary in (3.25). Therefore, an alternative declaration such as, for instance,

$$\textbf{data}\ U = Stop \mid Join\ (A, U) \tag{3.28}$$

would have been perfectly acceptable, generating another solution for the equation under algebra $[\underline{Stop}, Join]$. It is easy to check that (3.28) is but a renaming of Nil to $Stop$ and of $Cons$ to $Join$. Therefore, both datatypes are isomorphic, or "abstractly the same".

Indeed, any other datatype X *inductively* defined by a constant and a binary constructor accepting A and X as parameters will be a solution to the equation. Because we are just renaming symbols in a consistent way, all such solutions are abstractly the same. All of them capture the abstract notion of a *list* of symbols.

We wrote "inductively" above because the set of all expressions (trees) which inhabit the type is defined by induction. Such types are called *inductive* and we shall have a lot more to say about them in chapter 8 .

***Exercise** 3.6. Obviously,*

```
either (const []) (:)
```

does not work as a HASKELL *realization of the mediating arrow in diagram (3.20). What do you need to write instead?*

□

3.4 OBSERVING AN INDUCTIVE DATATYPE

Suppose that one is asked to express a particular *observation* of an inductive such as T (3.25), that is, a function of signature $B \xleftarrow{f} \mathsf{T}$ for some target type B. Suppose, for instance, that A is $\mathbb{N}_0$ (the set of all non-negative integers) and that we want to add all elements which occur in a T-list. Of course, we have to ensure that addition is available in $\mathbb{N}_0$,

$$add : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$$
$$add(x,y) \stackrel{\text{def}}{=} x + y$$

and that $0 \in \mathbb{N}_0$ is a value denoting "the addition of nothing". So constant arrow $\mathbb{N}_0 \xleftarrow{\underline{0}} 1$ is available. Of course, $add(0,x) = add(x,0) = x$ holds, for all $x \in \mathbb{N}_0$. This property means that $\mathbb{N}_0$, together with operator add and constant 0, forms a *monoid*, a very important algebraic structure in computing which will be exploited intensively later in this book. The following arrow "packaging" $\mathbb{N}_0$, add and $\underline{0}$,

$$\mathbb{N}_0 \xleftarrow{[\underline{0}, add]} 1 + \mathbb{N}_0 \times \mathbb{N}_0 \tag{3.29}$$

is a convenient way to express such a structure. Combining this arrow with the algebra

$$\mathsf{T} \xleftarrow{in_{\mathsf{T}}} 1 + \mathbb{N}_0 \times \mathsf{T} \tag{3.30}$$

which defines T, and the function f we want to define, the target of which is $B = \mathbb{N}_0$, we get the almost closed diagram which follows, in which only the dashed arrow is yet to be filled in:

$$\begin{tikzcd}
\mathsf{T} \arrow[d, "f"'] & & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[ll, "in_{\mathsf{T}}"'] \arrow[d, dashed] \\
\mathbb{N}_0 & & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \arrow[ll, "{[\underline{0}, add]}"]
\end{tikzcd} \tag{3.31}$$

We know that $in_{\mathsf{T}} = [\underline{Nil}, Cons]$. A pattern for the missing arrow is not difficult to guess: in the same way f bridges T and $\mathbb{N}_0$ on the left-hand side, it will do the same job on the right-hand side. So pattern $\cdots + \cdots \times f$ comes to mind (recall section 2.10), where the "$\cdots$" are very naturally filled in by identity functions. All in all, we obtain diagram

$$\begin{tikzcd}
\mathsf{T} \arrow[d, "f"'] & & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[ll, "{[\underline{Nil}, Cons]}"'] \arrow[d, "id + id \times f"] \\
\mathbb{N}_0 & & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \arrow[ll, "{[\underline{0}, add]}"]
\end{tikzcd} \tag{3.32}$$

which pictures the following property of f

$$f \cdot [\underline{Nil}, Cons] \quad = \quad [\underline{0}, add] \cdot (id + id \times f) \tag{3.33}$$

and is easy to convert to pointwise notation:

$$\begin{array}{ll}
 & f \cdot [\underline{Nil}, Cons] = [\underline{0}, add] \cdot (id + id \times f) \\
\equiv & \quad \{ \text{ (2.42) on the lefthand side, (2.43) and identity } id \text{ on the righthand side } \} \\
 & [f \cdot \underline{Nil}, f \cdot Cons] = [\underline{0}, add \cdot (id \times f)] \\
\equiv & \quad \{ \textit{ either} \text{ structural equality (2.66) } \} \\
 & \left\{ \begin{array}{l} f \cdot \underline{Nil} = \underline{0} \\ f \cdot Cons = add \cdot (id \times f) \end{array} \right. \\
\equiv & \quad \{ \text{ going pointwise } \} \\
 & \left\{ \begin{array}{l} (f \cdot \underline{Nil}) x = \underline{0}\, x \\ (f \cdot Cons)(a, x) = (add \cdot (id \times f))(a, x) \end{array} \right. \\
\equiv & \quad \{ \text{ composition (2.6), constant (2.12), product (2.24) and definition of } add \ \} \\
 & \left\{ \begin{array}{l} f\, Nil = 0 \\ f(Cons(a, x)) = a + f\, x \end{array} \right.
\end{array}$$

Note that we could have used out_{T} in diagram (3.31),

$$\begin{tikzcd}
\mathsf{T} \arrow[d, "f"'] \arrow[rr, "out_{\mathsf{T}}"] & & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[d, "id + id \times f"] \\
\mathbb{N}_0 & & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \arrow[ll, "{[\underline{0}, add]}"]
\end{tikzcd} \tag{3.34}$$

obtaining another version of the *definition* of f,

$$f \quad = \quad [\underline{0}, add] \cdot (id + id \times f) \cdot out_{\mathsf{T}} \tag{3.35}$$

which would lead to exactly the same pointwise recursive definition:

$$\begin{array}{ll}
& f = [\underline{0}, add] \cdot (id + id \times f) \cdot out_{\mathsf{T}} \\
\equiv & \quad \{ \text{ (2.43) and identity } id \text{ on the righthand side } \} \\
& f = [\underline{0}, add \cdot (id \times f)] \cdot out_{\mathsf{T}} \\
\equiv & \quad \{ \text{ going pointwise on } out_{\mathsf{T}} \text{ (3.26) } \} \\
& \left\{ \begin{array}{l} f\, Nil = ([\underline{0}, add \cdot (id \times f)] \cdot out_{\mathsf{T}}) Nil \\ f(Cons(a,x)) = ([\underline{0}, add \cdot (id \times f)] \cdot out_{\mathsf{T}})(a,x) \end{array} \right. \\
\equiv & \quad \{ \text{ definition of } out_{\mathsf{T}} \text{ (3.26) } \} \\
& \left\{ \begin{array}{l} f\, Nil = ([\underline{0}, add \cdot (id \times f)] \cdot i_1) Nil \\ f(Cons(a,x)) = ([\underline{0}, add \cdot (id \times f)] \cdot i_2)(a,x) \end{array} \right. \\
\equiv & \quad \{ \text{ +-cancellation (2.40) } \} \\
& \left\{ \begin{array}{l} f\, Nil = \underline{0}\, Nil \\ f(Cons(a,x)) = (add \cdot (id \times f))\,(a,x) \end{array} \right. \\
\equiv & \quad \{ \text{ simplification } \} \\
& \left\{ \begin{array}{l} f\, Nil = 0 \\ f(Cons(a,x)) = a + f\, x \end{array} \right.
\end{array}$$

Pointwise f mirrors the structure of type T in having as many definition clauses as constructors in T. Such functions are said to be defined *by induction on* the structure of their input type. If we repeat this calculation for $\mathbb{N}_0{}^{\star}$ instead of T, that is, for

$$out = (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?)$$

— recall (3.21) — taking place of out_{T}, we get a "more algorithmic" version of f:

$$\begin{array}{ll}
& f = [\underline{0}, add] \cdot (id + id \times f) \cdot (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?) \\
\equiv & \quad \{ \text{ +-functor (2.44), identity and } \times\text{-absorption (2.27) } \} \\
& f = [\underline{0}, add] \cdot (! + \langle hd, f \cdot tl \rangle) \cdot (=_{[\,]}?) \\
\equiv & \quad \{ \text{ +-absorption (2.43) and constant } \underline{0} \text{ } \} \\
& f = [\underline{0}, add \cdot \langle hd, f \cdot tl \rangle] \cdot (=_{[\,]}?) \\
\equiv & \quad \{ \text{ going pointwise on guard } =_{[\,]}? \text{ (2.69) and simplifying } \} \\
& f\, l = \left\{ \begin{array}{rcl} l = [\,] & \Rightarrow & \underline{0}\, l \\ \neg(l = [\,]) & \Rightarrow & (add \cdot \langle hd, f \cdot tl \rangle)\, l \end{array} \right. \\
\equiv & \quad \{ \text{ simplification } \} \\
& f\, l = \left\{ \begin{array}{rcl} l = [\,] & \Rightarrow & 0 \\ \neg(l = [\,]) & \Rightarrow & hd\, l + f(tl\, l) \end{array} \right.
\end{array}$$

The outcome of this calculation can be encoded in HASKELL syntax as

$$\begin{array}{l} f\ l \mid l \equiv [\,] = 0 \\ \quad \mid \mathsf{otherwise} = \mathsf{head}\ l + f\ (\mathsf{tail}\ l) \end{array}$$

or

$$f\ l = \mathbf{if}\ l \equiv [\,]\ \mathbf{then}\ 0\ \mathbf{else}\ \mathsf{head}\ l + f\ (\mathsf{tail}\ l)$$

both requiring the equality predicate $\equiv$ and destructors head and tail.

3.5 SYNTHESIZING AN INDUCTIVE DATATYPE

The issue which concerns us in this section dualizes what we have just dealt with: instead of analyzing or *observing* an inductive type such as T (3.25), we want to be able to synthesize (generate) particular inhabitants of T. In other words, we want to be able to specify functions with signature $B \xrightarrow{f} \mathsf{T}$ for some given source type B. Let $B = \mathbb{N}_0$ and suppose we want f to generate, for a given natural number $n > 0$, the list containing all numbers less or equal to n in decreasing order

$$Cons(n, Cons(n-1, Cons(\ldots, Nil)))$$

or the empty list Nil, in case $n = 0$.

Let us try and draw a diagram similar to (3.34) applicable to the new situation. In trying to "re-use" this diagram, it is immediate that arrow f should be reversed. Bearing duality in mind, we may feel tempted to reverse all arrows just to see what happens. Identity functions are their own inverses, and in_T takes the place of out_T:

$$\begin{tikzcd}
\mathsf{T} & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[l, "in_\mathsf{T}"'] \\
\mathbb{N}_0 \arrow[u, "f"] \arrow[r, dotted] & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \arrow[u, "id+id\times f"']
\end{tikzcd}$$

Interestingly enough, the bottom arrow is the one which is not obvious to reverse, meaning that we have to "invent" a particular destructor of $\mathbb{N}_0$, say

$$\mathbb{N}_0 \xrightarrow{g} 1 + \mathbb{N}_0 \times \mathbb{N}_0$$

fitting in the diagram and *generating* the particular computational effect we have in mind. Once we do this, a recursive definition for f will pop out immediately,

$$f = in_\mathsf{T} \cdot (id + id \times f) \cdot g \tag{3.36}$$

which is equivalent to:

$$f = [\underline{Nil}, Cons \cdot (id \times f)] \cdot g \tag{3.37}$$

Because we want $f\,0 = Nil$ to hold, g (the actual generator of the computation) should distinguish input 0 from all the others. One thus decomposes g as follows,

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[r, "=_0?"] \arrow[rr, bend right, "g"'] & \mathbb{N}_0 + \mathbb{N}_0 \arrow[r, "!+h"] & 1 + \mathbb{N}_0 \times \mathbb{N}_0
\end{tikzcd}$$

leaving h to fill in. This will be a *split* providing, on the lefthand side, for the value to be *Cons*'ed to the output and, on the righthand side, for the "seed" to the next recursive call. Since we want the output values to be produced contiguously and in decreasing order, we may define $h = \langle id, pred\rangle$ where, for $n > 0$,

$$pred\, n \stackrel{\mathrm{def}}{=} n - 1 \tag{3.38}$$

computes the *predecessor* of n. Altogether, we have synthesized

$$g \;=\; (! + \langle id, pred\rangle) \cdot (=_0?) \tag{3.39}$$

Filling this in (3.37) we get

$$\begin{array}{ll}
& f = [\underline{Nil}, Cons \cdot (id \times f)] \cdot (! + \langle id, pred\rangle) \cdot (=_0?) \\
\equiv & \quad \{ \text{ +-absorption (2.43) followed by } \times\text{-absorption (2.27) } \textit{etc.} \ \} \\
& f = [\underline{Nil}, Cons \cdot \langle id, f \cdot pred\rangle] \cdot (=_0?) \\
\equiv & \quad \{ \text{ going pointwise on guard } =_0? \text{ (2.69) and simplifying } \} \\
& f\,n = \left\{ \begin{array}{rcl} n = 0 & \Rightarrow & Nil \\ \neg(n = 0) & \Rightarrow & Cons(n, f\,(n-1)) \end{array} \right.
\end{array}$$

which matches the function we had in mind:

```
f n
  | n ≡ 0 = Nil
  | otherwise = Cons (n,f (n − 1))
```

We shall see briefly that the constructions of the f function adding up a list of numbers in the previous section and, in this section, of the f function generating a list of numbers are very standard in algorithm design and can be broadly generalized. Let us first introduce some standard terminology.

3.6 INTRODUCING (LIST) CATAS, ANAS AND HYLOS

Suppose that, back to section 3.4, we want to *multiply*, rather than add, the elements occurring in lists of type T (3.25). How much of the program synthesis effort presented there can be reused in the design of the new function?

It is intuitive that only the bottom arrow $\mathbb{N}_0 \xleftarrow{[\underline{0}, add]} 1 + \mathbb{N}_0 \times \mathbb{N}_0$ of diagram (3.34) needs to be replaced, because this is the only place where we can specify that target datatype $\mathbb{N}_0$ is now regarded as the carrier of another (multiplicative rather than additive) monoidal structure,

$$\mathbb{N}_0 \xleftarrow{[\underline{1},mul]} 1 + \mathbb{N}_0 \times \mathbb{N}_0 \tag{3.40}$$

for $mul(x,y) \stackrel{\rm def}{=} x\ y$. We are saying that the argument list is now to be reduced by the multiplication operator and that output value 1 is expected as the result of "nothing left to multiply".

Moreover, in the previous section we might have wanted our number-list generator to produce the list of even numbers smaller than a given number, in decreasing order (see exercise 3.9). Intuition will once again help us in deciding that only arrow g in (3.36) needs to be updated.

The following diagrams generalize both constructions by leaving such bottom arrows unspecified,

$$\begin{tikzcd}
\mathsf{T} \arrow[r, "out_{\mathsf{T}}"] \arrow[d, "f"'] & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[d, "id+id\times f"] & \mathsf{T} & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[l, "in_{\mathsf{T}}"'] \\
B & 1 + \mathbb{N}_0 \times B \arrow[l, "g"] & B \arrow[u, "f"] \arrow[r, "g"'] & 1 + \mathbb{N}_0 \times B \arrow[u, "id+id\times f"']
\end{tikzcd} \tag{3.41}$$

and express their duality (*cf.* the directions of the arrows). It so happens that, for each of these diagrams, f is uniquely dependent on the g arrow, that is to say, each particular instantiation of g will determine the corresponding f. So both gs can be regarded as "seeds" or "genetic material" of the f functions they uniquely define [7].

CATAS AND ANAS Following the standard terminology, we express these facts by writing $f = (\!|\, g \,|\!)$ with respect to the lefthand side diagram and by writing $f = [\!(g)\!]$ with respect to the righthand side diagram. Read $(\!|\, g \,|\!)$ as "the T-*catamorphism* induced by g" and $[\!(g)\!]$ as "the T-*anamorphism* induced by g". This terminology is derived from the Greek words $\kappa\alpha\tau\alpha$ (cata) and $\alpha\nu\alpha$ (ana) meaning, respectively, "downwards" and "upwards" (compare with the direction of the f arrow in each diagram). The exchange of parentheses "( )" and "[ ]" in double parentheses "$(\!|\ \ |\!)$" and "$[\!(\ )\!]$" is aimed at expressing the duality of both concepts.

We shall have a lot to say about catamorphisms and anamorphisms of a given type such as T. For the moment, it suffices to say that

- the T-catamorphism induced by $B \xleftarrow{g} 1 + \mathbb{N}_0 \times B$ is the unique function $B \xleftarrow{(\!|\, g \,|\!)} \mathsf{T}$ which obeys to property (or is defined by)

$$(\!|\, g \,|\!) \;=\; g \cdot (id + id \times (\!|\, g \,|\!)) \cdot out_{\mathsf{T}} \tag{3.42}$$

[7] The theory which supports the statements of this paragraph will not be dealt with until chapter 8 .

which is the same as

$$\llparenthesis g \rrparenthesis \cdot in_{\mathsf{T}} \quad = \quad g \cdot (id + id \times \llparenthesis g \rrparenthesis) \tag{3.43}$$

- given $B \xrightarrow{g} 1 + \mathbb{N}_0 \times B$ the T-anamorphism induced by g is the unique function $B \xrightarrow{[\!(g)\!]} \mathsf{T}$ which obeys to property (or is defined by)

$$[\!(g)\!] \quad = \quad in_{\mathsf{T}} \cdot (id + id \times [\!(g)\!]) \cdot g \tag{3.44}$$

From (3.41) it can be observed that T can act as a mediator between any T-anamorphism and any T-catamorphism, that is to say, $B \xleftarrow{\llparenthesis g \rrparenthesis} \mathsf{T}$ composes with $\mathsf{T} \xleftarrow{[\!(h)\!]} C$, for some $C \xrightarrow{h} 1 + \mathbb{N}_0 \times C$. In other words, a T-catamorphism call always observe (consume) the output of a T-anamorphism. The latter produces a list of $\mathbb{N}_0$s which is consumed by the former. This is depicted in the diagram which follows:

$$\begin{tikzcd}
B & 1 + \mathbb{N}_0 \times B \arrow[l, "g"'] \\
\mathsf{T} \arrow[u, "\llparenthesis g \rrparenthesis"] & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[l, "in_{\mathsf{T}}"'] \arrow[u, "id + id \times \llparenthesis g \rrparenthesis"'] \\
C \arrow[u, "{[\!(h)\!]}"] \arrow[r, "h"'] & 1 + \mathbb{N}_0 \times C \arrow[u, "{id + id \times [\!(h)\!]}"']
\end{tikzcd} \tag{3.45}$$

What can we say about the $\llparenthesis g \rrparenthesis \cdot [\!(h)\!]$ composition? It is a function from C to B which resorts to T as an *intermediate* data-structure and can be subject to the following calculation (*cf.* outermost rectangle in (3.45)):

$$\begin{array}{cl}
& \llparenthesis g \rrparenthesis \cdot [\!(h)\!] = g \cdot (id + id \times \llparenthesis g \rrparenthesis) \cdot (id + id \times [\!(h)\!]) \cdot h \\
\equiv & \quad \{ \text{ +-functor (2.44) } \} \\
& \llparenthesis g \rrparenthesis \cdot [\!(h)\!] = g \cdot ((id \cdot id) + (id \times \llparenthesis g \rrparenthesis) \cdot (id \times [\!(h)\!])) \cdot h \\
\equiv & \quad \{ \text{ identity and } \times\text{-functor (2.30) } \} \\
& \llparenthesis g \rrparenthesis \cdot [\!(h)\!] = g \cdot (id + id \times \llparenthesis g \rrparenthesis \cdot [\!(h)\!]) \cdot h
\end{array}$$

This calculation shows how to define $C \xleftarrow{\llparenthesis g \rrparenthesis \cdot [\!(h)\!]} B$ in one go, that is to say, doing without any intermediate data-structure:

$$\begin{tikzcd}
B & 1 + \mathbb{N}_0 \times B \arrow[l, "g"'] \\
C \arrow[u, "\llparenthesis g \rrparenthesis \cdot {[\!(h)\!]}"] \arrow[r, "h"'] & 1 + \mathbb{N}_0 \times C \arrow[u, "id + id \times \llparenthesis g \rrparenthesis \cdot {[\!(h)\!]}"']
\end{tikzcd} \tag{3.46}$$

As an example, let us see what comes out of $(\!| g |\!) \cdot [\!(h)\!]$ for h and g respectively given by (3.39) and (3.40):

$$\begin{array}{ll}
& (\!| g |\!) \cdot [\!(h)\!] = g \cdot (id + id \times (\!| g |\!) \cdot [\!(h)\!]) \cdot h \\
\equiv & \quad \{ \ (\!| g |\!) \cdot [\!(h)\!] \text{ abbreviated to } f \text{ and instantiating } h \text{ and } g \ \} \\
& f = [\underline{1}, mul] \cdot (id + id \times f) \cdot (! + \langle id, pred \rangle) \cdot (=_0?) \\
\equiv & \quad \{ \ +\text{-functor (2.44) and identity} \ \} \\
& f = [\underline{1}, mul] \cdot (! + (id \times f) \cdot \langle id, pred \rangle) \cdot (=_0?) \\
\equiv & \quad \{ \ \times\text{-absorption (2.27) and identity} \ \} \\
& f = [\underline{1}, mul] \cdot (! + \langle id, f \cdot pred \rangle) \cdot (=_0?) \\
\equiv & \quad \{ \ +\text{-absorption (2.43) and constant } \underline{1} \text{ (2.15)} \ \} \\
& f = [\underline{1}, mul \cdot \langle id, f \cdot pred \rangle] \cdot (=_0?) \\
\equiv & \quad \{ \ \text{McCarthy conditional (2.70)} \ \} \\
& f = (=_0?) \rightarrow \underline{1}, mul \cdot \langle id, f \cdot pred \rangle
\end{array}$$

Going pointwise, we get — via (2.70) —

$$\begin{array}{lcl}
f\,0 & = & [\underline{1}, mul \cdot \langle id, f \cdot pred \rangle](i_1\,0) \\
& = & \quad \{ \ +\text{-cancellation (2.40)} \ \} \\
& & \underline{1}\,0 \\
& = & \quad \{ \ \text{constant function (2.12)} \ \} \\
& & 1
\end{array}$$

and

$$\begin{array}{lcl}
f(n+1) & = & [\underline{1}, mul \cdot \langle id, f \cdot pred \rangle](i_2(n+1)) \\
& = & \quad \{ \ +\text{-cancellation (2.40)} \ \} \\
& & mul \cdot \langle id, f \cdot pred \rangle (n+1) \\
& = & \quad \{ \ \text{pointwise definitions of } split\text{, identity, predecessor and } mul \ \} \\
& & (n+1) \times f\,n
\end{array}$$

In summary, f is but the well-known factorial function:

$$\left\{ \begin{array}{l} f\,0 = 1 \\ f(n+1) = (n+1) \times f\,n \end{array} \right.$$

This result comes to no surprise if we look at diagram (3.45) for the particular g and h we have considered above and recall a popular "definition" of factorial:

$$n! \quad = \quad \underbrace{n \times (n-1) \times \ldots \times 1}_{n \text{ times}} \tag{3.47}$$

In fact, $[\!(h)\!]\,n$ produces T-list

$$Cons(n, Cons(n-1, \ldots Cons(1, Nil)))$$

as an intermediate data-structure which is consumed by $(\!| g |\!)$, the effect of which is but the "replacement" of *Cons* by × and *Nil* by 1, therefore accomplishing (3.47) and realizing the computation of factorial.

HYLOS The moral of this example is that a function as simple as factorial can be *decomposed* into two components (producer/consumer functions) which share a common intermediate inductive datatype. The producer function is an anamorphism which "represents" or produces a "view" of its input argument as a value of the intermediate datatype. The consumer function is a catamorphism which reduces this intermediate data-structure and produces the final result. Like factorial, many functions can be handsomely expressed by a $(\!| g |\!) \cdot [\!(h)\!]$ composition for a suitable choice of the intermediate type, and of g and h.

The intermediate data-structure is said to be *virtual* in the sense that it only exists as a means to induce the associated pattern of recursion and disappears by calculation. The composition

$$(\!| g |\!) \cdot [\!(h)\!]$$

of a T-catamorphism with a T-anamorphism is called a T-*hylomorphism* [8] and is denoted by $[\![g, h]\!]$. Because g and h fully determine the behaviour of the $[\![g, h]\!]$ function, they can be regarded as the "genes" of the function they define. As we shall see, this analogy with biology will prove specially useful for algorithm analysis and classification.

***Exercise** 3.7. A way of computing n^2, the square of a given natural number n, is to sum up the n first odd numbers. In fact,* $1^2 = 1$, $2^2 = 1 + 3$, $3^2 = 1 + 3 + 5$, etc., $n^2 = (2n - 1) + (n - 1)^2$. *Following this hint, express function*

$$sq\, n \stackrel{\mathrm{def}}{=} n^2 \tag{3.48}$$

as a T*-hylomorphism and encode it in* HASKELL.
□

***Exercise** 3.8. Write function x^n as a* T*-hylomorphism and encode it in* HASKELL.
□

***Exercise** 3.9. The following function in* HASKELL *computes the* T*-sequence of all even numbers less or equal to n:*

$$\begin{array}{l} f\ n = \mathbf{if}\ n \leqslant 1\ \mathbf{then}\ Nil\ \mathbf{else}\ Cons\ (m, f\ (m-2)) \\ \quad \mathbf{where}\ m = \mathbf{if}\ even\ n\ \mathbf{then}\ n\ \mathbf{else}\ n-1 \end{array}$$

[8] This terminology is derived from the Greek word $\upsilon\lambda o\sigma$ (hylos) meaning "matter".

Find its "genetic material", that is, function g such that f=$(\!|g|\!)$ *in*

$$\begin{tikzcd}
\mathsf{T} & 1+\mathbb{N}_0\times\mathsf{T} \arrow[l, "in_{\mathsf{T}}"'] \\
\mathbb{N}_0 \arrow[u, "(\!|g|\!)"] \arrow[r, "g"'] & 1+\mathbb{N}_0\times\mathbb{N}_0 \arrow[u, "id+id\times(\!|g|\!)"']
\end{tikzcd}$$

□

3.7 INDUCTIVE TYPES MORE GENERALLY

So far we have focussed our attention exclusively to a particular inductive type T (3.30) — that of finite sequences of non-negative integers. This is, of course, of a very limited scope. First, because one could think of finite sequences of other datatypes, *e.g.* Booleans or many others. Second, because other datatypes such as trees, hash-tables *etc.* exist which our notation and method should be able to take into account.

Although a generic theory of arbitrary datatypes requires a theoretical elaboration which cannot be explained at once, we can move a step further by taking the two observations above as starting points. We shall start from the latter in order to talk generically about inductive types. Then we introduce parameterization and functorial behaviour.

Suppose that, as a mere notational convention, we abbreviate every expression of the form "$1+\mathbb{N}_0\times\ldots$" occurring in the previous section by "$\mathsf{F}\ldots$", *e.g.* $1+\mathbb{N}_0\times B$ by $\mathsf{F}\,B$, *e.g.* $1+\mathbb{N}_0\times\mathsf{T}$ by $\mathsf{F}\,\mathsf{T}$

$$\begin{tikzcd}
\mathsf{T} \arrow[r, bend left, "out_{\mathsf{T}}"] \arrow[r, phantom, "\cong"] & \mathsf{F}\,\mathsf{T} \arrow[l, bend left, "in_{\mathsf{T}}"]
\end{tikzcd} \tag{3.49}$$

etc. This is the same as introducing a datatype-level operator

$$\mathsf{F}\,X \stackrel{\mathrm{def}}{=} 1+\mathbb{N}_0\times X \tag{3.50}$$

which maps every datatype A into datatype $1+\mathbb{N}_0\times A$. Operator F captures the pattern of recursion which is associated to so-called "right" lists (of non-negative integers), that is, lists which grow to the right. The slightly different pattern $\mathsf{G}\,X \stackrel{\mathrm{def}}{=} 1+X\times\mathbb{N}_0$ will generate a different, although related, inductive type

$$X \cong 1+X\times\mathbb{N}_0 \tag{3.51}$$

— that of so-called "left" lists (of non-negative integers). And it is not difficult to think of the pattern which is merges both right and left lists and gives rise to bi-linear lists, better known as *binary trees*:

$$X \cong 1+X\times\mathbb{N}_0\times X \tag{3.52}$$

One may think of many other expressions $\mathsf{F}\,X$ and guess the inductive datatype they generate, for instance $\mathsf{H}\,X \stackrel{\mathrm{def}}{=} \mathbb{N}_0 + \mathbb{N}_0 \times X$ generating non-empty lists of non-negative integers ($\mathbb{N}_0^+$). The general rule is that, given an inductive datatype definition of the form

$$X \;\cong\; \mathsf{F}\,X \tag{3.53}$$

(also called a domain equation), its pattern of recursion is captured by a so-called *functor* F.

3.8 FUNCTORS

The concept of a functor F, borrowed from category theory, is a most generic and useful device in programming [9]. As we have seen, F can be regarded as a datatype constructor which, given datatype A, builds a more elaborate datatype $\mathsf{F}\,A$; given another datatype B, builds a similarly elaborate datatype $\mathsf{F}\,B$; and so on. But what is more important and has the most beneficial consequences is that, if F is regarded as a functor, then its data-structuring effect extends smoothly to functions in the following way: suppose that $B \xleftarrow{f} A$ is a function which observes A into B, which are parameters of $\mathsf{F}\,A$ and $\mathsf{F}\,B$, respectively. By definition, if F is a functor then $\mathsf{F}\,B \xleftarrow{\mathsf{F}\,f} \mathsf{F}\,A$ exists for every such f:

$$\begin{tikzcd}
A \arrow[r, dotted, no head] \arrow[d, "f"'] & \mathsf{F}\,A \arrow[d, "\mathsf{F}\,f"] \\
B \arrow[r, dotted, no head] & \mathsf{F}\,B
\end{tikzcd}$$

$\mathsf{F}\,f$ extends f to F-structures and will, by definition, obey to two very basic properties: it commutes with identity

$$\mathsf{F}\,id_A \;=\; id_{(\mathsf{F}\,A)} \tag{3.54}$$

and with composition

$$\mathsf{F}(g \cdot h) \;=\; (\mathsf{F}\,g) \cdot (\mathsf{F}\,h) \tag{3.55}$$

Two simple examples of a functor follow:

- Identity functor: define $\mathsf{F}\,X = X$, for every datatype X, and $\mathsf{F}\,f = f$. Properties (3.54) and (3.55) hold trivially just by removing symbol F wherever it occurs.

- Constant functors: for a given C, define $\mathsf{F}\,X = C$ (for all datatypes X) and $\mathsf{F}\,f = id_C$, as expressed in the following diagram:

$$\begin{tikzcd}
A \arrow[r, dotted, no head] \arrow[d, "f"'] & C \arrow[d, "id_C"] \\
B \arrow[r, dotted, no head] & C
\end{tikzcd}$$

[9] The category theory practitioner must be warned of the fact that the word *functor* is used here in a too restrictive way. A proper (generic) definition of a functor will be provided later in this book.

Data construction	Universal construct	Functor	Description
$A \times B$	$\langle f, g \rangle$	$f \times g$	Product
$A + B$	$[f, g]$	$f + g$	Coproduct
B^A	$\overline{f}$	f^A	Exponential

Table 2.: Datatype constructions and associated operators.

Properties (3.54) and (3.55) hold trivially again.

In the same way functions can be unary, binary, *etc.*, we can have functors with more than one argument. So we get binary functors (also called *bifunctors*), ternary functors *etc.*. Of course, properties (3.54) and (3.55) have to hold for every parameter of an n-ary functor. For a binary functor B, for instance, equation (3.54) becomes

$$\mathsf{B}\,(id_A, id_B) \;=\; id_{\mathsf{B}\,(A,B)} \tag{3.56}$$

and equation (3.55) becomes

$$\mathsf{B}\,(g \cdot h, i \cdot j) \;=\; \mathsf{B}\,(g, i) \cdot \mathsf{B}\,(h, j) \tag{3.57}$$

Product and coproduct are typical examples of bifunctors. In the former case one has $\mathsf{B}\,(A, B) = A \times B$ and $\mathsf{B}\,(f, g) = f \times g$ — recall (2.24). Properties (2.31) and (2.30) instantiate (3.56) and (3.57), respectively, and this explains why we called them the functorial properties of product. In the latter case, one has $\mathsf{B}\,(A, B) = A + B$ and $\mathsf{B}\,(f, g) = f + g$ — recall (2.39) — and functorial properties (2.45) and (2.44). Finally, exponentiation is a functorial construction too: assuming A, one has $\mathsf{F}\,X \overset{\text{def}}{=} X^A$ and $\mathsf{F}\,f \overset{\text{def}}{=} \overline{f \cdot ap}$ and functorial properties (2.91) and (2.92). All this is summarized in table 2.

Such as functions, functors may compose with each other in the obvious way: the composition of F and G, denoted $\mathsf{F} \cdot \mathsf{G}$, is defined by

$$(\mathsf{F} \cdot \mathsf{G})X \;\overset{\text{def}}{=}\; \mathsf{F}\,(\mathsf{G}\,X) \tag{3.58}$$

$$(\mathsf{F} \cdot \mathsf{G})f \;\overset{\text{def}}{=}\; \mathsf{F}\,(\mathsf{G}\,f) \tag{3.59}$$

3.9 POLYNOMIAL FUNCTORS

We may put constant, product, coproduct and identity functors together to obtain so-called *polynomial functors*, which are described by polynomial expressions, for instance

$$\mathsf{F}\,X = 1 + A \times X$$

— recall (3.17). A polynomial functor is either

- a constant functor or the identity functor, or

- the (finitary) product or coproduct (sum) of other polynomial functors, or
- the composition of other polynomial functors.

So the effect on arrows of a polynomial functor is computed in an easy and structured way, for instance:

$$\begin{array}{rcl}
\mathsf{F}\,f & = & (1 + A \times X)f \\
& = & \qquad \{ \text{ sum of two functors where } A \text{ is a constant and } X \text{ is a variable } \} \\
& & (1)f + (A \times X)f \\
& = & \qquad \{ \text{ constant functor and product of two functors } \} \\
& & id_1 + (A)f \times (X)f \\
& = & \qquad \{ \text{ constant functor and identity functor } \} \\
& & id_1 + id_A \times f \\
& = & \qquad \{ \text{ subscripts dropped for simplicity } \} \\
& & id + id \times f
\end{array}$$

So, $1 + A \times f$ denotes the same as $id_1 + id_A \times f$, or even the same as $id + id \times f$ if one drops the subscripts.

It should be clear at this point that what was referred to in section 2.10 as a "symbolic pattern" applicable to both datatypes and arrows is after all a functor in the mathematical sense. The fact that the same polynomial expression is used to denote both the data and the operators which structurally transform such data is of great conceptual economy and practical application. For instance, once polynomial functor (3.50) is assumed, the diagrams in (3.41) can be written as simply as

$$\begin{tikzcd}
\mathsf{T} \arrow[r, "out_{\mathsf{T}}"] \arrow[d, "f"'] & \mathsf{F}\,\mathsf{T} \arrow[d, "\mathsf{F}\,f"] \\
B & \mathsf{F}\,B \arrow[l, "g"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{T} \arrow[from=d, "f"] & \mathsf{F}\,\mathsf{T} \arrow[l, "in_{\mathsf{T}}"'] \\
B \arrow[r, "g"'] & \mathsf{F}\,B \arrow[u, "\mathsf{F}\,f"']
\end{tikzcd}
\tag{3.60}$$

It is useful to know that, thanks to the isomorphism laws studied in chapter 2, every polynomial functor F may be put into the canonical form,

$$\begin{array}{rcl}
\mathsf{F}\,X & \cong & C_0 + (C_1 \times X) + (C_2 \times X^2) + \cdots + (C_n \times X^n) \\
& = & \sum_{i=0}^{n} C_i \times X^i
\end{array}
\tag{3.61}$$

and that *Newton's binomial formula*

$$(A + B)^n \cong \sum_{p=0}^{n} {}^nC_p \times A^{n-p} \times B^p \tag{3.62}$$

can be used in such conversions. These are performed up to isomorphism, that is to say, after the conversion one gets a different but isomorphic datatype. Consider, for instance, functor

$$\mathsf{F}\,X \stackrel{\mathrm{def}}{=} A \times (1+X)^2$$

(where A is a constant datatype) and check the following reasoning:

$$\begin{array}{cl}
& \mathsf{F}\,X \;=\; A \times (1+X)^2 \\
\cong & \quad \{ \text{ law (2.104) } \} \\
& A \times ((1+X) \times (1+X)) \\
\cong & \quad \{ \text{ law (2.52) } \} \\
& A \times ((1+X) \times 1 + (1+X) \times X)) \\
\cong & \quad \{ \text{ laws (2.57), (2.33) and (2.52) } \} \\
& A \times ((1+X) + (1 \times X + X \times X)) \\
\cong & \quad \{ \text{ laws (2.57) and (2.104) } \} \\
& A \times ((1+X) + (X + X^2)) \\
\cong & \quad \{ \text{ law (2.48) } \} \\
& A \times (1 + (X+X) + X^2) \\
\cong & \quad \{ \text{ canonical form obtained via laws (2.52) and (2.105) } \} \\
& \underbrace{A}_{C_0} + \underbrace{A \times 2}_{C_1} \times X + \underbrace{A}_{C_2} \times X^2
\end{array}$$

***Exercise** 3.10. Synthesize the isomorphism*

$$A + A \times 2 \times X + A \times X^2 \xleftarrow{\nu} A \times (1 + X^2)$$

implicit in the above reasoning.

□

3.10 POLYNOMIAL INDUCTIVE TYPES

An inductive datatype is said to be *polynomial* wherever its pattern of recursion is described by a polynomial functor, that is to say, wherever F in equation (3.53) is polynomial. For instance, datatype T (3.30) is polynomial ($n = 1$) and its associated polynomial functor is canonically defined with coefficients $C_0 = 1$ and $C_1 = \mathbb{N}_0$. For reasons that will become apparent later on, we shall always impose $C_0 \neq 0$ to hold in a *polynomial* datatype expressed in canonical form.

Polynomial types are easy to encode in HASKELL wherever the associated functor is in canonical polynomial form, that is, wherever one has

$$\begin{tikzcd} \mathsf{T} \arrow[r, phantom, "\cong"] & \sum_{i=0}^{n} C_i \times \mathsf{T}^i \arrow[l, bend left, "in_{\mathsf{T}}"] \end{tikzcd} \tag{3.63}$$

Then we have

$$in_{\mathsf{T}} \stackrel{\mathrm{def}}{=} [\, f_1, \ldots, f_n \,]$$

where, for $i = 1, n$, f_i is an arrow of type $\mathsf{T} \leftarrow C_i \times \mathsf{T}^i$. Since n is finite, one may expand exponentials according to (2.104) and encode this in HASKELL as follows:

$$\mathbf{data}\ \mathsf{T} = \mathit{C0} \mid \mathit{C1}\ (\mathit{C1}, \mathsf{T}) \mid \mathit{C2}\ (\mathit{C2}, (\mathsf{T}, \mathsf{T})) \mid \ldots \mid \mathit{Cn}\ (\mathit{Cn}, (\mathsf{T}, \ldots, \mathsf{T}))$$

Of course the choice of symbol `Ci` to realize each f_i is arbitrary [10]. Several instances of polynomial inductive types (in canonical form) will be mentioned in section 3.14. Section 3.19 will address the conversion between inductive datatypes induced by so-called *natural transformations*.

The concepts of catamorphism, anamorphism and hylomorphism introduced in section 3.6 can be extended to arbitrary polynomial types. We devote the following sections to explaining catamorphisms in the polynomial setting. Polynomial anamorphisms and hylomorphisms will not be dealt with until chapter 8.

3.11 F-ALGEBRAS AND F-HOMOMORPHISMS

Our interest in polynomial types is basically due to the fact that, for polynomial F, equation (3.53) always has a particularly interesting solution which corresponds to our notion of a recursive datatype.

In order to explain this, we need two notions which are easy to understand: first, that of an F-*algebra*, which simply is any function α of signature $A \xleftarrow{\alpha} \mathsf{F}\,A$. A is called the *carrier* of F-algebra α and contains the values which α manipulates by computing new A-values out of existing ones, according to the F-pattern (the "type" of the algebra). As examples, consider $[\underline{0}, add]$ (3.29) and in_{T} (3.30), which are both algebras of type $\mathsf{F}\,X = 1 + \mathbb{N}_0 \times X$. The type of an algebra clearly determines its form. For instance, any algebra α of type $\mathsf{F}\,X = 1 + X \times X$ will be of form $[\alpha_1, \alpha_2]$, where α_1 is a constant and α_2 is a binary operator. So monoids are algebras of this type [11].

[10] A more traditional (but less close to (3.63)) encoding will be

$$\mathbf{data}\ \mathsf{T} = \mathit{C0} \mid \mathit{C1}\ \mathit{C1}\ \mathsf{T} \mid \mathit{C2}\ \mathit{C2}\ \mathsf{T}\ \mathsf{T} \mid \ldots \mid \mathit{Cn}\ \mathit{Cn}\ \mathsf{T} \ldots \mathsf{T}$$

delivering every constructor in curried form.

[11] But not every algebra of this type is a monoid, since the type of an algebra only fixes its syntax and does not impose any properties such as associativity, *etc.*

Secondly, we introduce the notion of an F-*homomorphism* which is but a function observing a particular F-algebra α into another F-algebra β:

$$\begin{tikzcd} A \arrow[d, "f"'] & \mathsf{F}\,A \arrow[l, "\alpha"'] \arrow[d, "\mathsf{F}\,f"] \\ B & \mathsf{F}\,B \arrow[l, "\beta"] \end{tikzcd} \qquad f \cdot \alpha = \beta \cdot (\mathsf{F}\,f) \tag{3.64}$$

Clearly, f can be regarded as a structural translation between A and B, that is, A and B have a similar structure [12]. Note that — thanks to (3.54) — identity functions are always (trivial) F-homomorphisms and that — thanks to (3.55) — these homomorphisms compose, that is, the composition of two F-homomorphisms is an F-homomorphism.

3.12 F-CATAMORPHISMS

An F-algebra can be epic, monic or both, that is, iso. Iso F-algebras are particularly relevant to our discussion because they describe solutions to the $X \cong \mathsf{F}\,X$ equation (3.53). Moreover, for polynomial F a particular iso F-algebra always exists, which is denoted by $\mu\mathsf{F} \xleftarrow{in} \mathsf{F}\,\mu\mathsf{F}$ and has special properties. First, its carrier is the smallest among the carriers of other iso F-algebras, and this is why it is denoted by $\mu\mathsf{F}$ — μ for "minimal" [13]. Second, it is the so-called *initial* F-algebra. What does this mean?

It means that, for every F-algebra α there exists one and only one F-homomorphism between *in* and α. This unique arrow mediating *in* and α is therefore determined by α itself, and is called the F-*catamorphism* generated by α. This construct, which was introduced in 3.6, is in general denoted by $(\!|\alpha|\!)_{\mathsf{F}}$:

$$\begin{tikzcd} \mu\mathsf{F} \arrow[d, "f=(\!|\alpha|\!)_{\mathsf{F}}"'] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "in"'] \arrow[d, "\mathsf{F}\,(\!|\alpha|\!)_{\mathsf{F}}"] \\ A & \mathsf{F}\,A \arrow[l, "\alpha"] \end{tikzcd} \tag{3.65}$$

We will drop the F subscript in $(\!|\alpha|\!)_{\mathsf{F}}$ wherever deducible from the context, and often call α the "gene" of $(\!|\alpha|\!)_{\mathsf{F}}$.

As happens with *splits*, *eithers* and transposes, the uniqueness of the catamorphism construct is captured by a universal property established in the class of all F-homomorphisms:

$$k = (\!|\,\alpha\,|\!) \quad\Leftrightarrow\quad k \cdot in = \alpha \cdot \mathsf{F}\,k \tag{3.66}$$

[12] Cf. *homomorphism* = *homo* (the same) + *morphos* (structure, shape).

[13] $\mu\mathsf{F}$ means the least fixpoint solution of equation $X \cong \mathsf{F}\,X$, as will be described in chapter 8 .

According to the experience gathered from section 2.13 onwards, a few properties can be expected as consequences of (3.66). For instance, one may wonder about the "gene" of the identity catamorphism. Just let $k = id$ in (3.66) and see what happens:

$$\begin{array}{cl}
& id = (\!|\, \alpha \,|\!) \Leftrightarrow id \cdot in = \alpha \cdot \mathsf{F}\, id \\
= & \quad \{ \text{ identity (2.10) and F is a functor (3.54) } \} \\
& id = (\!|\, \alpha \,|\!) \Leftrightarrow in = \alpha \cdot id \\
= & \quad \{ \text{ identity (2.10) once again } \} \\
& id = (\!|\, \alpha \,|\!) \Leftrightarrow in = \alpha \\
= & \quad \{ \ \alpha \text{ replaced by } in \text{ and simplifying } \} \\
& id = (\!|\, in \,|\!)
\end{array}$$

Thus one finds out that the genetic material of the identity catamorphism is the initial algebra in. Which is the same as establishing the *reflection property* of catamorphisms:

Cata-reflection :

$$\begin{tikzcd}
\mu\mathsf{F} \arrow[d, "(\!|\, in \,|\!)"'] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "in"'] \arrow[d, "\mathsf{F}\,(\!|\, in \,|\!)"] \\
\mu\mathsf{F} & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "in"]
\end{tikzcd} \qquad (\!|\, in \,|\!) = id_{\mu\mathsf{F}} \qquad (3.67)$$

In a more intuitive way, one might have observed that $(\!|\, in \,|\!)$ is, by definition of in, the unique arrow mediating $\mu\mathsf{F}$ and itself. But another arrow of the same type is already known: the identity $id_{\mu\mathsf{F}}$. So these two arrows must be the same.

Another property following immediately from (3.66), for $k = (\!|\, \alpha \,|\!)$, is

Cata-cancellation :

$$(\!|\, \alpha \,|\!) \cdot in = \alpha \cdot \mathsf{F}\,(\!|\, \alpha \,|\!) \qquad (3.68)$$

Because in is iso, this law can be rephrased as follows

$$(\!|\, \alpha \,|\!) = \alpha \cdot \mathsf{F}\,(\!|\, \alpha \,|\!) \cdot out \qquad (3.69)$$

where out denotes the inverse of in:

$$\begin{tikzcd}
\mu\mathsf{F} \arrow[r, bend left, "out"] \arrow[r, phantom, "\cong"] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, bend left, "in"]
\end{tikzcd}$$

Now, let f be F-homomorphism (3.64) between F-algebras α and β. How does it relate to $(\!|\, \alpha \,|\!)$ and $(\!|\, \beta \,|\!)$? Note that $f \cdot (\!|\, \alpha \,|\!)$ is an arrow mediating $\mu\mathsf{F}$ and B. But B is the carrier of β and $(\!|\, \beta \,|\!)$ is the unique arrow mediating $\mu\mathsf{F}$ and B. So the two arrows are the same:

Cata-fusion :

$$\begin{tikzcd}
\mu\mathsf{F} \arrow[d, "\llparenthesis\, \alpha \,\rrparenthesis"'] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "in"'] \arrow[d, "\mathsf{F}\llparenthesis\, \alpha \,\rrparenthesis"] \\
A \arrow[d, "f"'] & \mathsf{F}\,A \arrow[l, "\alpha"] \arrow[d, "\mathsf{F}\,f"] \\
B & \mathsf{F}\,B \arrow[l, "\beta"]
\end{tikzcd}
\qquad f \cdot \llparenthesis\, \alpha \,\rrparenthesis = \llparenthesis\, \beta \,\rrparenthesis \quad \text{if} \quad f \cdot \alpha = \beta \cdot \mathsf{F}\,f \quad (3.70)$$

Of course, this law is also a consequence of the universal property, for $k = f \cdot \llparenthesis\, \alpha \,\rrparenthesis$:

$$\begin{aligned}
f \cdot \llparenthesis\, \alpha \,\rrparenthesis = \llparenthesis\, \beta \,\rrparenthesis \;\Leftrightarrow\;& (f \cdot \llparenthesis\, \alpha \,\rrparenthesis) \cdot in = \beta \cdot \mathsf{F}\,(f \cdot \llparenthesis\, \alpha \,\rrparenthesis) \\
\Leftrightarrow\;& \qquad \{ \text{ composition is associative and } \mathsf{F} \text{ is a functor (3.55) } \} \\
& f \cdot (\llparenthesis\, \alpha \,\rrparenthesis \cdot in) = \beta \cdot (\mathsf{F}\,f) \cdot (\mathsf{F}\,\llparenthesis\, \alpha \,\rrparenthesis) \\
\Leftrightarrow\;& \qquad \{ \text{ cata-cancellation (3.68) } \} \\
& f \cdot \alpha \cdot \mathsf{F}\,\llparenthesis\, \alpha \,\rrparenthesis = \beta \cdot \mathsf{F}\,f \cdot \mathsf{F}\,\llparenthesis\, \alpha \,\rrparenthesis \\
\Leftarrow\;& \qquad \{ \text{ require } f \text{ to be a } \mathsf{F}\text{-homomorphism (3.64) } \} \\
& f \cdot \alpha \cdot \mathsf{F}\,\llparenthesis\, \alpha \,\rrparenthesis = f \cdot \alpha \cdot \mathsf{F}\,\llparenthesis\, \alpha \,\rrparenthesis \wedge f \cdot \alpha = \beta \cdot \mathsf{F}\,f \\
\Leftrightarrow\;& \qquad \{ \text{ simplify } \} \\
& f \cdot \alpha = \beta \cdot \mathsf{F}\,f
\end{aligned}$$

The presentation of the *absorption* property of catamorphisms entails the very important issue of parameterization and deserves to be treated in a separate section, as follows.

3.13 PARAMETERIZATION AND TYPE FUNCTORS

By analogy with what we have done about *splits* (product), *eithers* (coproduct) and transposes (exponential), we now look forward to identifying F-catamorphisms which exhibit functorial behaviour.

Suppose that one wishes to square all numbers that are members of lists of type T (3.30). It can be checked that

$$\llparenthesis\, [\underline{Nil}, Cons \cdot (sq \times id)] \,\rrparenthesis \qquad (3.71)$$

will do this for us, where $\mathbb{N}_0 \xleftarrow{sq} \mathbb{N}_0$ is given by (3.48). This catamorphism, which converted to pointwise notation is nothing but function h which follows

$$\left\{ \begin{array}{l} h\,Nil = Nil \\ h(Cons(a, l)) = Cons(sq\,a, h\,l) \end{array} \right.$$

maps type T to itself. This is because sq maps $\mathbb{N}_0$ to $\mathbb{N}_0$. Now suppose that, instead of sq, one would like to apply a given function

$B \xleftarrow{f} \mathbb{N}_0$ (for some B other than $\mathbb{N}_0$) to all elements of the argument list. It is easy to see that it suffices to replace f for sq in (3.71). However, the output type no longer is T, but rather type $\mathsf{T}' \cong 1 + B \times \mathsf{T}'$.

Types T and T' are very close to each other. They share the same "shape" (recursive pattern) and only differ with respect to the type of elements — $\mathbb{N}_0$ in T and B in T'. This suggests that these two types can be regarded as instances of a more generic list datatype List

$$\begin{tikzcd} \mathsf{List}\,X \arrow[r, phantom, "\cong"] & 1 + X \times \mathsf{List}\,X \arrow[l, bend left, "in=[\underline{Nil},Cons]"] \end{tikzcd} \tag{3.72}$$

in which the type of elements X is allowed to vary. Thus one has $\mathsf{T} = \mathsf{List}\,\mathbb{N}_0$ and $\mathsf{T}' = \mathsf{List}\,B$.

By inspection, it can be checked that, for every $B \xleftarrow{f} A$,

$$\llparenthesis\, [\underline{Nil}, Cons \cdot (f \times id)] \,\rrparenthesis \tag{3.73}$$

maps $\mathsf{List}\,A$ to $\mathsf{List}\,B$. Moreover, for $f = id$ one has:

$$\begin{aligned} & \llparenthesis\, [\underline{Nil}, Cons \cdot (id \times id)] \,\rrparenthesis \\ = \quad & \{ \text{ by the } \times\text{-functor-id property (2.31) and identity } \} \\ & \llparenthesis\, [\underline{Nil}, Cons] \,\rrparenthesis \\ = \quad & \{ \text{ cata-reflection (3.67) } \} \\ & id \end{aligned}$$

Therefore, by defining

$$\mathsf{List}\,f \overset{\text{def}}{=} \llparenthesis\, [\underline{Nil}, Cons \cdot (f \times id)] \,\rrparenthesis$$

what we have just seen can be written thus:

$$\mathsf{List}\,id_A = id_{\mathsf{List}\,A}$$

This is nothing but law (3.54) for F replaced by List. Moreover, it will not be too difficult to check that

$$\mathsf{List}\,(g \cdot f) = \mathsf{List}\,g \cdot \mathsf{List}\,f$$

also holds — *cf.* (3.55). Altogether, this means that List can be regarded as a functor.

In programming terminology one says that $\mathsf{List}\,X$ (the "lists of Xs datatype") is *parametric* and that, by instantiating parameter X, one gets ground lists such as lists of integers, booleans, *etc.* The illustration above deepens one's understanding of parameterization by identifying the functorial behaviour of the parametric datatype along with its parameter instantiations.

All this can be broadly generalized and leads to what is commonly known by a *type functor*. First of all, it should be clear that the generic format

$$\mathsf{T} \quad \cong \quad \mathsf{F}\,\mathsf{T}$$

adopted so far for the definition of an inductive type is not sufficiently detailed because it does not provide a parametric view of T. For simplicity, let us suppose (for the moment) that only one parameter is identified in T. Then we may factor this out via *type variable* X and write (overloading symbol T)

$$\mathsf{T}\,X \quad \cong \quad \mathsf{B}(X, \mathsf{T}\,X)$$

where B is called the type's *base functor*. Binary functor $\mathsf{B}(X,Y)$ is given this name because it is the basis of the whole inductive type definition. By instantiation of X one obtains F. In the example above, $\mathsf{B}\,(X,Y) = 1 + X \times Y$ and in fact $\mathsf{F}\,Y = \mathsf{B}\,(\mathbb{N}_0, Y) = 1 + \mathbb{N}_0 \times Y$, recall (3.50). Moreover, one has

$$\mathsf{F}\,f \quad = \quad \mathsf{B}\,(id, f) \tag{3.74}$$

and so every F-homomorphism can be written in terms of the base-functor of F, *e.g.*

$$f \cdot \alpha = \beta \cdot \mathsf{B}\,(id, f)$$

instead of (3.64).

$\mathsf{T}\,X$ will be referred to as the *type functor* generated by B:

$$\underbrace{\mathsf{T}X}_{\text{type functor}} \quad \cong \quad \underbrace{\mathsf{B}(X, \mathsf{T}X)}_{\text{base functor}}$$

We proceed to the description of its functorial behaviour — $\mathsf{T}\,f$ — for a given $B \xleftarrow{f} A$. As far as typing rules are concerned, we shall have

$$\frac{B \xleftarrow{f} A}{\mathsf{T}\,B \xleftarrow{\mathsf{T}\,f} \mathsf{T}\,A}$$

So we should be able to express $\mathsf{T}\,f$ as a $\mathsf{B}\,(A, \_)$-catamorphism $(\!|\, g \,|\!)$:

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{T}\,A \arrow[d, "\mathsf{T}\,f = (\!|\, g \,|\!)"'] & & \mathsf{B}\,(A, \mathsf{T}\,A) \arrow[ll, "in_{\mathsf{T}\,A}"'] \arrow[d, "\mathsf{B}\,(id, \mathsf{T}\,f)"] \\
B & \mathsf{T}\,B & & \mathsf{B}\,(A, \mathsf{T}\,B) \arrow[ll, "g"]
\end{tikzcd}$$

As we know that $in_{\mathsf{T}\,B}$ is the standard constructor of values of type $\mathsf{T}\,B$, we may put it into the diagram too:

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{T}\,A \arrow[d, "\mathsf{T}\,f = \llparenthesis g \rrparenthesis"'] & & \mathsf{B}\,(A, \mathsf{T}\,A) \arrow[ll, "in_{\mathsf{T}\,A}"'] \arrow[d, "\mathsf{B}\,(id, \mathsf{T}\,f)"] \\
B & \mathsf{T}\,B & & \mathsf{B}\,(A, \mathsf{T}\,B) \arrow[ll, "g"] \\
 & & \mathsf{B}\,(B, \mathsf{T}\,B) \arrow[ul, "in_{\mathsf{T}\,B}"] \arrow[ur, dotted, dash] &
\end{tikzcd}$$

The catamorphism's gene g will be synthesized by filling the dashed arrow in the diagram with the "obvious" $\mathsf{B}\,(f, id)$, whereby one gets

$$\mathsf{T}\,f \stackrel{\mathrm{def}}{=} \llparenthesis\, in_{\mathsf{T}\,B} \cdot \mathsf{B}\,(f, id) \,\rrparenthesis \tag{3.75}$$

and a final diagram, where $in_{\mathsf{T}\,A}$ is abbreviated by in_A (ibid. $in_{\mathsf{T}\,B}$ by in_B):

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{T}\,A \arrow[d, "\mathsf{T}\,f = \llparenthesis in_B \cdot \mathsf{B}\,(f, id) \rrparenthesis"'] & & \mathsf{B}\,(A, \mathsf{T}\,A) \arrow[ll, "in_A"'] \arrow[d, "\mathsf{B}\,(id, \mathsf{T}\,f)"] \\
B & \mathsf{T}\,B & \mathsf{B}\,(B, \mathsf{T}\,B) \arrow[l, "in_B"] & \mathsf{B}\,(A, \mathsf{T}\,B) \arrow[l, "\mathsf{B}\,(f, id)"]
\end{tikzcd}$$

Next, we proceed to derive the useful law of *cata-absorption*

$$\llparenthesis\, g \,\rrparenthesis \cdot \mathsf{T}\,f = \llparenthesis\, g \cdot \mathsf{B}\,(f, id) \,\rrparenthesis \tag{3.76}$$

as consequence of the laws studied in section 3.12. Our target is to show that, for $k = \llparenthesis\, g \,\rrparenthesis \cdot \mathsf{T}\,f$ in (3.66), one gets $\alpha = g \cdot \mathsf{B}\,(f, id)$:

$$\begin{array}{cl}
 & \llparenthesis\, g \,\rrparenthesis \cdot \mathsf{T}\,f = \llparenthesis\, \alpha \,\rrparenthesis \\
\Leftrightarrow & \quad \{ \text{ type-functor definition (3.75) } \} \\
 & \llparenthesis\, g \,\rrparenthesis \cdot \llparenthesis\, in_B \cdot \mathsf{B}\,(f, id) \,\rrparenthesis = \llparenthesis\, \alpha \,\rrparenthesis \\
\Leftarrow & \quad \{ \text{ cata-fusion (3.70) } \} \\
 & \llparenthesis\, g \,\rrparenthesis \cdot in_B \cdot \mathsf{B}\,(f, id) = \alpha \cdot \mathsf{B}\,(id, \llparenthesis\, g \,\rrparenthesis) \\
\Leftrightarrow & \quad \{ \text{ cata-cancellation (3.68) } \} \\
 & g \cdot \mathsf{B}\,(id, \llparenthesis\, g \,\rrparenthesis) \cdot \mathsf{B}\,(f, id) = \alpha \cdot \mathsf{B}\,(id, \llparenthesis\, g \,\rrparenthesis) \\
\Leftrightarrow & \quad \{ \text{ B is a bi-functor (3.57) } \} \\
 & g \cdot \mathsf{B}\,(id \cdot f, \llparenthesis\, g \,\rrparenthesis \cdot id) = \alpha \cdot \mathsf{B}\,(id, \llparenthesis\, g \,\rrparenthesis) \\
\Leftrightarrow & \quad \{ \; id \text{ is natural (2.11) } \} \\
 & g \cdot \mathsf{B}\,(f \cdot id, id \cdot \llparenthesis\, g \,\rrparenthesis) = \alpha \cdot \mathsf{B}\,(id, \llparenthesis\, g \,\rrparenthesis) \\
\Leftrightarrow & \quad \{ \text{ (3.57) again, this time from left to right } \} \\
 & g \cdot \mathsf{B}\,(f, id) \cdot \mathsf{B}\,(id, \llparenthesis\, g \,\rrparenthesis) = \alpha \cdot \mathsf{B}\,(id, \llparenthesis\, g \,\rrparenthesis) \\
\Leftarrow & \quad \{ \text{ Leibniz } \} \\
 & g \cdot \mathsf{B}\,(f, id) = \alpha
\end{array}$$

The following diagram pictures this property of catamorphisms:

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{T}\,A \arrow[d, "\mathsf{T}\,f"'] & & \mathsf{B}\,(A, \mathsf{T}\,A) \arrow[ll, "in_A"'] \arrow[d, "\mathsf{B}\,(id, \mathsf{T}\,f)"] \\
C & \mathsf{T}\,C \arrow[d, "(\!| g |\!)"'] & \mathsf{B}\,(C, \mathsf{T}\,C) \arrow[l, "in_C"] \arrow[d, "\mathsf{B}\,(id, (\!| g |\!))"] & \mathsf{B}\,(A, \mathsf{T}\,C) \arrow[l, "\mathsf{B}\,(f, id)"] \arrow[d, "\mathsf{B}\,(id, (\!| g |\!))"] \\
 & D & \mathsf{B}\,(C, D) \arrow[l, "g"] & \mathsf{B}\,(A, D) \arrow[l, "\mathsf{B}\,(f, id)"]
\end{tikzcd}$$

It remains to show that (3.75) indeed defines a functor. This can be verified by checking properties (3.54) and (3.55) for $\mathsf{F} = \mathsf{T}$:

- Property **type-functor-id**, *cf.* (3.54):

$$\begin{array}{cl}
& \mathsf{T}\,id \\
= & \quad \{ \text{ by definition (3.75) } \} \\
& (\!|\, in_B \cdot \mathsf{B}\,(id, id) \,|\!) \\
= & \quad \{ \text{ B is a bi-functor (3.56) } \} \\
& (\!|\, in_B \cdot id \,|\!) \\
= & \quad \{ \text{ identity and cata-reflection (3.67) } \} \\
& id
\end{array}$$

- Property **type-functor**, *cf.* (3.55) :

$$\begin{array}{cl}
& \mathsf{T}\,(f \cdot g) \\
= & \quad \{ \text{ by definition (3.75) } \} \\
& (\!|\, in_B \cdot \mathsf{B}\,(f \cdot g, id) \,|\!) \\
= & \quad \{ \ id \cdot id = id \text{ and B is a bi-functor (3.57) } \} \\
& (\!|\, in_B \cdot \mathsf{B}\,(f, id) \cdot \mathsf{B}\,(g, id) \,|\!) \\
= & \quad \{ \text{ cata-absorption (3.76) } \} \\
& (\!|\, in_B \cdot \mathsf{B}\,(f, id) \,|\!) \cdot \mathsf{T}\,g \\
= & \quad \{ \text{ by definition (3.75) } \} \\
& \mathsf{T}\,f \cdot \mathsf{T}\,g
\end{array}$$

***Exercise** 3.11. Function*

$$\mathsf{length} = (\!|[\mathsf{zero}, \mathsf{succ} \cdot \pi_2]|\!)$$

counts the number of elements of a finite list. In case the input list has one element at least it suffices to count the elements of its tail starting with count 1 *instead of* 0*:*

$$\mathsf{length} \cdot (a{:}) = (\!|[\mathsf{one}, \mathsf{succ} \cdot \pi_2]|\!) \tag{3.77}$$

Prove (3.77) knowing that

$$\mathsf{length} \cdot (a{:}) = \mathsf{succ} \cdot \mathsf{length}$$

follows from the definition of length. *(**NB:** assume* zero $\_ = 0$ *and* one $\_ = 1$.*)*

□

***Exercise** 3.12. Function* concat, *extracted from Haskell's* Prelude, *can be defined as list catamorphism,*

$$\mathsf{concat} = (\!|[\mathsf{nil}, \mathsf{conc}]|\!) \tag{3.78}$$

where conc $(x, y) = x +\!\!+ y$, nil $\_ = [\,]$, $\mathsf{B}\ (f, g) = id + f \times g$, $\mathsf{F} f = \mathsf{B}\ (id, f)$, *and* $\mathsf{T} f = \mathsf{map}\, f$. *Prove property*

$$\mathsf{length} \cdot \mathsf{concat} = \mathsf{sum} \cdot \mathsf{map\ length} \tag{3.79}$$

resorting to cata-fusion and cata-absorption.

□

3.14 A CATALOGUE OF STANDARD POLYNOMIAL INDUCTIVE TYPES

The following table contains a collection of standard polynomial inductive types and associated base type bi-functors, which are in canonical form (3.63). The table contains two extra columns which may be used as bookmarks for equations (3.74) and (3.75), respectively [14]:

Description	$\mathsf{T}\,X$	$\mathsf{B}\,(X, Y)$	$\mathsf{B}\,(id, f)$	$\mathsf{B}\,(f, id)$
"Right" Lists	$\mathsf{List}\,X$	$1 + X \times Y$	$id + id \times f$	$id + f \times id$
"Left" Lists	$\mathsf{LList}\,X$	$1 + Y \times X$	$id + f \times id$	$id + id \times f$
Non-empty Lists	$\mathsf{NList}\,X$	$X + X \times Y$	$id + id \times f$	$f + f \times id$
Binary Trees	$\mathsf{BTree}\,X$	$1 + X \times Y^2$	$id + id \times f^2$	$id + f \times id$
"Leaf" Trees	$\mathsf{LTree}\,X$	$X + Y^2$	$id + f^2$	$f + id$

(3.80)

All type functors T in this table are unary. In general, one may think of inductive datatypes which exhibit more than one type parameter. Should n parameters be identified in T, then this will be based on an $n + 1$-ary base functor B, *cf.*

$$\mathsf{T}(X_1, \ldots, X_n) \quad \cong \quad \mathsf{B}(X_1, \ldots, X_n, \mathsf{T}(X_1, \ldots, X_n))$$

So, every $n + 1$-ary polynomial functor $\mathsf{B}(X_1, \ldots, X_n, X_{n+1})$ can be identified as the basis of an inductive n-ary type functor (the convention is to stick to the canonical form and reserve the last variable X_{n+1} for the "recursive call"). While type bi-functors ($n = 2$) are often found in programming, the situation in which $n > 2$ is relatively rare.

[14] Since $(id_A)^2 = id_{(A^2)}$ one writes id^2 for id in this table.

For instance, the combination of leaf-trees with binary-trees in (3.80) leads to the so-called "full tree" type bi-functor

Description	$\mathsf{T}(X_1, X_2)$	$\mathsf{B}(X_1, X_2, Y)$	$\mathsf{B}(id, id, f)$	$\mathsf{B}(f, g, id)$
"Full" Trees	$\mathsf{FTree}(X_1, X_2)$	$X_1 + X_2 \times Y^2$	$id + id \times f^2$	$f + g \times id$

(3.81)

As we shall see later on, these types are widely used in programming. In the actual encoding of these types in HASKELL, exponentials are normally expanded to products according to (2.104), see for instance

$$\mathbf{data}\ \mathsf{BTree}\ a = \mathit{Empty} \mid \mathit{Node}\ (a, (\mathsf{BTree}\ a, \mathsf{BTree}\ a)) \tag{3.82}$$

Moreover, one may choose to curry the type constructors as in, *e.g.*

$$\mathbf{data}\ \mathsf{BTree}\ a = \mathit{Empty} \mid \mathit{Node}\ a\ (\mathsf{BTree}\ a)\ (\mathsf{BTree}\ a)$$

Exercise *3.13. Write as a catamorphisms*

- *the function which counts the number of elements of a non-empty list (type* NList *in (3.80)).*
- *the function which computes the maximum element of a binary-tree of natural numbers.*

□

Exercise *3.14. Let*

$$\begin{aligned}&\mathsf{nil}\ \_ = [\,]\\ &\mathsf{singl}\ a = [a]\end{aligned}$$

be defined. Characterize the function which is defined by $(\!|\,[\mathsf{nil}, h]\,|\!)$ *for each of the following definitions of h:*

$$\begin{aligned} h(x, (y_1, y_2)) &= y_1 +\!\!+ [x] +\!\!+ y_2 && (3.83)\\ h &= +\!\!+ \cdot (singl \times +\!\!+) && (3.84)\\ h &= +\!\!+ \cdot (+\!\!+ \times singl) \cdot \mathsf{swap} && (3.85)\end{aligned}$$

Identify in (3.80) which datatypes are involved as base functors.

□

Exercise *3.15. Write as a catamorphism the function which computes the* frontier *of a tree of type* LTree *(3.80), listed from left to right.*

□

Exercise *3.16. Function*

$$
\begin{aligned}
&\mathsf{mirror}\ (\mathit{Leaf}\ a) = \mathit{Leaf}\ a\\
&\mathsf{mirror}\ (\mathit{Fork}\ (x,y)) = \mathit{Fork}\ (\mathsf{mirror}\ y, \mathsf{mirror}\ x)
\end{aligned}
$$

which mirrors binary trees of type $\mathsf{LTree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathsf{LTree}\ a, \mathsf{LTree}\ a)$ *can be defined both as a catamorphism*

$$\mathsf{mirror} = (\!|\mathsf{in} \cdot (id + \mathsf{swap})|\!) \tag{3.86}$$

and as an anamorphism

$$\mathsf{mirror} = [\!((id + \mathsf{swap}) \cdot \mathsf{out})\!] \tag{3.87}$$

where out *is the converse of*

$$\mathsf{in} = [\mathit{Leaf}, \mathit{Fork}] \tag{3.88}$$

Show that both definitions are effectively the same, that is, complete the etc *steps of the rasoning:*

$$
\begin{aligned}
&\quad \mathsf{mirror} = (\!|\mathsf{in} \cdot (id + \mathsf{swap})|\!)\\
\equiv &\qquad \{\ \ldots\ etc\ \ldots\ \}\\
&\quad \mathsf{mirror} = [\!((id + \mathsf{swap}) \cdot \mathsf{out})\!]
\end{aligned}
$$

□

*(**Hint**: recall that* $\mathsf{F} f = id + f \times f$ *for this type and mind the natural property of* $id + \mathsf{swap}$.*)*

□

***Exercise** 3.17. Let parametric type* T *be given with base* B*, that is, such that* $\mathsf{T}\, f = (\!|\ \mathbf{in} \cdot \mathsf{B}\ (f, id)|\!)$. *Define the so-called* triangular combinator *of* T*,* $tri\ f$*, as follows:*

$$tri\ f = (\!|\ \mathbf{in} \cdot \mathsf{B}\ (id, \mathsf{T}\, f)|\!) \tag{3.89}$$

Show that the instance of this combinator for type $\mathsf{LTree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathsf{LTree}\ a, \mathsf{LTree}\ a)$ *— such that* $\mathsf{in} = [\mathit{Leaf}, \mathit{Fork}]$ *and* $\mathsf{B}\ (f,g) = f + g \times g$ *— is the following function*

$$
\begin{aligned}
&tri :: (a \to a) \to \mathsf{LTree}\ a \to \mathsf{LTree}\ a\\
&tri\ f\ (\mathit{Leaf}\ x) = \mathit{Leaf}\ x\\
&tri\ f\ (\mathit{Fork}\ (t,t')) = \mathit{Fork}\ (\mathit{fmap}\ f\ (tri\ f\ t), \mathit{fmap}\ f\ (tri\ f\ t'))
\end{aligned}
$$

written in Haskell syntax.

□

3.15 HYLO-FACTORIZATION

A well-known example of a hylomorphism is the algorithm than computes the sequence of disk moves in the Towers of Hanoi puzzle:

$$
\begin{aligned}
&hanoi\ (d,0) = [\,]\\
&hanoi\ (d,n+1) = hanoi\ (\neg\ d,n) +\!\!+ [(n,d)] +\!\!+ hanoi\ (\neg\ d,n)
\end{aligned}
\tag{3.90}
$$

Here is a nice account of this puzzle, taken from [6]:

Figure 3.1.: Towers of Hanoi.

> *The Towers of Hanoi problem comes from a puzzle marketed in 1883 by the French mathematician Édouard Lucas, under the pseudonym Claus. The puzzle is based on a legend according to which there is a temple, apparently in Bramah rather than in Hanoi as one might expect, where there are three giant poles fixed in the ground. On the first of these poles, at the time of the world's creation, God placed sixty four golden disks, each of different size, in decreasing order of size. The Bramin monks were given the task of moving the disks, one per day, from one pole to another subject to the rule that no disk may ever be above a smaller disk. The monks' task would be complete when they had succeeded in moving all the disks from the first of the poles to the second and, on the day that they completed their task the world would come to an end!*
>
> *There is a wellknown inductive solution to the problem (...) In this solution we make use of the fact that the given problem is symmetrical with respect to all three poles. Thus it is undesirable to name the individual poles. Instead we visualize the poles as being arranged in a circle [See figure 3.1]; the problem is to move the tower of disks from one pole to the next pole in a specified direction around the circle. The code defines H n d§ to be a sequence of pairs* (k,d') *where n is the number of disks, k is a disk number and d and d′ are directions. Disks are numbered from 0 onwards, disk 0 being the smallest. (Assigning number 0 to the smallest rather than the largest disk has the advantage that the number of the disk that is moved on any day is independent of the total number of disks to be moved.) Directions are boolean values, true representing a clockwise movement and false an anticlockwise movement. The pair* (k,d') *means move the disk numbered k from its current position in the direction d′. (...) Taking the pairs in order from left to right, the complete sequence (...) prescribes how to move the n smallest disks onebyone from one pole to the next pole in the direction d following the rule of never placing a larger disk on top of a smaller disk.*

Next, here is how the same function (3.90) can be viewed as a hylomorphism:[15]

$$
\begin{array}{l}
hanoi = (\!|\, inord\, |\!) \cdot [\!(strategy)\!]\ \mathbf{where} \\
\quad strategy\ (d,0) = i_1\ ()
\end{array}
$$

[15] Recall (3.83) concerning function *inord*.

$$\begin{array}{l}
\mathit{strategy}\ (d,n+1) = i_2\ ((d,n),((\neg\ d,n),(\neg\ d,n)))\\
\mathit{inord} = [\mathsf{nil},f]\\
f\ (x,(l,r)) = l +\!\!+ [x] +\!\!+ r
\end{array}$$

This means that, for some functor F,

$$\mathit{hanoi} = \mathit{inord} \cdot \mathsf{F}\ \mathit{hanoi} \cdot \mathit{strategy} \tag{3.91}$$

holds. The question is: what is the functor F capturing the *recursive pattern* of the algorithm? From $\mathit{strategy}\ (d,0) = i_1\ ()$ we infer the type

$$\mathit{strategy} : \mathbb{B} \times \mathbb{N}_0 \to 1 + ...$$

and from $\mathit{strategy}\ (d,n+1) = i_2\ ((d,n),((\neg\ d,n),(\neg\ d,n)))$ we infer

$$\mathit{strategy} : \mathbb{B} \times \mathbb{N}_0 \to \ldots + (\mathbb{B} \times \mathbb{N}_0) \times (\mathbb{B} \times \mathbb{N}_0)^2$$

Altogether:

$$\begin{tikzcd}
(\mathbb{B} \times \mathbb{N}_0)^* & 1 + (\mathbb{B} \times \mathbb{N}_0) \times ((\mathbb{B} \times \mathbb{N}_0)^*)^2 \arrow[l, "\mathit{inord}"'] \\
\mathbb{B} \times \mathbb{N}_0 \arrow[u, "\mathit{hanoi}"] \arrow[r, "\mathit{strategy}"'] & 1 + (\mathbb{B} \times \mathbb{N}_0) \times (\mathbb{B} \times \mathbb{N}_0)^2 \arrow[u, "id + id \times \mathit{hanoi}^2"']
\end{tikzcd}$$

We conclude that $\mathsf{F}\ X = 1 + (\mathbb{B} \times \mathbb{N}_0) \times X^2$:

$$\begin{tikzcd}
(\mathbb{B} \times \mathbb{N}_0)^* & \mathsf{F}\ (\mathbb{B} \times \mathbb{N}_0)^* \arrow[l, "\mathit{inord}"'] \\
\mathbb{B} \times \mathbb{N}_0 \arrow[u, "\mathit{hanoi}"] \arrow[r, "\mathit{strategy}"'] & \mathsf{F}\ (\mathbb{B} \times \mathbb{N}_0) \arrow[u, "\mathsf{F}\ \mathit{hanoi}"']
\end{tikzcd}$$

Since $\mathsf{F}\ X = \mathsf{B}\ (Y,X)$ for some B, we get

$$\mathsf{F}\ X = \mathsf{B}\ (\mathbb{B} \times \mathbb{N}_0, X)$$

for $\mathsf{B}\ (Y,X) = 1 + Y \times X^2$. Finally, from the table in (3.80) we conclude that the intermediate (virtual) structure of the *hanoi* hylomorphism is $\mathsf{BTree}\ (\mathbb{B} \times \mathbb{N}_0)$:

$$\begin{tikzcd}
(\mathbb{B} \times \mathbb{N}_0)^* & & \mathsf{F}\ (\mathbb{B} \times \mathbb{N}_0)^* \arrow[ll, "\mathit{inord}"'] \\
\mathsf{BTree}\ (\mathbb{B} \times \mathbb{N}_0) \arrow[u, "(\!|\mathit{inord}|\!)"] \arrow[rr, bend left, "\mathsf{out}"] & \cong & \mathsf{F}\ (\mathsf{BTree}\ (\mathbb{B} \times \mathbb{N}_0)) \arrow[u, "\mathsf{F}\ (\!|\mathit{inord}|\!)"'] \arrow[ll, bend left, "\mathsf{in}"] \\
\mathbb{B} \times \mathbb{N}_0 \arrow[u, "{[\!(\mathit{strategy})\!]}"] \arrow[rr, "\mathit{strategy}"'] \arrow[uu, bend left=60, "\mathit{hanoi}"] & & \mathsf{F}\ (\mathbb{B} \times \mathbb{N}_0) \arrow[u, "\mathsf{F}\ {[\!(\mathit{strategy})\!]}"'] \arrow[uu, bend right=60, "\mathsf{F}\ \mathit{hanoi}"']
\end{tikzcd}$$

***Exercise** 3.18. Show that (3.91) unfolds to (3.90) for* $\mathsf{F}\ X = 1 + (\mathbb{B} \times \mathbb{N}_0) \times X^2$.

□

Exercise *3.19. From the hanoi function (3.90) one can derive the function that gives the total number of disk movements of the puzzle:*

$$nm\ 0 = 0$$
$$nm\ (n+1) = 2 * (nm\ n) + 1$$

That is:

$$nm = \mathsf{for}\ odd\ 0\ \textbf{where}\ odd\ n = 2 * n + 1 \tag{3.92}$$

Show that

$$nm\ n = 2^n - 1.$$

***Hint**: define $k\ n = 2^n - 1$ and solve the equation $k = \mathsf{for}\ odd\ 0$ using the laws of catamorphisms and basic properties of arithmetics.*

□

Exercise *3.20. From the* pointwise *version on 'quicksort'',*

$$qSort\ [\,] = [\,]$$
$$qSort\ (h:t) = qSort\ [a \mid a \leftarrow t, a < h] +\!\!+ [h] +\!\!+ qSort\ [a \mid a \leftarrow t, a \geqslant h]$$

infer g and h in the hylo-factorization $qSort = (\!|g|\!) \cdot [\!(h)\!]$, knowing that the intermediate structure is a BTree *as in the case of hanoi.*

□

Exercise *3.21. Consider the well-known function which computes the n-th Fibonacci number:*

$$fib\ 0 = 1$$
$$fib\ 1 = 1$$
$$fib\ (n+2) = fib\ (n+1) + fib\ n$$

Show that fib is a hylomorphism of type LTree *(3.80),*

$$fib = [\![count, fibd]\!]$$

for

$$count = [\underline{1}, \mathsf{add}]$$
$$\mathsf{add}\ (x,y) = x + y$$
$$fibd\ 0 = i_1\ ()$$
$$fibd\ 1 = i_1\ ()$$
$$fibd\ (n+2) = i_2\ (n+1, n)$$

□

***Exercise** 3.22. Consider the following definition of the factorial function,*

$$\begin{array}{l} \mathit{dfac}\ 0 = 1 \\ \mathit{dfac}\ n = [\![[\mathit{id}, \mathsf{mul}], \mathit{dfacd}]\!]\ (1, n) \end{array}$$

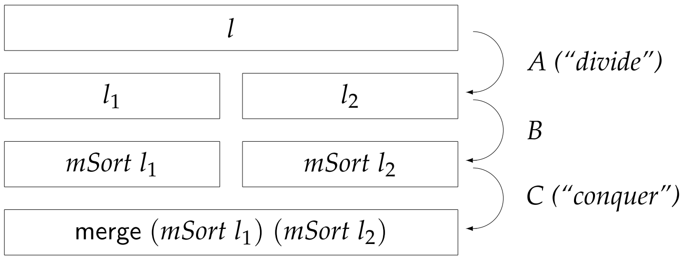
where

$$\begin{array}{l} \mathsf{mul}\ (x, y) = x * y \\ \mathit{dfacd}\ (n, m) \\ \quad \mid n \equiv m = i_1\ n \\ \quad \mid \mathsf{otherwise} = i_2\ ((n, k), (k + 1, m))\ \mathbf{where}\ k = (n + m) \div 2 \end{array}$$

Derive from the above the corresponding (doubly recursive) pointwise definition of dfac. (This is known as the double factorial *implementation of factorial.)*

□

***Exercise** 3.23. The drawing below describes how the so-called* merge sort *algorithm works*[16]*:*



Define the function merge *and then the hylomorphism*

$$\mathit{mSort} = (\!| g |\!) \cdot [\!([\mathsf{singl}, \mathsf{merge}])\!]$$

(find g) knowing that its virtual data-structure is of type LTree*. Note: the empty list should be left out of the hylomorphism and handled separately.*

□

3.16 FUNCTORS AND TYPE FUNCTORS IN HASKELL

The concept of a (unary) functor is provided in HASKELL in the form of a particular class exporting the `fmap` operator:

$$\begin{array}{l} \mathbf{class}\ \mathit{Functor}\ f\ \mathbf{where} \\ \quad \mathit{fmap} :: (a \to b) \to (f\ a \to f\ b) \end{array}$$

[16] Only the case of inputs with more than one element is depicted.

So `fmap g` encodes $\mathsf{F}\,g$ once we declare F as an instance of class `Functor`. The most popular use of `fmap` has to do with HASKELL lists, as allowed by declaration

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where}\\ \quad \mathit{fmap}\,f\ [\,] = [\,]\\ \quad \mathit{fmap}\,f\ (x:xs) = f\ x : \mathit{fmap}\,f\ xs\end{array}$$

in language's *Standard Prelude*.

In order to encode the type functors we have seen so far we have to do the same concerning their declaration. For instance, should we write

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor}\ \mathsf{BTree}\\ \quad \textbf{where}\ \mathit{fmap}\,f = \mathit{cataBTree}\ (\mathit{inBTree}\cdot(\mathit{id}+(f\times \mathit{id})))\end{array}$$

concerning the binary-tree datatype of (3.80) and assuming appropriate declarations of `cataBTree` and `inBTree`, then `fmap` is overloaded and used across such binary-trees.

Bi-functors can be added easily by writing

$$\begin{array}{l}\textbf{class}\ \mathit{BiFunctor}\,f\ \textbf{where}\\ \quad \mathit{bmap} :: (a\rightarrow b)\rightarrow(c\rightarrow d)\rightarrow(f\ a\ c\rightarrow f\ b\ d)\end{array}$$

***Exercise** 3.24. Declare all datatypes in (3.80) in* HASKELL *notation and turn them into* HASKELL *type functors, that is, define* `fmap` *in each case.*

□

***Exercise** 3.25. Declare datatype (3.81) in* HASKELL *notation and turn it into an instance of class* `BiFunctor`.

□

3.17 THE MUTUAL-RECURSION LAW

The theory developed so far for building (and reasoning about) recursive functions doesn't cope with mutual recursion. As a matter of fact, the pattern of recursion of a given cata(ana,hylo)morphism involves only the recursive function being defined, even though more than once, in general, as dictated by the relevant base functor.

It turns out that rules for handling mutual recursion are surprisingly simple to calculate. As motivation, recall section 2.10 where, by mixing products with coproducts, we obtained a result — the *exchange rule* (2.49) — which stemmed from putting together the two universal properties of product and coproduct, (2.63) and (2.65), respectively.

The question we want to address in this section is of the same brand: *what can one tell about catamorphisms which output pairs of values*? By (2.63), such catamorphisms are bound to be *splits*, as are the corresponding *genes*:[17]

$$\begin{tikzcd}
\mathsf{T} \arrow[d, "\llparenthesis \langle h,k \rangle \rrparenthesis"'] & \mathsf{F}\,\mathsf{T} \arrow[l, "in"'] \arrow[d, "\mathsf{F}\,\llparenthesis \langle h,k \rangle \rrparenthesis"] \\
A \times B & \mathsf{F}\,(A \times B) \arrow[l, "\langle h,k \rangle"]
\end{tikzcd}$$

As we did for the exchange rule, we put (2.63) and the universal property of catamorphisms (3.66) against each other and calculate:

$$\begin{aligned}
& \langle f,g \rangle = \llparenthesis \langle h,k \rangle \rrparenthesis \\
\equiv \quad & \quad \{ \text{ cata-universal (3.66) } \} \\
& \langle f,g \rangle \cdot \mathsf{in} = \langle h,k \rangle \cdot \mathsf{F}\,\langle f,g \rangle \\
\equiv \quad & \quad \{ \ \times\text{-fusion (2.26) twice } \} \\
& \langle f \cdot \mathsf{in}, g \cdot \mathsf{in} \rangle = \langle h \cdot \mathsf{F}\,\langle f,g \rangle, k \cdot \mathsf{F}\,\langle f,g \rangle \rangle \\
\equiv \quad & \quad \{ \text{ (2.64) } \} \\
& \left\{ \begin{array}{l} f \cdot \mathsf{in} = h \cdot \mathsf{F}\,\langle f,g \rangle \\ g \cdot \mathsf{in} = k \cdot \mathsf{F}\,\langle f,g \rangle \end{array} \right.
\end{aligned}$$

The rule thus obtained,

$$\left\{ \begin{array}{l} f \cdot \mathsf{in} = h \cdot \mathsf{F}\,\langle f,g \rangle \\ g \cdot \mathsf{in} = k \cdot \mathsf{F}\,\langle f,g \rangle \end{array} \right. \quad \equiv \quad \langle f,g \rangle = \llparenthesis \langle h,k \rangle \rrparenthesis \tag{3.93}$$

is referred to as the *mutual recursion law* (or as "Fokkinga's law") and is useful in combining two mutually recursive functions f and g

$$\begin{tikzcd}
\mathsf{T} \arrow[d, "f"'] & \mathsf{F}\,\mathsf{T} \arrow[l, "in"'] \arrow[d, "\mathsf{F}\,\langle f,g \rangle"] & \mathsf{T} \arrow[d, "g"'] & \mathsf{F}\,\mathsf{T} \arrow[l, "in"'] \arrow[d, "\mathsf{F}\,\langle f,g \rangle"] \\
A & \mathsf{F}\,(A \times B) \arrow[l, "h"] & B & \mathsf{F}\,(A \times B) \arrow[l, "k"]
\end{tikzcd}$$

into a single catamorphism.

When applied from left to right, law (3.93) is surprisingly useful in optimizing recursive functions in a way which saves redundant traversals of the input inductive type T. Let us take the Fibonacci function as example:

$$\begin{aligned}
fib\,0 &= 1 \\
fib\,1 &= 1 \\
fib(n+2) &= fib(n+1) + fib\,n
\end{aligned}$$

It can be shown — recall exercise 3.21 — that fib is a hylomorphism of type LTree (3.80). This hylo-factorization of fib tells something

[17] Using T to denote μ_{F}.

about its internal algorithmic structure: the *divide step* $[\![fibd]\!]$ builds a tree whose number of leaves is a Fibonacci number; the *conquer step* $(\!| count |\!)$ just counts such leaves.

There is, of course, much re-calculation in this hylomorphism. Can we improve its performance? The clue is to regard the two instances of fib in the recursive branch as mutually recursive over the natural numbers. This clue is suggested not only by fib having two base cases (so, perhaps it hides two functions) but also by the lookahead $n+2$ in the recursive clause.

We start by defining a function which reduces such a lookahead by 1,

$$f\ n = fib(n+1)$$

Clearly, $f(n+1) = fib(n+2) = f\ n + fib\ n$ and $f\ 0 = fib\ 1 = 1$. Putting f and fib together,

$$\begin{array}{rcl} f\,0 &=& 1 \\ f(n+1) &=& f\ n + fib\ n \\ fib\,0 &=& 1 \\ fib(n+1) &=& f\ n \end{array}$$

we obtain two mutually recursive functions over the natural numbers ($\mathbb{N}_0$) which transform into pointfree equalities

$$\begin{array}{rcl} f \cdot [\underline{0}, suc] &=& [\underline{1}, add \cdot \langle f, fib \rangle] \\ fib \cdot [\underline{0}, suc] &=& [\underline{1}, f] \end{array}$$

over

$$\begin{tikzcd} \mathbb{N}_0 \arrow[r, bend left] \arrow[r, phantom, "\cong"] & \underbrace{1+\mathbb{N}_0}_{\mathsf{F}\,\mathbb{N}_0} \arrow[l, bend left, "\mathbf{in}=[\underline{0},suc]"] \end{tikzcd} \qquad (3.94)$$

Reverse $+$-absorption (2.43) will further enable us to rewrite the above into

$$\begin{array}{rcl} f \cdot \mathsf{in} &=& [\underline{1}, add] \cdot \mathsf{F}\,\langle f, fib \rangle \\ fib \cdot \mathsf{in} &=& [\underline{1}, \pi_1] \cdot \mathsf{F}\,\langle f, fib \rangle \end{array}$$

thus bringing functor $\mathsf{F} f = id + f$ explicit and preparing for mutual recursion removal:

$$\begin{array}{cl} & \begin{array}{rcl} f \cdot \mathsf{in} &=& [\underline{1}, add] \cdot \mathsf{F}\,\langle f, fib \rangle \\ fib \cdot \mathsf{in} &=& [\underline{1}, \pi_1] \cdot \mathsf{F}\,\langle f, fib \rangle \end{array} \\ \equiv & \quad \{\ (3.93)\ \} \\ & \langle f, fib \rangle = (\!| \langle [\underline{1}, add], [\underline{1}, \pi_1] \rangle |\!) \\ \equiv & \quad \{\ \text{exchange law (2.49)}\ \} \\ & \langle f, fib \rangle = (\!| [\langle \underline{1}, \underline{1} \rangle, \langle add, \pi_1 \rangle] |\!) \end{array}$$

$$\equiv \qquad \{ \text{ going pointwise and denoting } \langle f, fib \rangle \text{ by } fib' \ \}$$

$$\begin{cases} fib'\,0 = (1,1) \\ fib'\,(n+1) = (x+y,x)\ \textit{where}\ (x,y) = fib'\,n \end{cases}$$

Since $fib = \pi_2 \cdot fib'$ we easily recover fib from fib' and obtain the intended linear version of Fibonacci, below encoded in Haskell:

```
fib n = m where
  (_, m) = fib' n
  fib' 0 = (1,1)
  fib' (n+1) = (x+y,x) where (x,y) = fib' n
```

This version of fib is actually the semantics of the "for-loop" — recall (3.7) — one would write in an imperative language which would initialize two global variables $x, y := 1, 1$, loop over assignment $x, y := x + y, x$ and yield the result in y. In the C programming language, one would write

```
int fib(int n)
{
int x=1; int y=1; int i;
for (i=1;i<=n;i++) {int a=x; x=x+y; y=a;}
return y;
};
```

where the extra variable a is required for ensuring that *simultaneous* assignment $x, y := x + y, x$ takes place in a sequential way.

Recall from section 3.1 that all $\mathbb{N}_0$ catamorphisms are of shape $(\!|\, [\underline{k}, g] \,|\!)$ and such that $(\!|\, [\underline{k}, g] \,|\!) n = g^n k$, where g^n is the n-th iteration of g, that is, $g^0 = id$ and $g^{n+1} = g \cdot g^n$. That is, g is the body of a "for-loop" which repeats itself n-times, starting with initial value k. Recall also that the for-loop combinator is nothing but the "fold combinator" (3.5) associated to the natural number data type.

In a sense, the mutual recursion law gives us a hint on how global variables "are born" in computer programs, out of the maths definitions themselves. Quite often more that two such variables are required in linearizing hylomorphisms by mutual recursion. Let us see an example. The question is: *how many squares can one draw on a $n \times n$-tiled wall?* The answer is given by function

$$ns\ n \stackrel{\text{def}}{=} \sum_{i=1,n} i^2$$

that is,

$$\begin{array}{rcl} ns\ 0 & = & 0 \\ ns\ (n+1) & = & (n+1)^2 + ns\ n \end{array}$$

in Haskell. However, this hylomorphism is inefficient because each iteration involves another hylomorphism computing square numbers.

One way of improving *ns* is to introduce function $bnm\ n \stackrel{\mathrm{def}}{=} (n+1)^2$ and express this over (3.94),

$$\begin{array}{rcl} bnm\,0 & = & 1 \\ bnm(n+1) & = & 2n+3+bnm\ n \end{array}$$

hoping to blend *ns* with *bnm* using the mutual recursion law. However, the same problem arises in *bnm* itself, which now depends on term $2n+3$. We invent $lin\ n \stackrel{\mathrm{def}}{=} 2n+3$ and repeat the process, thus obtaining:

$$\begin{array}{rcl} lin\,0 & = & 3 \\ lin(n+1) & = & 2+lin\ n \end{array}$$

By redefining

$$\begin{array}{rcl} bnm'\,0 & = & 1 \\ bnm'(n+1) & = & lin\ n + bnm'\ n \end{array}$$

$$\begin{array}{rcl} ns'\,0 & = & 0 \\ ns'(n+1) & = & bnm'\ n + ns'\ n \end{array}$$

we obtain three functions — ns', bnm' and lin — mutually recursive over the polynomial base $\mathsf{F}\,g = id + g$ of the natural numbers.

Exercise 3.29 below shows how to extend (3.93) to three mutually recursive functions (3.95). (From this it is easy to extend it further to the n-ary case.) It is routine work to show that, by application of (3.95) to the above three functions, one obtains the linear version of *ns* which follows:

$$\begin{array}{l} ns''\ n = a\ \mathbf{where} \\ \quad (a,\_,\_) = aux\ n \\ \quad aux\ 0 = (0,1,3) \\ \quad aux\ (n+1) = \mathbf{let}\ (a,b,c) = aux\ n\ \mathbf{in}\ (a+b, b+c, 2+c) \end{array}$$

In retrospect, note that (in general) not every system of n mutually recursive functions

$$\left\{ \begin{array}{l} f_1 = \phi_1(f_1,\ldots,f_n) \\ \vdots \\ f_n = \phi_n(f_1,\ldots,f_n) \end{array} \right.$$

involving n functions and n functional combinators $\phi_1,\ldots,\phi_n$ can be handled by a suitably extended version of (3.93). This only happens if all f_i have the same "shape", that is, if they share the same base functor F.

***Exercise** 3.26. Use the mutual recursion law (3.93) to show that each of the two functions*

$$\left\{ \begin{array}{l} odd\,0 = False \\ odd(n+1) = even\ n \end{array} \right. \qquad \left\{ \begin{array}{l} even\,0 = True \\ even(n+1) = odd\ n \end{array} \right.$$

checking natural number parity can be expressed as a projection of

$$\mathsf{for}\ \mathsf{swap}\ (\mathsf{False}, \mathsf{True})$$

Encode this for-loop in C syntax.
□

Exercise *3.27. The following Haskell function computes the list of the first n natural numbers in reverse order:*

$$\begin{array}{l} insg\ 0 = [\,] \\ insg\ (n+1) = (n+1) : insg\ n \end{array}$$

1. *Show that insg can also be defined as follows:*

$$\begin{array}{l} insg\ 0 = [\,] \\ insg\ (n+1) = (fsuc\ n) : insg\ n \\ fsuc\ 0 = 1 \\ fsuc\ (n+1) = fsuc\ n + 1 \end{array}$$

2. *Based on the mutual recursion law derive from such a definition the following version of insg encoded as a for-loop:*

$$\begin{array}{l} insg = \pi_2 \cdot insgfor \\ insgfor = \mathsf{for}\ \langle (1+) \cdot \pi_1, \mathsf{cons} \rangle\ \underline{(1, [\,])} \end{array}$$

where $\mathsf{cons}\ (n, m) = n : m$.

□

Exercise *3.28. The number of steps that solve the Hanoi Towers "puzzle", for n discs, is:*

$$k\ n = 2^n - 1$$

— recall exercise 3.19. Using the mutual recursion law, show that another way of computing k is

$$\begin{array}{l} k = \pi_1 \cdot g\ \mathbf{where} \\ \quad g = \mathsf{for}\ loop\ (0, 1) \\ \quad loop\ (k, e) = (k + e, 2 * e) \end{array}$$

knowing that

$$\begin{array}{l} k\ 0 = 0 \\ k\ (n+1) = 2^n + k\ n \end{array}$$

hold (as can be easily shown) and that $2^n = \mathsf{for}\ (2*)\ 1\ n$.
□

***Exercise** 3.29. Show that law (3.93) generalizes to more than two mutually recursive functions, three in this case:*

$$\left\{\begin{array}{l} f \cdot in = h \cdot \mathsf{F}\,\langle f, \langle g, j\rangle\rangle \\ g \cdot in = k \cdot \mathsf{F}\,\langle f, \langle g, j\rangle\rangle \\ j \cdot in = l \cdot \mathsf{F}\,\langle f, \langle g, j\rangle\rangle \end{array}\right. \quad \equiv \quad \langle f, \langle g, j\rangle\rangle = (\!|\, \langle h, \langle k, l\rangle\rangle \,|\!) \tag{3.95}$$

□

***Exercise** 3.30. Prove the law*

$$\left\{\begin{array}{l} f = \mathsf{in} \cdot \mathsf{F}\,[f, g] \cdot h \\ g = \mathsf{in} \cdot \mathsf{F}\,[f, g] \cdot k \end{array}\right. \quad \equiv \quad [f, g] = [\!(\, [h, k] \,)\!] \tag{3.96}$$

which dualizes the mutual recursion law, cf.

$$\begin{tikzcd} \mathsf{T} & \mathsf{F\,T} \arrow[l, "\mathsf{in}"'] \\ A + B \arrow[u, "{[f,g] = [\!([h,k])\!]}"] \arrow[r, "{[h,k]}"'] & \mathsf{F}\,(A + B) \arrow[u, "{\mathsf{F}\,[f,g])}"'] \end{tikzcd} \qquad \begin{tikzcd} \mathsf{T} & \mathsf{F\,T} \arrow[l, "\mathsf{in}"'] \arrow[r, "\mathsf{in}"] & \mathsf{T} \\ A \arrow[u, "f"] \arrow[r, "h"'] & \mathsf{F}\,(A + B) \arrow[u, "{\mathsf{F}\,[f,g]}"'] & B \arrow[u, "g"'] \arrow[l, "k"] \end{tikzcd}$$

□

***Exercise** 3.31. The mutual recursion law generalizes to hylomorphisms sharing the same anamorfism:*

$$\langle f, g\rangle = (\!|\, \langle h, k\rangle \,|\!) \cdot [\!(\, q \,)\!] \quad \equiv \quad \left\{\begin{array}{l} f = h \cdot \mathsf{F}\,\langle f, g\rangle \cdot q \\ g = k \cdot \mathsf{F}\,\langle f, g\rangle \cdot q \end{array}\right. \tag{3.97}$$

Prove (3.97).

□

***Exercise** 3.32. The exponential function $e^x : \mathbb{R} \to \mathbb{R}$ (where "e" denotes Euler's number) can be defined in several ways, one being the calculation of Taylor series:*

$$e^x \quad = \quad \sum_{n=0}^{\infty} \frac{x^n}{n!} \tag{3.98}$$

The following function, in Haskell,

```
exp :: Double → Integer → Double
exp x 0 = 1
exp x (n + 1) = x ↑ (n + 1) / fac (n + 1) + (exp x n)
```

computes an approximation of e^x, where the second parameter tells how many terms to compute. For instance, while exp 1 1 = 2.0, exp 1 10 *yields* 2.7182818011463845.

Function $\mathsf{exp}\ x\ n$ *performs badly for* n *larger and larger: while* $\mathsf{exp}\ 1\ 100$ *runs instantaneously,* $\mathsf{exp}\ 1\ 1000$ *takes around 9 seconds,* $\mathsf{exp}\ 1\ 2000$ *takes circa 33 seconds, and so on.*

Decompose exp *into mutually recursive functions so as to apply (3.95) and obtain the following linear version,*

```
$\mathsf{exp}\ x\ n = \textbf{let}\ (e,b,c) = aux\ x\ n$
  $\textbf{in}\ e\ \textbf{where}$
    $aux\ x\ 0 = (1,2,x)$
    $aux\ x\ (n+1) =$
      $\textbf{let}\ (e,s,h) = aux\ x\ n$
      $\textbf{in}\ (e+h, s+1, (x\ /\ s) * h)$
```

which translates directly to the encoding in C:

```
float exp(float x, int n)
{
  float h=x; float e=1; int s=2; int i;
  for (i=0;i<n+1;i++) {e=e+h;h=(x/s)*h;s++;}
  return e;
};
```

□

Exercise *3.33. Show that, for all* $n \in \mathbb{N}_0$, $n = suc^n 0$. ***Hint:*** *use cata-reflection (3.67).*

□

MUTUAL RECURSION OVER LISTS. As example of application of (3.93) for μ_{F} other than $\mathbb{N}_0$, consider the following recursive predicate which checks whether a (non-empty) list is ordered,

$$\begin{aligned}&ord : A^+ \to 2\\&ord\ [a] = \text{TRUE}\\&ord\ (cons(a,l)) = a \geqslant (listMax\ l) \wedge (ord\ l)\end{aligned}$$

where $\geqslant$ is assumed to be a total order on datatype A and

$$listMax = (\!|\, [id, max]\,|\!) \tag{3.99}$$

computes the greatest element of a given list of As:

$$\begin{tikzcd}A^+ \arrow[d, "listMax"'] & A + A \times A^+ \arrow[l, "{[singl\,,cons]}"'] \arrow[d, "id+id\times listMax"] \\ A & A + A \times A \arrow[l, "{[id\,,max]}"]\end{tikzcd}$$

(In the diagram, $singl\ a = [a]$.)

Predicate *ord* is not a catamorphism because of the presence of $listMax\ l$ in the recursive branch. However, the following diagram depicting *ord*

$$\begin{tikzcd}
A^+ \arrow[d, "ord"'] & A + A \times A^+ \arrow[l, "{[singl\,,cons]}"'] \arrow[d, "id+id\times\langle listMax,ord\rangle"] \\
2 & A + A \times (A \times 2) \arrow[l, "{[\underline{\mathrm{TRUE}}\,,\alpha]}"]
\end{tikzcd}$$

(where $\alpha(a,(m,b)) \stackrel{\mathrm{def}}{=} a \geqslant m \wedge b$) suggests the possibility of using the mutual recursion law. One only has to find a way of letting *listMax* depend also on *ord*, which isn't difficult: for any $A^+ \xrightarrow{g} B$, one has

$$\begin{tikzcd}
A^+ \arrow[d, "listMax"'] & A + A \times A^+ \arrow[l, "{[singl\,,cons]}"'] \arrow[d, "id+id\times\langle listMax,g\rangle"] \\
A & A + A \times (A \times B) \arrow[l, "{[id\,,max\cdot(id\times\pi_1)]}"]
\end{tikzcd}$$

where the extra presence of g is cancelled by projection π_1.

For $B = 2$ and $g = ord$ we are in position to apply Fokkinga's law and obtain:

$$\begin{aligned}
\langle listMax, ord\rangle &= \llparenthesis\, \langle [id, max\cdot(id\times\pi_1)], [\underline{\mathrm{TRUE}}, \alpha]\rangle \,\rrparenthesis \\
&= \qquad \{ \textit{ exchange law } (2.49) \} \\
&\quad\ \llparenthesis\, [\langle id, \underline{\mathrm{TRUE}}\rangle, \langle max\cdot(id\times\pi_1), \alpha\rangle] \,\rrparenthesis
\end{aligned}$$

Of course, $ord = \pi_2 \cdot \langle listMax, ord\rangle$. By denoting the above synthesized catamorphism by *aux*, we end up with the following version of *ord*:

$$ord\ l = \mathbf{let}\ (a,b) = aux\ l\ \mathbf{in}\ b$$

where

$$\begin{aligned}
&aux : A^+ \to A \times 2 \\
&aux\ [a] = (a, \mathsf{True}) \\
&aux\ (\mathsf{cons}\ (a,l)) = (max\ (a,m), a > m \wedge b)\ \mathbf{where}\ (m,b) = aux\ l
\end{aligned}$$

Exercise *3.34. What do the following Haskell functions do?*

$$\begin{aligned}
&f_1\ [\,] = [\,] \\
&f_1\ (h:t) = h:(f_2\ t) \\
&f_2\ [\,] = [\,] \\
&f_2\ (h:t) = f_1\ t
\end{aligned}$$

Write $f = \langle f_1, f_2\rangle$ as a list catamorphism and encode f back into Haskell syntax.

□

3.18 "BANANA-SPLIT": A COROLLARY OF THE MUTUAL-RECURSION LAW

Let $f = (\!|\, i \,|\!)$ and $g = (\!|\, j \,|\!)$ be given. Clearly:

$$\begin{array}{cl}
 & f = (\!|\, i \,|\!) \\
\equiv & \qquad \{ \text{ by cata-universal (3.66) } \} \\
 & f \cdot \mathsf{in} = i \cdot \mathsf{F}\, f \\
\equiv & \qquad \{ \text{ by } \times\text{-cancellation (2.22) } \} \\
 & f \cdot \mathsf{in} = i \cdot \mathsf{F}\, (\pi_1 \cdot \langle f, g \rangle) \\
\equiv & \qquad \{ \text{ F is a functor } \} \\
 & f \cdot \mathsf{in} = i \cdot \mathsf{F}\, \pi_1 \cdot \mathsf{F}\, \langle f, g \rangle
\end{array}$$

Similarly,

$$\begin{array}{cl}
 & g = (\!|\, j \,|\!) \\
\equiv & \qquad \{ \text{ as above for } g = (\!|\, j \,|\!) \ \} \\
 & g \cdot \mathsf{in} = j \cdot \mathsf{F}\, \pi_2 \cdot \mathsf{F}\, \langle f, g \rangle
\end{array}$$

Then:

$$\begin{array}{cl}
 & \left\{ \begin{array}{l} f \cdot \mathsf{in} = i \cdot \mathsf{F}\, \pi_1 \cdot \mathsf{F}\, \langle f, g \rangle \\ g \cdot \mathsf{in} = j \cdot \mathsf{F}\, \pi_2 \cdot \mathsf{F}\, \langle f, g \rangle \end{array} \right. \\
\equiv & \qquad \{ \text{ mutual recursion law (3.93) } \} \\
 & \langle f, g \rangle = (\!|\, \langle i \cdot \mathsf{F}\, \pi_1, j \cdot \mathsf{F}\, \pi_2 \rangle \,|\!) \\
\equiv & \qquad \{ \ f = (\!|\, i \,|\!) \text{ and } g = (\!|\, j \,|\!) \ \} \\
 & \langle (\!|\, i \,|\!), (\!|\, j \,|\!) \rangle = (\!|\, \langle i \cdot \mathsf{F}\, \pi_1, j \cdot \mathsf{F}\, \pi_2 \rangle \,|\!)
\end{array}$$

Altogether, we get

$$\langle (\!|\, i \,|\!), (\!|\, j \,|\!) \rangle = (\!|\, \langle i \cdot \mathsf{F}\, \pi_1, j \cdot \mathsf{F}\, \pi_2 \rangle \,|\!)$$

that is

$$\langle (\!|\, i \,|\!), (\!|\, j \,|\!) \rangle = (\!|\, (i \times j) \cdot \langle \mathsf{F}\, \pi_1, \mathsf{F}\, \pi_2 \rangle \,|\!) \qquad (3.100)$$

by (reverse) ×-absorption (2.27).

This law provides us with a very useful tool for "parallel loop" intercombination: "loops" $(\!|\, i \,|\!)$ and $(\!|\, j \,|\!)$ are fused together into a single "loop" $(\!|\, (i \times j) \cdot \langle \mathsf{F}\, \pi_1, \mathsf{F}\, \pi_2 \rangle \,|\!)$. The need for this kind of calculation arises very often. Consider, for instance, the function which computes the average of a non-empty list of natural numbers,

$$average \stackrel{\mathrm{def}}{=} (/) \cdot \langle sum, length \rangle \qquad (3.101)$$

where *sum* and *length* are the expected $\mathbb{N}^+$ catamorphisms:

$$\begin{array}{l}
sum = (\!|\, [id, +] \,|\!) \\
length = (\!|\, [\underline{1}, \mathsf{succ} \cdot \pi_2] \,|\!)
\end{array}$$

As defined by (3.101), function *average* performs two independent traversals of the argument list before division $(/)$ takes place. Banana-split will fuse such two traversals into a single one (see function *aux* below), thus leading to a function which will run "twice as fast":

$$\begin{array}{l} average\ l = x\ /\ y\ \mathbf{where} \\ \quad (x,y) = aux\ l \\ \quad aux\ [\,] = (0,0) \\ \quad aux\ (a : l) = (a + x, y + 1)\ \mathbf{where}\ (x,y) = aux\ l \end{array} \tag{3.102}$$

***Exercise** 3.35. Calculate (3.102) from (3.101). Which of these two versions of the same function is easier to understand?*

□

***Exercise** 3.36. The following diagram depicts "banana-split" (3.100):*

$$\begin{tikzcd}
 & & & & \mathsf{F}\,\mathsf{T} \arrow[dlll, "\mathsf{in}"'] \arrow[ddl, "\mathsf{F}\,\llparenthesis i \rrparenthesis"'] \arrow[ddd, dotted, "f_6"'] \arrow[ddddr, "\mathsf{F}\,\llparenthesis j \rrparenthesis"] & \\
 & \mathsf{T} \arrow[ddl, "\llparenthesis i \rrparenthesis"'] \arrow[ddd, dotted, "f_7"'] \arrow[ddddr, "\llparenthesis j \rrparenthesis"] & & & & \\
 & & & \mathsf{F}\,A \arrow[dlll, "i"'] & & \\
A & & & & \mathsf{F}\,(A \times B) \arrow[ul, "f_1"'] \arrow[llll, dotted, "f_5"'] \arrow[dlll, dotted, "f_4"'] \arrow[ddll, dotted, "f_3"] \arrow[dr, "f_2"] & \\
 & A \times B \arrow[ul, "\pi_1"] \arrow[dr, "\pi_2"'] & & & & \mathsf{F}\,B \arrow[dlll, "j"] \\
 & & B & & &
\end{tikzcd}$$

Identify all functions f_1 to f_7.

□

***Exercise** 3.37. Show that the standard Haskell function*

$$\mathsf{unzip}\ xs = (\mathsf{map}\ \pi_1\ xs, \mathsf{map}\ \pi_2\ xs)$$

*can be defined as a catamorphism (*fold*) thanks to (3.100). Generalize this calculation to the* generic *unzip function over an inductive (polynomial) type* T*:*

$$unzip_{\mathsf{T}} = \langle \mathsf{T}\pi_1, \mathsf{T}\pi_2 \rangle$$

Suggestion: recall (3.75).

□

3.19 INDUCTIVE DATATYPE ISOMORPHISM

Transforming inductive datatypes into other inductive datatypes is an important topic in programming, enabling the *migration* of "data in one format" to data in "another format". Ensuring properties of data migration operations is an important issue in data processing.

Think of a migration f over a dataset which is such that there are two data items x and y such that $f\ x = f\ y$. Clearly, there is information loss in this migration since the source x or y cannot be recovered from the target $f\ x$. Technically, one would say that, in this case, f is not *injective*.

In the general setting, the source and target datatypes are structured by different base functors. Suppose one is given two inductive datatypes

$$\begin{tikzcd} \mathsf{T} \arrow[r, phantom, "\cong"] & \mathsf{F}\,\mathsf{T} \arrow[l, bend left, "\mathsf{in}_\mathsf{T}"] \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} \mathsf{U} \arrow[r, phantom, "\cong"] & \mathsf{G}\,\mathsf{U} \arrow[l, bend left, "\mathsf{in}_\mathsf{U}"] \end{tikzcd}$$

defined by functors F and G, respectively. Moreover suppose that recursion pattern G can be converted to recursion pattern F via polymorphic map $\mathsf{F}\,X \xleftarrow{\alpha_X} \mathsf{G}\,X$. It can be easily checked that

$$(\!|\mathsf{in}_\mathsf{T} \cdot \alpha_\mathsf{T}|\!)_\mathsf{G} \tag{3.103}$$

is a map that converts U-data into T-data by structural application of transformation α:[18]

$$\begin{tikzcd} \mathsf{U} \arrow[d, "(\!|\mathsf{in}_\mathsf{T}\cdot\alpha_\mathsf{T}|\!)"'] & & \mathsf{G}\,\mathsf{U} \arrow[ll, "\mathsf{in}_\mathsf{U}"'] \arrow[d, "\mathsf{G}(\!|\mathsf{in}_\mathsf{T}\cdot\alpha_\mathsf{T}|\!)"] \\ \mathsf{T} & \mathsf{F}\,\mathsf{T} \arrow[l, "\mathsf{in}_\mathsf{T}"] & \mathsf{G}\,\mathsf{T} \arrow[l, "\alpha_\mathsf{T}"] \end{tikzcd}$$

It is useful to know that some properties of α extend inductively to catamorphism (3.103). For instance, should α be an isomorphism then $(\!|\mathsf{in}_\mathsf{T} \cdot \alpha_\mathsf{T}|\!)_\mathsf{G}$ will be an isomorphism as well, that is, T and U will be isomorphic. Before checking this fact, let us see an example. By inspection of table (3.80), it is easy to check that the base functors of *RList* and *LList* — resp. $1 + X \times Y$ and $1 + Y \times X$ — are isomorphic via $\alpha = id + \mathsf{swap}$. This will be enough to establish *RList* and *LList* as isomorphic inductive datatypes.

It is well known that wherever $\beta \cdot \alpha = id$ then β will be a surjective function and α will be injective.[19] As simple examples of application of this fact recall $\times$-cancellation (2.22) and $+$-cancellation (2.40). By instantiating one of the functions involved to the identity, in each

[18] Note that, for $\mathsf{F}\,X = \mathsf{B}\,(A, X)$, $\mathsf{G}\,X = \mathsf{B}\,(B, X)$ and $\alpha = \mathsf{B}\,(f, id)$, (3.103) instantiates to (3.75). We could exploit full parametricity here by working with the base bifunctors but that would add little to what we intend to show at the cost of some notation convolution. Such an extension will be dealt with in section 8.5.

[19] See section 5.12 later on for the technical details.

cas, one immediately finds left-inverses for the injections i_1, i_2 (respectively: right-inverses for the projections π_1, π_2) meaning that they are injective (respectively: surjective) — thus the choice of terminology.

We show next that such properties (injectivity or surjectivity) of α extend to the catamorphism that performs the corresponding inductive type conversion. Two thumb rules will be derived

- cata of injective α is injective (3.104)
- cata of surjective α is surjective (3.105)

and, as a consequence:

- cata of bijective α is bijective (3.106)

Looking at

$$\begin{tikzcd}
\mathsf{U} \arrow[d, "(\!|\mathsf{in}_\mathsf{T} \cdot \alpha|\!)_\mathsf{G}"'] & \mathsf{F\,U} \arrow[d, "\mathsf{F}\,(\!|\mathsf{in}_\mathsf{T} \cdot \alpha|\!)_\mathsf{G}"] & \mathsf{G\,U} \arrow[l, "\alpha"'] \arrow[ll, bend right, "\mathsf{in}_\mathsf{U}"'] \arrow[d, "\mathsf{G}\,(\!|\mathsf{in}_\mathsf{T} \cdot \alpha|\!)_\mathsf{G}"] \\
\mathsf{T} & \mathsf{F\,T} \arrow[l, "\mathsf{in}_\mathsf{T}"] & \mathsf{G\,T} \arrow[l, "\alpha"]
\end{tikzcd}$$

and knowing that $\beta \cdot \alpha = id$, let us find a left-inverse $f : \mathsf{T} \to \mathsf{U}$ for $(\!|\, \mathsf{in}_\mathsf{T} \cdot \alpha \,|\!)$:

$$\begin{array}{cl}
 & f \cdot (\!|\, \mathsf{in}_\mathsf{T} \cdot \alpha \,|\!) = id \\
\Leftarrow & \quad \{ \text{ cata-reflection and cata-fusion } \} \\
 & f \cdot \mathsf{in}_\mathsf{T} \cdot \alpha = \mathsf{in}_\mathsf{U} \cdot \mathsf{G}\,f \\
\equiv & \quad \{ \text{ head for a catamorphism: } f = (\!|\, \gamma \,|\!) \ \} \\
 & (\!|\, \gamma \,|\!) \cdot \mathsf{in}_\mathsf{T} \cdot \alpha = \mathsf{in}_\mathsf{U} \cdot \mathsf{G}\ (\!|\, \gamma \,|\!) \\
\equiv & \quad \{ \text{ cata-cancellation } \} \\
 & \gamma \cdot \mathsf{F}\ (\!|\, \gamma \,|\!) \cdot \alpha = \mathsf{in}_\mathsf{U} \cdot \mathsf{G}\ (\!|\, \gamma \,|\!) \\
\equiv & \quad \{ \ \alpha \text{ is natural } \} \\
 & \gamma \cdot \alpha \cdot \mathsf{G}\ (\!|\, \gamma \,|\!) = \mathsf{in}_\mathsf{U} \cdot \mathsf{G}\ (\!|\, \gamma \,|\!) \\
\Leftarrow & \quad \{ \text{ Leibniz } \} \\
 & \gamma \cdot \alpha = \mathsf{in}_\mathsf{U} \\
\equiv & \quad \{ \ \beta \cdot \alpha = id \ \} \\
 & \gamma \cdot \alpha = \mathsf{in}_\mathsf{U} \cdot \beta \cdot \alpha \\
\Leftarrow & \quad \{ \text{ Leibniz } \} \\
 & \gamma = \mathsf{in}_\mathsf{U} \cdot \beta
\end{array}$$

Thus $f = (\!|\, \mathsf{in}_\mathsf{U} \cdot \beta \,|\!)$ is a left-inverse of $(\!|\, \mathsf{in}_\mathsf{T} \cdot \alpha \,|\!)$, that is

$$\beta \cdot \alpha = id \Rightarrow (\!|\, \mathsf{in}_\mathsf{U} \cdot \beta \,|\!) \cdot (\!|\, \mathsf{in}_\mathsf{T} \cdot \alpha \,|\!) = id$$

and therefore: $(\!|\, \mathsf{in}_{\mathsf{T}} \cdot \alpha \,|\!)$ is injective provided α is so; and $(\!|\, \mathsf{in}_{\mathsf{U}} \cdot \beta \,|\!)$ is surjective provided β is so.[20]

***Exercise** 3.38. Show that the function* mirror *of exercise 3.16 is an isomorphism.*
□

3.20 BIBLIOGRAPHY NOTES

It is often the case that the expressive power of a particular programming language or paradigm is counter-productive in the sense that too much freedom is given to programmers. Sooner or later, these will end up writing unintelligible (authorship dependent) code which will become a burden to whom has to maintain it. Such has been the case of imperative programming in the past (inc. assembly code), where the unrestricted use of `goto` instructions eventually gave place to `if-then-else`, `while` and `repeat` *structured* programming constructs.

A similar trend has been observed over the last decades at a higher programming level: arbitrary recursion and/or (side) effects have been considered harmful in functional programming. Instead, programmers have been invited to structure their code around generic program devices such as e.g. *fold/unfold* combinators, which bring discipline to recursion. One witnesses progress in the sense that the loss of freedom is balanced by the increase of formal semantics and the availability of program calculi.

Such disciplined programming combinators have been extended from list-processing to other inductive structures thanks to one of the most significant advances in programming theory over the last decade: the so-called *functorial* approach to datatypes which originated mainly from [53], was popularized by [52] and reached textbook format in [12]. A comfortable basis for exploiting *polymorphism* [85], the "datatypes as functors" moto has proved beneficial at a higher level of abstraction, giving birth to *polytypism* [40].

The literature on *anas*, *catas* and *hylos* is vast (see e.g. [56], [39], [28]) and it is part of a broader discipline which has become known as the *mathematics of program construction* [5]. This chapter provides an introduction to such as discipline. Only the calculus of catamorphisms is presented. The corresponding theory of anamorphisms and hylomorphisms demands further mathematical machinery (functions generalized to binary relations) and won't be dealt with before chapter 8. The results on mutual recursion presented in this chapter, pionered by Maarten Fokkinga [23], have been extended towards probabilistic functions [62]. They have also shown to help in program understand-

[20] This topic will be revisited in section 8.5 in a wider setting.

ing and reverse engineering [84]. Recently, the whole theory has undergone significant advances throught further use of category theory notions such as adjunctions [21] and conjugate functors [32, 33].

[21] See chapter 4.

4

WHY MONADS MATTER

In this chapter we present a powerful device in state-of-the-art functional programming, that of a *monad*. The monad concept is nowadays of primary importance in computing science because it makes it possible to describe computational effects as disparate as input/output, comprehension notation, state variable updating, probabilistic behaviour, context dependence, partial behaviour *etc.* in an elegant and uniform way.

Our motivation to this concept will start from a well-known problem in functional programming (and computing as a whole) — that of coping with undefined computations.

4.1 PARTIAL FUNCTIONS

Recall the function head that yields the first element of a finite list. Clearly, head x is undefined for $x = [\,]$ because the empty list has no elements at all. As expected, the HASKELL output for head $[\,]$ is just "panic":

```
*Main> head []
*** Exception: Prelude.head: empty list
*Main>
```

Functions such as head are called *partial functions* because they cannot be applied to all of their (well-typed) inputs, *i.e.*, they diverge for some of such inputs. Partial functions are very common in mathematics or programming — for other examples think of *e.g.* tail, and so on.

Panic is very dangerous in programming. In order to avoid this kind of behaviour one has two alternatives, either (a) ensuring that every call to head x is *protected* — *i.e.*, the contexts which wrap up such calls ensure *pre-condition* $x \neq [\,]$, or (b) *raising* exceptions, *i.e.* explicit error values, as above. In the former case, mathematical proofs need to be carried out in order to ensure *safety* (that is, *pre-condition* compliance). The overall effect is that of *restricting* the domain of the partial function. In the latter case one goes the other way round, by *extending* the co-domain (vulg. range) of the function so that it accommodates exceptional outputs. In this way one might define, in HASKELL:

data *ExtVal a* = *Ok a* | *Error*

and then define the "extended" version of head:

```
exthead :: [a] → ExtVal a
exthead [ ] = Error
exthead x = Ok (head x)
```

Note that *ExtVal* is a *parametric* type which extends an arbitrary data type *a* with its (polymorphic) exception (or error value). It turns out that, in HASKELL, *ExtVal* is redundant because such a parametric type already exists and is called *Maybe*:

```
data Maybe a = Nothing | Just a
```

Clearly, the isomorphisms hold:

$$\mathsf{ExtVal}\ A \cong \mathsf{Maybe}\ A \cong 1 + A$$

So, we might have written the more standard code

```
exthead :: [a] → Maybe a
exthead [ ] = Nothing
exthead x = Just (head x)
```

In abstract terms, both alternatives coincide, since one may regard as *partial* every function of type

$$\begin{tikzcd} 1+A & B \arrow[l, "g"'] \end{tikzcd}$$

for some A and B [1].

4.2 PUTTING PARTIAL FUNCTIONS TOGETHER

Do partial functions compose? Their types won't match in general:

$$\begin{tikzcd} & 1+B \arrow[d, dash, dotted] & A \arrow[l, "g"'] \\ 1+C & B \arrow[l, "f"'] & \end{tikzcd}$$

Clearly, we have to extend f — which is itself a partial function — to some f' able to accept arguments from $1 + B$:

$$\begin{tikzcd} & 1 \arrow[d, "i_1"] \arrow[ddl, "\cdots"'] & \\ & 1+B \arrow[dl, "f'"] & A \arrow[l, "g"'] \\ 1+C & B \arrow[l, "f"] \arrow[u, "i_2"'] & \end{tikzcd}$$

[1] In conventional programming, every function delivering a *pointer* as result — as in *e.g.* the C programming language — can be regarded as one of these functions.

The most "obvious" instance of the ellipsis (...) in the diagram above is i_1 and this corresponds to what is called *strict* composition: an exception produced by the *producer* function g is propagated to the output of the *consumer* function f. We define:

$$f \bullet g \stackrel{\mathrm{def}}{=} [i_1, f] \cdot g \qquad (4.1)$$

Expressed in terms of *Maybe*, composite function $f \bullet g$ works as follows:

$$(f \bullet g)a \;=\; f'(g\,a)$$

where

$$\begin{array}{l} f' \ \mathsf{Nothing} = \mathsf{Nothing} \\ f' \ (\mathsf{Just}\ b) = f\ b \end{array}$$

Altogether, we have the following Haskell pointwise expression for $f \bullet g$:

$$\begin{array}{l} \lambda a \rightarrow f'\ (g\ a)\ \textbf{where} \\ \quad f' \ \mathsf{Nothing} = \mathsf{Nothing} \\ \quad f' \ (\mathsf{Just}\ b) = f\ b \end{array}$$

Note that the adopted extension of f can be decomposed — by reverse $+$-absorption (2.43) — into

$$f' \;=\; [i_1, id] \cdot (id + f)$$

as displayed in diagram

$$\begin{tikzcd}
1 + (1 + C) \arrow[d, "{[i_1, id]}"'] & 1 + B \arrow[l, "id+f"'] \arrow[d, dash, dotted] & A \arrow[l, "g"'] \\
1 + C & B \arrow[l, "f"'] &
\end{tikzcd}$$

All in all, we have the following version of (4.1):

$$f \bullet g \stackrel{\mathrm{def}}{=} [i_1, id] \cdot (id + f) \cdot g$$

Does this functional composition scheme have a unit, that is, is there a function u such that

$$f \bullet u = f = u \bullet f \qquad (4.2)$$

holds? Clearly, if it exists, it must bear type $1 + A \xleftarrow{u} A$. $1 + A \xleftarrow{i_2} A$ has the same type, so $u = i_2$ is a very likely solution. Let us check it:

$$\begin{array}{ll}
& f \bullet u = f = u \bullet f \\
\equiv & \quad \{ \text{ substitutions } \} \\
& [i_1, f] \cdot i_2 = f = [i_1, i_2] \cdot f \\
\equiv & \quad \{ \text{ by } +\text{-cancellation (2.40) and } +\text{-reflection (2.41) } \} \\
& f = f = id \cdot f \\
\equiv & \quad \{ \text{ trivial } \} \\
& \mathit{true}
\end{array}$$

So $f \bullet u = f = u \bullet f$ for $u = i_2$.

***Exercise** 4.1. Prove that property*

$$f \bullet (g \bullet h) \quad = \quad (f \bullet g) \bullet h$$

holds, for $f \bullet g$ defined by (4.1).
□

4.3 LISTS

In contrast to partial functions, which may produce *no* output, let us now consider functions which may deliver *too many* outputs, for instance, lists of output values:

$$\begin{tikzcd}
 & B^\star \arrow[d, dotted, no head] & A \arrow[l, "g"'] \\
C^\star & B \arrow[l, "f"'] &
\end{tikzcd}$$

Functions f and g do not compose but, once again, one can think of extending the consumer function (f) by mapping it along the output of the producer function (g):

$$\begin{tikzcd}
(C^\star)^\star \arrow[d, dotted, no head] & B^\star \arrow[l, "f^\star"'] \arrow[d, dotted, no head] \\
C^\star & B \arrow[l, "f"']
\end{tikzcd}$$

To complete the process, one has to *flatten* the nested-sequence output in $(C^\star)^\star$ via the obvious list-catamorphism $C^\star \xleftarrow{\mathsf{concat}} (C^\star)^\star$, recall $\mathsf{concat} = (\!|\, [\underline{[\,]}, \mathsf{conc}] \,|\!)$ where $\mathsf{conc}\ (x, y) = x +\!\!+\, y$. In summary:

$$f \bullet g \quad \stackrel{\mathrm{def}}{=} \quad \mathsf{concat} \cdot f^\star \cdot g \tag{4.3}$$

as captured in the following diagram:

$$\begin{tikzcd}
(C^\star)^\star \arrow[d, "\mathsf{concat}"'] & B^\star \arrow[l, "f^\star"'] \arrow[d, dotted, no head] & A \arrow[l, "g"'] \\
C^\star & B \arrow[l, "f"'] &
\end{tikzcd}$$

***Exercise** 4.2. Show that* singl *(recall exercise 3.14) is the unit u of $\bullet$ as defined by (4.3) above.*
□

***Exercise** 4.3. Encode in* HASKELL *a pointwise version of (4.3). **Hint**: start by applying (list) cata-absorption (3.76).*

□

4.4 MONADS

Both function composition schemes (4.1) and (4.3) above share the same polytypic pattern: the output of the producer function g is *"*T*-times" more elaborate* than the input of the consumer function f, where T is some parametric datatype: $\mathsf{T}\,X = 1 + X$ in case of (4.1), and $\mathsf{T}\,X = X^{\star}$ in case of (4.3). Then a composition scheme is devised for such functions, which is displayed in

$$\begin{tikzcd}
\mathsf{T}(\mathsf{T}\,C) \arrow[d, "\mu"'] & \mathsf{T}\,B \arrow[l, "\mathsf{T}\,f"'] \arrow[d, dash, dotted] & A \arrow[l, "g"'] \arrow[dll, bend left, "f \bullet g"] \\
\mathsf{T}\,C & B \arrow[l, "f"'] &
\end{tikzcd} \tag{4.4}$$

and is given by

$$f \bullet g \overset{\text{def}}{=} \mu \cdot \mathsf{T}\,f \cdot g \tag{4.5}$$

where $\mathsf{T}\,A \xleftarrow{\mu} \mathsf{T}^2\,A$ is a suitable polymorphic function. (We have already seen $\mu = [i_1, id]$ in case (4.1), and $\mu = \mathsf{concat}$ in case (4.3).)

Together with a unit function $\mathsf{T}\,A \xleftarrow{u} A$ and μ, that is

$$A \xrightarrow{u} \mathsf{T}\,A \xleftarrow{\mu} \mathsf{T}^2\,A$$

datatype T will form a so-called *monad* type, of which $(1+)$ and $(\_)^{\star}$ are the two examples seen above. Arrow $\mu \cdot \mathsf{T}\,f$ is called the *extension* of f. Functions μ and u are referred to as the monad's *multiplication* and *unit*, respectively. The monadic composition scheme (4.5) is referred to as *Kleisli composition*.

A *monadic arrow* $\mathsf{T}B \xleftarrow{f} A$ conveys the idea of a function which produces an output of "type" B "wrapped by T", where datatype T describes some kind of (computational) "effect". The monad's unit $\mathsf{T}B \xleftarrow{u} B$ is a primitive monadic arrow which injects (*i.e.* promotes, wraps) data *inside* such an effect.

The monad concept is nowadays of primary importance in computing science because it makes it possible to describe computational effects as disparate as input/output, state variable updating, context dependence, partial behaviour (seen above) *etc.* in an elegant, generic and uniform way. Moreover, the monad's operators exhibit notable properties which make it possible to *reason* about such computational effects.

The remainder of this section is devoted to such properties. First of all, the properties implicit in the following diagrams will be *required* for T to be regarded as a monad:

Multiplication :

$$\begin{tikzcd}
\mathsf{T}^2\,A \arrow[d, "\mu"'] & \mathsf{T}^3\,A \arrow[l, "\mu"'] \arrow[d, "\mathsf{T}\,\mu"] \\
\mathsf{T}\,A & \mathsf{T}^2\,A \arrow[l, "\mu"]
\end{tikzcd}
\qquad \mu \cdot \mu = \mu \cdot \mathsf{T}\,\mu \tag{4.6}$$

Unit :

$$\begin{tikzcd}
\mathsf{T}^2\,A \arrow[d, "\mu"'] & \mathsf{T}\,A \arrow[l, "u"'] \arrow[d, "\mathsf{T}\,u"] \arrow[dl, "id"] \\
\mathsf{T}\,A & \mathsf{T}^2\,A \arrow[l, "\mu"]
\end{tikzcd}
\qquad \mu \cdot u = \mu \cdot \mathsf{T}\,u = id \tag{4.7}$$

The simple and beautiful symmetries apparent in these diagrams will make it easy to memorize their laws and check them for particular cases. For instance, for the $(1+)$ monad, law (4.7) will read as follows:

$$[i_1, id] \cdot i_2 = [i_1, id] \cdot (id + i_2) = id$$

These equalities are easy to check.

In laws (4.6) and (4.7), the different instances of μ and u are differently typed, as these are polymorphic and exhibit natural properties:

μ-natural :

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{T}\,A \arrow[d, "\mathsf{T}\,f"'] & \mathsf{T}^2\,A \arrow[l, "\mu"'] \arrow[d, "\mathsf{T}^2\,f"] \\
B & \mathsf{T}\,B & \mathsf{T}^2\,B \arrow[l, "\mu"]
\end{tikzcd}
\qquad \mathsf{T}\,f \cdot \mu = \mu \cdot \mathsf{T}^2\,f \tag{4.8}$$

u-natural :

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{T}\,A \arrow[d, "\mathsf{T}\,f"'] & A \arrow[l, "u"'] \arrow[d, "f"] \\
B & \mathsf{T}\,B & B \arrow[l, "u"]
\end{tikzcd}
\qquad \mathsf{T}\,f \cdot u = u \cdot f \tag{4.9}$$

The simplest of all monads is the *identity monad* $\mathsf{T}\,X \overset{\text{def}}{=} X$, which is such that $\mu = id$, $u = id$ and $f \bullet g = f \cdot g$. So — in a sense — the *whole functional discipline* studied thus far was already *monadic*, living inside the simplest of all monads: the identity one. Put in other words, such functional discipline can be framed into a wider discipline in which an arbitrary monad is present. Describing this is the main aim of the current chapter.

PROPERTIES INVOLVING (KLEISLI) COMPOSITION The following properties arise from the definitions and monadic properties presented above:

$$f \bullet (g \bullet h) = (f \bullet g) \bullet h \tag{4.10}$$

$$u \bullet f = f = f \bullet u \tag{4.11}$$

$$(f \bullet g) \cdot h = f \bullet (g \cdot h) \tag{4.12}$$

$$(f \cdot g) \bullet h = f \bullet (\mathsf{T}\, g \cdot h) \tag{4.13}$$

$$id \bullet id = \mu \tag{4.14}$$

Properties (4.10) and (4.11) are the monadic counterparts of, respectively, (2.8) and (2.10), meaning that monadic composition preserves the properties of normal functional composition. In fact, for the identity monad, these properties coincide with each other.

Above we have shown that property (4.11) holds for the list monad, recall (4.2). A general proof can be produced similarly. We select property (4.10) as an illustration of the rôle of the monadic properties:

$$\begin{array}{cl} & f \bullet (g \bullet h) \\ = & \quad \{ \text{ definition (4.5) twice } \} \\ & \mu \cdot \mathsf{T} f \cdot (\mu \cdot \mathsf{T}\, g \cdot h) \\ = & \quad \{ \ \mu \text{ is natural (4.8) } \} \\ & \mu \cdot \mu \cdot \mathsf{T}^2 f \cdot \mathsf{T}\, g \cdot h \\ = & \quad \{ \text{ monad property (4.6) } \} \\ & \mu \cdot \mathsf{T}\, \mu \cdot \mathsf{T}^2 f \cdot \mathsf{T}\, g \cdot h \\ = & \quad \{ \text{ functor } \mathsf{T} \text{ (3.55) } \} \\ & \mu \cdot \mathsf{T}\, (\mu \cdot \mathsf{T} f \cdot g) \cdot h \\ = & \quad \{ \text{ definition (4.5) twice } \} \\ & (f \bullet g) \bullet h \end{array}$$

Clearly, this calculation generalizes that of exercise 4.1 to any monad T.

***Exercise** 4.4. Prove the other laws above and also the following one,*

$$(\mathsf{T} f) \cdot (h \bullet k) = (\mathsf{T} f \cdot h) \bullet k \tag{4.15}$$

where Kleilsi composition again trades with normal composition.

□

4.5 MONADIC APPLICATION (BINDING)

We have seen above that, given a monad $A \xrightarrow{u} \mathsf{T}\,A \xleftarrow{\mu} \mathsf{T}^2\,A$, u is the unit of Kleisli composition, $f \bullet u = f$, recall (4.11). Now, what does happen in case we Kleisli compose f with the identity *id* of *standard* composition? Looking at diagram (4.4) for this case,

$$\begin{tikzcd}
\mathsf{T}(\mathsf{T}\,C) \arrow[d, "\mu"'] & \mathsf{T}\,B \arrow[l, "\mathsf{T}\,f"'] \arrow[d, dotted, no head] & \mathsf{T}\,B \arrow[l, "id"'] \\
\mathsf{T}\,C & B \arrow[l, "f"'] &
\end{tikzcd}$$

we realize that $f \bullet id$ accepts a value of type $\mathsf{T}\ B$ that is passed to $\mathsf{T}\,C \xleftarrow{f} B$, yielding an output of type $\mathsf{T}\ C$. This construction is called *binding* and denoted by $\gg\!\!= f$:

$$(\gg\!\!= f) = f \bullet id \tag{4.16}$$

Expressed pointwise, we get:[2]

$$x \gg\!\!= f \overset{\text{def}}{=} (\mu \cdot \mathsf{T}\,f)x \tag{4.17}$$

This operator exhibits properties that arise from its definition and the basic monadic properties, *e.g.*

$$\begin{aligned}
& x \gg\!\!= u \\
= & \quad \{ \text{ definition (4.17) } \} \\
& (\mu \cdot \mathsf{T}\,u)x \\
= & \quad \{ \text{ law (4.7) } \} \\
& (id)x \\
= & \quad \{ \text{ identity function } \} \\
& x
\end{aligned}$$

At pointwise level, one may chain monadic compositions from left to right, *e.g.*

$$(((x \gg\!\!= f_1) \gg\!\!= f_2) \gg\!\!= \ldots f_{n-1}) \gg\!\!= f_n$$

for functions $A \xrightarrow{f_1} \mathsf{T}\,B_1$, $B_1 \xrightarrow{f_2} \mathsf{T}\,B_2$, … $B_{n-1} \xrightarrow{f_n} \mathsf{T}\,B_n$.

4.6 SEQUENCING AND THE DO-NOTATION

Recall from above that $x \gg\!\!= f$ is the monadic *generalization* of function application $f\ x$, since both coincide for the identity monad. Also recall that, for $f = \underline{y}$ (the "everywhere"-y constant function) one gets $\underline{y}\ x = y$.

[2] In the case of the identity monad one has: $x \gg\!\!= f = f\ x$. So, $\gg\!\!=$ can be regarded as denoting *monadic function application*.

What does the corresponding monadic generalization, $x \ggg\!= \underline{y}$ mean? In the standard notation, this leads to another monadic operator,

$$x \gg y \quad \stackrel{\mathrm{def}}{=} \quad x \ggg\!= \underline{y} \qquad (4.18)$$

of type

$$(\gg) : \mathsf{T}\, A \to \mathsf{T}\, B \to \mathsf{T}\, B$$

called "sequencing". For instance, within the finite-list monad, one has

$$[1,2] \gg [3,4] = (\mathsf{concat} \cdot \underline{[3,4]^{\star}})[1,2] = \mathsf{concat}[[3,4],[3,4]] = [3,4,3,4]$$

Because this operator is associative[3], one may iterate it to more than two arguments and write, for instance,

$$x_1 \gg x_2 \gg \ldots \gg x_n$$

This leads to the popular "**do**-notation", which is another piece of (pointwise) notation which makes sense in a monadic context:

$$\mathbf{do}\ \{x_1; x_2; \ldots; x_n\} \quad \stackrel{\mathrm{def}}{=} \quad x_1 \gg \mathbf{do}\ \{x_2; \ldots; x_n\}$$

for $n \geqslant 1$. For $n = 1$ one trivially has

$$\mathbf{do}\ x_1 \quad \stackrel{\mathrm{def}}{=} \quad x_1$$

4.7 GENERATORS AND COMPREHENSIONS

The monadic **do**-notation paves the way to a device that is very useful in (pointwise) monadic programming. As before, we consider its (non-monadic) counterpart first. Consider for instance the expression $x + \mathsf{sum}\ y$, where sum is some operator in some context, e.g. adding up all elements of a list. Nothing impedes us from "structuring" expression $x + \mathsf{sum}\ y$ in the following way:

$$\begin{array}{l} \mathbf{let}\ a = \mathsf{sum}\ y \\ \mathbf{in}\ x + a \end{array}$$

It turns out that the above is the same as the following monadic expression,

$$\begin{array}{l} \mathbf{do}\ \{ \\ \quad a \leftarrow \mathsf{sum}\ y; \\ \quad u\ (x + a)\} \end{array}$$

provided the underlying monad is the *identity* monad. Now, what does the notation $a \leftarrow \ldots$ mean for an arbitrary monad $A \xrightarrow{u} \mathsf{T}\, A \xleftarrow{\mu} \mathsf{T}^2\, A$?

[3] See exercise 4.7 later on.

The **do**-notation accepts a variant in which the arguments of the $\gg$ operator are “generators” of the form

$$a \leftarrow x \tag{4.19}$$

where, for a of type A, x is an inhabitant of monadic type $\mathsf{T}\,A$. One may regard $a \leftarrow x$ as meaning “let a be taken from x”. Then the **do**-notation unfolds as follows:

$$\mathbf{do}\ \{a \leftarrow x_1; x_2; \ldots; x_n\} \overset{\mathrm{def}}{=} x_1 \gg\!= \lambda a \cdot (\mathbf{do}\ \{x_2; \ldots; x_n\}) \tag{4.20}$$

Of course, we should now allow for the x_i to range over terms involving variable a. For instance, by writing (again in the list-monad)

$$\mathbf{do}\ \{a \leftarrow [1,2,3]; [a^2]\} \tag{4.21}$$

we mean

$$\begin{array}{ll} & [1,2,3] \gg\!= \lambda a.[a^2] \\ = & \mathsf{concat}((\lambda a.[a^2])^{\star}[1,2,3]) \\ = & \mathsf{concat}[[1],[4],[9]] \\ = & [1,4,9] \end{array}$$

The analogy with classical set-theoretic ZF-notation, whereby one might write $\{a^2 \mid a \in \{1,2,3\}\}$ to describe the set of the first three perfect squares, suggests the following notation,

$$[\,a^2 \mid a \leftarrow [1,2,3]\,] \tag{4.22}$$

as a “shorthand” of (4.21). This is an instance of the so-called *comprehension* notation, which can be defined in general as follows:

$$[\,e \mid a_1 \leftarrow x_1, \ldots, a_n \leftarrow x_n\,] = \mathbf{do}\ \{a_1 \leftarrow x_1; \ldots; a_n \leftarrow x_n; u\} \tag{4.23}$$

where u is the monad’s unit (4.7,4.9).

Alternatively, comprehensions can be defined as follows, where p, q stand for arbitrary generators:

$$\begin{array}{rcl} [t] & = & u\,t \qquad (4.24) \\ [\,f\,x \mid x \leftarrow l\,] & = & (\mathsf{T}\,f)l \qquad (4.25) \\ [\,t \mid p, q\,] & = & \mu[\,[\,t \mid q\,] \mid p\,] \qquad (4.26) \end{array}$$

Note, however, that comprehensions are not restricted to lists or sets — they can be defined for any monad T thanks to the **do**-notation.

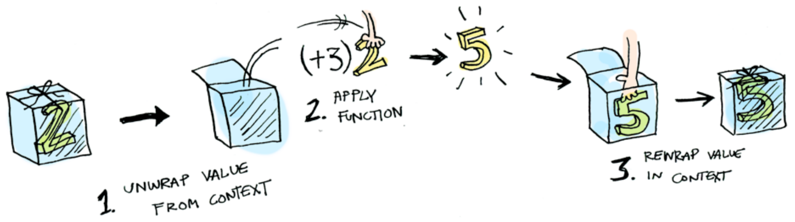
***Exercise* 4.5.** *Show that*

$$\begin{array}{rcl} (f \bullet g)\ a & = & \mathbf{do}\ \{b \leftarrow g\ a; f\ b\} \qquad (4.27) \\ \mathsf{T} f\ x & = & \mathbf{do}\ \{a \leftarrow x; u\ (f\ x)\} \qquad (4.28) \end{array}$$

Note that the second **do** *expression is equivalent to* $x \gg\!= (u \cdot f)$.

□

Fact (4.28) is illustrated in the cartoon[4]



for the computation of $\mathsf{T}\ (+3)\ x$, where $x = u\ 2$ is the T-monadic object containing number 2.

Exercise *4.6. Show that* $x \gg\!= f = \mathbf{do}\ \{a \leftarrow x; f\ a\}$ *and then that* $(x \gg\!= g) \gg\!= f$ *is the same as* $x \gg\!= f \bullet g$.
□

Exercise *4.7. Prove that* $(\gg)$ *is associative:*

$$x \gg (y \gg z) = (x \gg y) \gg z : \tag{4.29}$$

Hint: the laws of constant functions and the previous exercise can help your proof.
□

4.8 MONADS IN HASKELL

In the *Standard Prelude* for HASKELL, one finds the following minimal definition of the *Monad* class,

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Monad}\ m\ \mathbf{where} \\
\quad \mathsf{return} :: a \rightarrow m\ a \\
\quad (\gg\!=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b
\end{array}
$$

where return refers to the unit of m, on top of which the "sequence" operator

$$
\begin{array}{ll}
(\gg) & :: m\ a \rightarrow m\ b \rightarrow m\ b \\
\mathit{fail} & :: \mathit{String} \rightarrow m\ a
\end{array}
$$

is defined by

$$p \gg q = p \gg\!= \lambda\_ \rightarrow q$$

[4] Credits: see this and other helpful, artistic illustrations in
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html.

as expected. This class is instantiated for finite sequences ([ ]), *Maybe* and IO, among others.

The μ multiplication operator is function join in module `Monad.hs`:

$$\begin{array}{l} \mathsf{join} :: (Monad\ m) \Rightarrow m\ (m\ a) \rightarrow m\ a \\ \mathsf{join}\ x = x \gg\!= id \end{array}$$

This is easily justified:

$$\begin{array}{lcl} \mathsf{join}\ x & = & x \gg\!= id \\ & = & \qquad \{\ \text{definition (4.17)}\ \} \\ & & (\mu \cdot \mathsf{T}\ id)x \\ & = & \qquad \{\ \text{functors commute with identity (3.54)}\ \} \\ & & (\mu \cdot id)x \\ & = & \qquad \{\ \text{law (2.10)}\ \} \\ & & \mu\ x \end{array} \tag{4.30}$$

The following infix notation for (Kleisli) monadic composition in HASKELL uses the binding operator:

$$\begin{array}{l} (\bullet) :: Monad\ t \Rightarrow (b \rightarrow t\ c) \rightarrow (d \rightarrow t\ b) \rightarrow d \rightarrow t\ c \\ (f \bullet g)\ a = (g\ a) \gg\!= f \end{array}$$

***Exercise** 4.8. Consider the* HASKELL *function*

$$\begin{array}{l} discollect :: [(a, [b])] \rightarrow [(a, b)] \\ discollect\ [\,] = [\,] \\ discollect\ ((a, x) : y) = [(a, b) \mid b \leftarrow x] +\!\!+ discollect\ y \end{array}$$

Knowing that finite lists form a monad where $\mu = \mathsf{concat} = (\![[\mathsf{nil}, \mathsf{conc}]]\!)$ *and* $\mathsf{conc}\ (x, y) = x +\!\!+ y$*, derive the above pointfree code from the definition*

$$discollect = lstr \bullet id \tag{4.31}$$

where $lstr\ (a, x) = [(a, b) \mid b \leftarrow x]$.

□

MONADIC I/O IO, a parametric datatype whose inhabitants are special values called *actions* or *commands*, is a most relevant monad. Actions perform the interconnection between HASKELL and the environment (file system, operating system). For instance,

$$getLine :: \mathsf{IO}\ String$$

is a particular such action. Parameter *String* refers to the fact that this action "delivers" — or extracts — a string from the environent. This meaning is clearly conveyed by the type *String* assigned to symbol l in

$$\textbf{do}\ l \leftarrow getLine; \ldots l \ldots$$

which is consistent with typing rule for generators (4.19). Sequencing corresponds to the ";" syntax in most programming languages (*e.g.* C) and the **do**-notation is particulary intuitive in the IO-context.

Examples of functions delivering actions are

$$FilePath \xrightarrow{readFile} \mathsf{IO}\ String$$

and

$$Char \xrightarrow{putChar} \mathsf{IO}\ ()$$

— both produce I/O commands as result.

As is to be expected, the implementation of the IO monad in HASKELL — available from the *Standard Prelude* — is not totally visible, for it is bound to deal with the intrincacies of the underlying machine:

$$\begin{array}{l} \textbf{instance}\ Monad\ \mathsf{IO}\ \textbf{where} \\ \quad (\gg\!=) = primbindIO \\ \quad \mathsf{return} = primretIO \end{array}$$

Rather interesting is the way IO is regarded as a functor:

$$fmap\ f\ x = x \gg\!= (\mathsf{return} \cdot f)$$

This goes the other way round, the monadic structure "helping" in defining the functor structure, everything consistent with the underlying theory:

$$\begin{array}{rcl} x \gg\!= (u \cdot f) & = & (\mu \cdot \mathsf{IO}(u \cdot f))x \\ & = & \qquad \{ \text{ functors commute with composition } \} \\ & & (\mu \cdot \mathsf{IO}\, u \cdot \mathsf{IO}\, f)x \\ & = & \qquad \{ \text{ law (4.7) for } \mathsf{T} = \mathsf{IO} \ \} \\ & & (\mathsf{IO}\, f)x \\ & = & \qquad \{ \text{ definition of } fmap \ \} \\ & & (fmap\ f)\ x \end{array}$$

For enjoyable reading on monadic input/output in HASKELL see [35], chapter 18.

***Exercise** 4.9. Extend the* Maybe *monad to the following "error message" exception handling datatype:*

$$\textbf{data}\ Error\ a = Err\ String \mid Ok\ a\ \textbf{deriving}\ Show$$

In case of several error messages issued in a `do` *sequence, how many turn up on the screen? Which ones?*

□

***Exercise** 4.10. Recalling section 3.13, show that any inductive type with base functor*

$$\mathsf{B}\ (f,g) = f + \mathsf{F}\ g$$

where F *is an arbitrary functor, forms a monad for*

$$\mu = (\![[id, \mathsf{in} \cdot i_2]]\!)$$
$$u = \mathsf{in} \cdot i_1.$$

Identify F *for known monads such as e.g.* Maybe, LTree *and (non-empty) lists.*

□

4.9 THE STATE MONAD

The so-called *state monad* is a monad whose inhabitants are state transitions encoding a particular brand of state-based automata known as *Mealy machines*. Given a set A (input alphabet), a set B (output alphabet) and a set of states S, a deterministic Mealy machine (DMM) is specified by a transition function of type

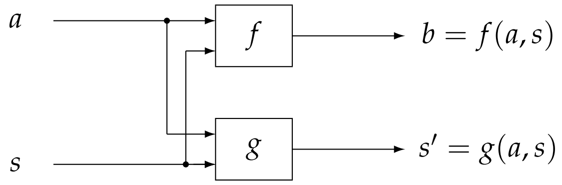
$$A \times S \xrightarrow{\delta} B \times S \tag{4.32}$$

Wherever $(b, s') = \delta(a, s)$, we say that the machine has transition

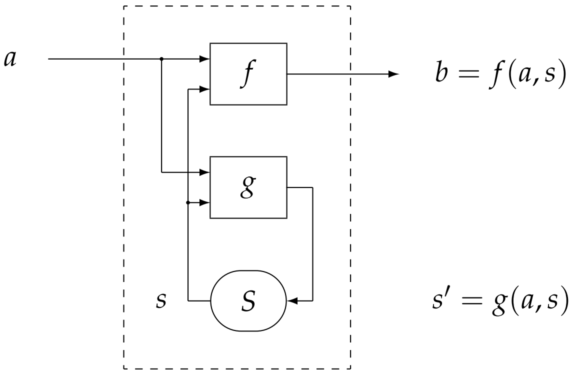
$$s \xrightarrow{a|b} s'$$

and refer to s as the *before* state, and to s' as the *after* state. Many programs that one writes in conventional programming languages such as C or Java can be regarded as DMMs.

It is clear from (4.32) that δ can be expressed as the *split* of two functions f and g — $\delta = \langle f, g\rangle$ — as depicted in the following drawing:



Note, however, that the information recorded in the state of a DMM is either meaningless to the user of the machine (as in e.g. the case of states represented by numbers) or too complex to be perceived and handled explicitly (as is the case of e.g. the data kept in a large database). So, it is convenient to *abstract* from it, via the "encapsulation" suggested by the following, transformed, version of the previous drawing,



(4.33)

in which the state is no longer accessible from the outside.

Such an abstraction is nicely captured by the so-called *state monad*, in the following way: taking (4.32) and recalling (2.93), we simply transpose (ie. *curry*) δ and obtain

$$A \xrightarrow{\overline{\delta}} \underbrace{(B \times S)^S}_{(\mathsf{St}\ S)\ B} \tag{4.34}$$

thus "shifting" the *input* state to the *output*. In this way, $\overline{\delta}\ a$ is a function capturing all state-transitions (and corresponding outputs) for input a. For instance, the function that *appends* a new element at the rear of a queue,

$$enq(a, s) \overset{\mathrm{def}}{=} s + [a]$$

can be converted into a DMM by adding to it a dummy output of type 1 and then transposing:

$$\begin{array}{rcl} enqueue & : & A \rightarrow (1 \times S)^S \\ enqueue\ a & \overset{\mathrm{def}}{=} & \langle !, (+[a]) \rangle \end{array} \tag{4.35}$$

Action *enqueue* performs *enq* on the state while acknowledging it by issuing an output of type 1.[5]

UNIT AND MULTIPLICATION. Let us now show that

$$(\mathsf{St}\ S)\ A \cong (A \times S)^S \tag{4.36}$$

forms a monad. As we shall see, the fact that the *values* of this monad are functions brings the theory of exponentiation to the forefront. Thus, a review of section 2.15 is recommended at this point.

[5] A kind of "done!" message.

Notation $\widehat{f}$ will be used to abbreviate *uncurry* f, enabling the following variant of universal law (2.83),

$$\widehat{k} = f \quad \Leftrightarrow \quad f = ap \cdot (k \times id) \tag{4.37}$$

whose cancellation

$$\widehat{k} = ap \cdot (k \times id) \tag{4.38}$$

is written pointwise as follows:

$$\widehat{k}(c, a) \quad = \quad (k\ c)a \tag{4.39}$$

First of all, what is the functor behind (4.36)? Fixing the state space S, we obtain

$$\mathsf{T}X \quad \stackrel{\text{def}}{=} \quad (X \times S)^S \tag{4.40}$$

on objects and

$$\mathsf{T}f \quad \stackrel{\text{def}}{=} \quad (f \times id)^S \tag{4.41}$$

on functions, where $(\_)^S$ is the exponential functor (2.87).

The unit of this monad is the transpose of the simplest of all Mealy machines — the identity:

$$\begin{array}{lcl} u & : & A \rightarrow (A \times S)^S \\ u & = & \overline{id} \end{array} \tag{4.42}$$

Let us see what this means:

$$\begin{array}{cl} & u = \overline{id} \\ \equiv & \qquad \{ \text{ (2.83) } \} \\ & ap \cdot (u \times id) = id \\ \equiv & \qquad \{ \text{ introducing variables } \} \\ & ap(u\ a, s) = (a, s) \\ \equiv & \qquad \{ \text{ definition of } ap \ \} \\ & (u\ a)s = (a, s) \end{array}$$

So, action $u\ a$ performed on state s keeps s unchanged and outputs a.

From the type of μ, for this monad,

$$((A \times S)^S \times S)^S \xrightarrow{\ \mu\ } (A \times S)^S$$

one figures out $\mu = x^S$ (recalling the exponential functor as defined by (2.87)) for some $((A \times S)^S \times S) \xrightarrow{\ x\ } (A \times S)$. This, on its turn, is easily recognized as an instance of the ap polymorphic function (2.83), which is such that $ap = \widehat{id}$, recall (2.85). Altogether, we define

$$\mu \quad = \quad ap^S \tag{4.43}$$

Let us inspect the behaviour of μ by checking the meaning of applying it to an action expressed as in diagram (2.93):

$$\begin{array}{ll}
 & \mu\langle f,g\rangle = ap^S\langle f,g\rangle \\
\equiv & \qquad \{ \text{ (2.87) } \} \\
 & \mu\langle f,g\rangle = ap \cdot \langle f,g\rangle \\
\equiv & \qquad \{ \text{ extensional equality (2.5) } \} \\
 & \mu\langle f,g\rangle s = ap(f\ s, g\ s) \\
\equiv & \qquad \{ \text{ definition of } ap \ \} \\
 & \mu\langle f,g\rangle s = (f\ s)(g\ s)
\end{array}$$

We find out that μ "unnests" the action inside f by applying it to the state delivered by g.

CHECKING THE MONADIC LAWS. The calculation of (4.7) is made in two parts, checking $\mu \cdot u = id$ first,

$$\begin{array}{ll}
 & \mu \cdot u \\
= & \qquad \{ \text{ definitions } \} \\
 & ap^S \cdot \overline{id} \\
= & \qquad \{ \text{ exponentials absorption (2.88) } \} \\
 & \overline{ap \cdot id} \\
= & \qquad \{ \text{ reflection (2.85) } \} \\
 & id \\
\Box &
\end{array}$$

and then checking $\mu \cdot (\mathsf{T}u) = id$:

$$\begin{array}{ll}
 & \mu \cdot (\mathsf{T}u) \\
= & \qquad \{ \text{ (4.43,4.41) } \} \\
 & ap^S \cdot (\overline{id} \times id)^S \\
= & \qquad \{ \text{ functor } \} \\
 & (ap \cdot (\overline{id} \times id))^S \\
= & \qquad \{ \text{ cancellation (2.84) } \} \\
 & id^S \\
= & \qquad \{ \text{ functor } \} \\
 & id \\
\Box &
\end{array}$$

The proof of (4.6) is also not difficult once supported by the laws of exponentials.

KLEISLI COMPOSITION. Let us calculate $f \bullet g$ for this monad:

$$\begin{array}{cl} & f \bullet g \\ = & \quad \{ \text{ (4.5) } \} \\ & \mu \cdot \mathsf{T} f \cdot g \\ = & \quad \{ \text{ (4.43) ; (4.41) } \} \\ & ap^S \cdot (f \times id)^S \cdot g \\ = & \quad \{ \text{ (\_)}^S \text{ is a functor } \} \\ & (ap \cdot (f \times id))^S \cdot g \\ = & \quad \{ \text{ (4.37) } \} \\ & \widehat{f}^S \cdot g \\ = & \quad \{ \text{ cancellation } \} \\ & \widehat{f}^S \cdot \overline{\widehat{g}} \\ = & \quad \{ \text{ absorption (2.88) } \} \\ & \overline{\widehat{f} \cdot \widehat{g}} \end{array}$$

In summary, we have:

$$f \bullet g \;=\; \overline{\widehat{f} \cdot \widehat{g}} \qquad (4.44)$$

which can be written alternatively as

$$\widehat{f \bullet g} \;=\; \widehat{f} \cdot \widehat{g}$$

Let us use this in calculating law

$$pop \bullet push \;=\; u \qquad (4.45)$$

where *push* and *pop* are such that

$$\begin{array}{lcl} push & : & A \rightarrow (1 \times S)^S \\ \widehat{push} & \overset{\mathrm{def}}{=} & \langle !, \widehat{(:)} \rangle \end{array} \qquad (4.46)$$

$$\begin{array}{lcl} pop & : & 1 \rightarrow (A \times S)^S \\ \widehat{pop} & \overset{\mathrm{def}}{=} & \langle head, tail \rangle \cdot \pi_2 \end{array} \qquad (4.47)$$

for S the datatype of finite lists. We reason:

$$\begin{array}{cl} & pop \bullet push \\ = & \quad \{ \text{ (4.44) } \} \\ & \overline{\widehat{pop} \cdot \widehat{push}} \\ = & \quad \{ \text{ (4.46, 4.47) } \} \\ & \overline{\langle head, tail \rangle \cdot \pi_2 \cdot \langle !, \widehat{(:)} \rangle} \end{array}$$

$$
\begin{array}{ll}
= & \qquad \{\ (2.22, 2.26)\ \} \\
& \overline{\langle head, tail\rangle \cdot \widehat{(:)}} \\
= & \qquad \{\ out \cdot in = id \text{ (lists)}\ \} \\
& \overline{id} \\
= & \qquad \{\ (4.42)\ \} \\
& u \\
\Box &
\end{array}
$$

BIND. The effect of binding a state transition x to a state-monadic function h is calculated in a similar way:

$$
\begin{array}{ll}
& x \gg= h \\
= & \qquad \{\ (4.17)\ \} \\
& (\mu \cdot \mathsf{T}h)x \\
= & \qquad \{\ (4.43) \text{ and } (4.41)\ \} \\
& (ap^S \cdot (h \times id)^S)x \\
= & \qquad \{\ (\_)^S \text{ is a functor}\ \} \\
& (ap \cdot (h \times id))^S x \\
= & \qquad \{\ \text{cancellation } (4.38)\ \} \\
& \widehat{h}^S x \\
= & \qquad \{\ \text{exponential functor } (2.87)\ \} \\
& \widehat{h} \cdot x
\end{array}
$$

Let us unfold $\widehat{h} \cdot x$ by splitting x into its components two components f and g:

$$
\begin{array}{ll}
& \langle f, g\rangle \gg= h \;=\; \widehat{h} \cdot \langle f, g\rangle \\
\equiv & \qquad \{\ \text{go pointwise}\ \} \\
& (\langle f, g\rangle \gg= h)s \;=\; \widehat{h}(\langle f, g\rangle s) \\
\equiv & \qquad \{\ (2.20)\ \} \\
& (\langle f, g\rangle \gg= h)s \;=\; \widehat{h}(f\ s, g\ s) \\
\equiv & \qquad \{\ (4.39)\ \} \\
& (\langle f, g\rangle \gg= h)s \;=\; h(f\ s)(g\ s)
\end{array}
$$

In summary, for a given "before state" s, $g\ s$ is the intermediate state upon which $f\ s$ runs and yields the output and (final) "after state".

TWO PROTOTYPICAL INHABITANTS OF THE STATE MONAD: *get* AND *put*. These generic actions are defined as follows, in the PF-style:

$$get \overset{\text{def}}{=} \langle id, id \rangle \tag{4.48}$$

$$put \overset{\text{def}}{=} \overline{\langle !, \pi_1 \rangle} \tag{4.49}$$

Action *g* retrieves the data stored in the state without changing it, while *put* stores a particular value in the state. Note that *put* can also be written

$$put\ s = \langle !, \underline{s} \rangle \tag{4.50}$$

or even as

$$put\ s = update\ \underline{s} \tag{4.51}$$

where

$$update\, f \overset{\text{def}}{=} \langle !, f \rangle \tag{4.52}$$

updates the state via state-to-state function f.

The following is an example, written in Haskell, of the standard use of *get*/*put* in managing context data, in this case a counter. Function *decBTree* decorates each node of a *BTree* (recall this datatype from page 102) with its position in the tree:

```
decBTree Empty = return Empty
decBTree (Node (a, (t1, t2))) = do {
   n ← get;
   put (n + 1);
   x ← decBTree t1;
   y ← decBTree t2;
   return (Node ((a, n), (x, y)))
   }
```

To close the chapter, we will present a strategy for deriving this kind of monadic functions.

4.10 'MONADIFICATION' OF HASKELL CODE MADE EASY

There is an easy roadmap for "monadification" of Haskell code. What do we mean by *monadification*? Well, in a sense — as we shall soon see — every piece of code is monadic: we don't notice this because the underlying monad is *invisible* (the *identity* monad). We are going to see how to make it visible taking advantage of monadic do notation and leaving it open for instantiation. This will bridge the gap between monads' theory and its application to handling particular effects in concrete programming situations.

Let us take as starting point the pointwise version of *sum*, the list catamorphism that adds all numbers found in its input:

$$\mathsf{sum}\ [\,] = 0$$
$$\mathsf{sum}\ (h : t) = h + \mathsf{sum}\ t$$

Notice that this code could have been written as follows

$$\mathsf{sum}\ [\,] = id\ 0$$
$$\mathsf{sum}\ (h : t) = \mathbf{let}\ x = \mathsf{sum}\ t\ \mathbf{in}\ id\ (h + x)$$

using *let* notation and two instances of the identity function. Question: what is the point of such a "baroque" version of the starting, so simple piece of code? Answer:

- The *let* ... *in* ... notation stresses the fact that recursive call happens *earlier* than the delivery of the result.
- The *id* functions signal the exit points of the algorithm, that is, the points where it *returns* something to the caller.

Next, let us

- re-write *id* into return—;
- re-write let x = ... in ...— into `do { x <- ... ; ... }`

One will obtain the following version of *sum*:

$$msum\ [\,] = \mathsf{return}\ 0$$
$$msum\ (h : t) = \mathbf{do}\ \{x \leftarrow msum\ t; \mathsf{return}\ (h + x)\}$$

Typewise, while *sum* has type $(Num\ a) \Rightarrow [a] \rightarrow a$, *msum* has type

$$(Monad\ m, Num\ a) \Rightarrow [a] \rightarrow m\ a$$

That is, *msum* is monadic — parametric on monad *m* — while *sum* is not.

There is a particular monad for which *sum* and *msum* coincide: the **identity** monad $\mathsf{Id}\ X = X$. It is very easy to show that inside this monad return is the identity and $\mathbf{do}\ x \leftarrow \ldots$ means the same as $\mathbf{let}\ x = \ldots$, as already mentioned — enough for the pointwise versions of the two functions to be the same. Thus the "invisible" monad mentioned earlier is the identity monad.

In summary, the monadic version is *generic* in the sense that it runs on whatever monad you like, enabling you to perform *side effects* while the code runs. If you don't need any effects then you get the "non-monadic" version as special case, as seen above. Otherwise, Haskell will automatically switch to the effects you want, depending on the monad you choose (often determined by context).

For each particular monad we may decide to add specific monadic code like `get` and `put` in the `decBTree` example, where we want to take advantage of the state monad. As another example, check the following enrichment of *msum* with state-monadic code helping you to trace the execution of your program:

```
msum′ [ ] = return 0
msum′ (h : t) =
  do {x ← msum′ t;
    print ("x=  " ++ show x);
    return (h + x)}
```

Thus one obtains traces of the code in the way prescribed by the particular usage of the *print* (state monadic) function:

```
*Main> msum' [3,5,1,3,4]
"x= 0"
"x= 4"
"x= 7"
"x= 8"
"x= 13"
*Main>
```

In the reverse direction, one may try and see what happens to monadic code upon removing all monad-specific functions and going into the identity monad once it gets monad generic. In the case of `decBTree`, for instance, we will get

```
decBTree Empty = return Empty
decBTree (Node (a, (t1, t2))) =
  do
    x ← decBTree t1;
    y ← decBTree t2;
    return (Node (a, (x, y)))
```

once `get` and `put` are removed (and therefore all instances of `n`), and then

```
decBTree Empty = Empty
decBTree (Node (a, (t1, t2))) =
  let
    x = decBTree t1
    y = decBTree t2
  in Node (a, (x, y))
```

This is the identity function on type BTree, recall the cata-reflection law (3.67). So, the *archetype* of (inspiration for) much monadic code is the most basic of all tree traversal functions — the identity [6]. The same could be said about imperative code of a particular class — the *recursive descent* one — much used in compiler construction, for instance.

[6] We have seen the same kind of "inspiration" before in building type functors (3.75) which, for $f = id$, boil down to the identity.

Playing with effects

As it may seem from the previous examples, adding effects to produce monadic code is far from arbitrary. This can be further appreciated by defining the function that yields the smallest element of a list,

$$getmin\ [a] = a$$
$$getmin\ (h : t) = min\ h\ (getmin\ t)$$

which is incomplete in the sense that it does not specify the meaning of *getmin* [ ]. As this is mathematically undefined, it should be expressed "outside the maths", that is, as an effect. Thus, to complete the defintion we first go monadic, as we did before,

$$mgetmin\ [a] = \mathsf{return}\ a$$
$$mgetmin\ (h : t) = \mathbf{do}\ \{x \leftarrow mgetmin\ t; \mathsf{return}\ (min\ h\ x)\}$$

and then chose a monad in which to express the meaning of *getmin* [ ], for instance the `Maybe` monad

$$mgetmin\ [\,] = \mathsf{Nothing}$$
$$mgetmin\ [a] = \mathsf{return}\ a$$
$$mgetmin\ (h : t) = \mathbf{do}\ \{x \leftarrow mgetmin\ t; \mathsf{return}\ (min\ h\ x)\}$$

Alternatively, we might have written

$$mgetmin\ [\,] = Error\ \texttt{"Empty input"}$$

going into the `Error` monad, or even the simpler (yet interesting) *mgetmin* [ ] = [ ], which shifts the code into the list monad, yielding singleton lists in the success case, otherwise the empty list.

Function *getmin* above is an example of a partial function, that is, a function which is undefined for some of its inputs.[7] These functions cause much interference in functional programming, which monads help us to keep under control.

Let us see how such interference is coped with in the case of higher order functions, taking map as example

$$\mathsf{map}\,f\ [\,] = [\,]$$
$$\mathsf{map}\,f\ (h : t) = (f\ h) : \mathsf{map}\,f\ t$$

and supposing *f* is not a total function. How do we cope with erring evaluations of *f h*? As before, we first "letify" the code,

$$\mathsf{map}\,f\ [\,] = [\,]$$
$$\mathsf{map}\,f\ (h : t) = \mathbf{let}$$
$$\quad b = f\ h$$
$$\quad x = \mathsf{map}\,f\ t\ \mathbf{in}\ b : x$$

[7] Recall that function partiality was our motivation for studying monads right from the beginning of this chapter.

we go monadic in the usual way,

$$\begin{array}{l} \mathit{mmap}\, f\ [\,] = \mathsf{return}\ [\,] \\ \mathit{mmap}\, f\ (h : t) = \mathbf{do}\ \{ b \leftarrow f\ h; x \leftarrow \mathit{mmap}\, f\ t; \mathsf{return}\ (b : x) \} \end{array}$$

and everything goes smoothly — as can be checked, the function thus built is of the expected (monadic) type:

$$\mathit{mmap} :: (\mathit{Monad}\ \mathsf{T}) \Rightarrow (a \to \mathsf{T}\ b) \to [a] \to \mathsf{T}\ [b] \tag{4.53}$$

Run *mmap* Just $[1,2,3,4]$, for instance: you will obtain Just $[1,2,3,4]$. Now run *mmap print* $[1,2,3,4]$. You will see the items in the sequence printed sequentially.

One may wonder about the behaviour of the *mmap* for f the identity function: will we get an error? No, we get a well-typed function of type $[m\ a] \to m\ [a]$, for m a monad. We thus obtain the well-known monadic function sequence which evaluates each *action* in the input sequence, from left to right, collecting the results. For instance, applying this function to input sequence [Just 1, Nothing, Just 2] the output will be Nothing.

***Exercise** 4.11. Use the* monadification *technique to encode monadic function*

$$\mathit{filterM} :: \mathit{Monad}\ m \Rightarrow (a \to m\ \mathbb{B}) \to [a] \to m\ [a]$$

which generalizes the list-based filter function.
□

***Exercise** 4.12. "Reverse" the following monadic code into its non-monadic archetype:*

$$\begin{array}{l} f :: (\mathit{Monad}\ m) \Rightarrow (a \to m\ \mathbb{B}) \to [a] \to m\ [a] \\ f\ p\ [\,] = \mathsf{return}\ [\,] \\ f\ p\ (h : t) = \mathbf{do}\ \{ \\ \quad b \leftarrow p\ h; \\ \quad t' \leftarrow f\ p\ t; \\ \quad \mathsf{return}\ (\mathbf{if}\ b\ \mathbf{then}\ h : t'\ \mathbf{else}\ [\,]) \\ \quad \} \end{array}$$

Which function of the Haskell Prelude do you get by such reverse monadification*?*
□

4.11 MONADIC RECURSION

There is much more one could say about monadic recursive programming. In particular, one can express the code "monadification" strategies of the previous section in terms of catamorphisms. As an example, recall (4.53):

$$\begin{tikzcd}
A \arrow[d, "f"'] & A^\star \arrow[d, "mmap\,f"'] & 1+A\times A^* \arrow[l, "in_{A^\star}"'] \arrow[d, "id+id\times mmap\,f"] \\
\mathsf{T}\,B & \mathsf{T}\,B^* & 1+A\times\mathsf{T}\,B^* \arrow[l, dotted, "g"']
\end{tikzcd}$$

How do we build g? Clearly, the recipe given by (3.75) needs to be adapted:

$$\begin{tikzcd}
A \arrow[d, "f"'] & A^\star \arrow[d, "mmap\,f"'] & 1+A\times A^* \arrow[l, "in_{A^\star}"'] \arrow[d, "id+id\times mmap\,f"] \\
\mathsf{T}\,B & \mathsf{T}\,B^* & 1+A\times\mathsf{T}\,B^* \arrow[l, dotted, "g"'] \arrow[d, "id+f\times id"] \\
 & & 1+\mathsf{T}\,B\times\mathsf{T}\,B^* \arrow[ul, dotted, "{[\mathsf{return}\cdot\mathsf{nil}, \lfloor\mathsf{cons}\rfloor]}"]
\end{tikzcd}$$

where

$$\lfloor f\rfloor\ (x,y) = \mathbf{do}\ \{a\leftarrow x; b\leftarrow y; \mathsf{return}\ (f\ (a,b))\}$$

By defining

$$\begin{array}{l}
\llparenthesis g\rrparenthesis^\flat = \llparenthesis\,[\mathsf{return}\cdot f, \lfloor h\rfloor]\,\rrparenthesis\ \mathbf{where} \\
\quad f = (g\cdot i_1) \\
\quad h = (g\cdot i_2)
\end{array}$$

we can write

$$mmap\,f = \llparenthesis\,(\mathsf{in}\cdot(id+f\times id))\,\rrparenthesis^\flat \tag{4.54}$$

where (recall) $\mathsf{in} = [\mathsf{nil}, \mathsf{cons}]$.

Handling monadic recursion in full generality calls for technical ingredients called *commutative laws* which fall outside the current scope of this chapter.

4.12 WHERE DO MONADS COME FROM?

In the current context, a good way to find an answer this question is to recall the universal property of exponentials (2.83):

$$k = \overline{f} \Leftrightarrow f = ap\cdot(k\times id) \qquad \begin{tikzcd}
B^A & B^A\times A \arrow[r, "ap"] & B \\
C \arrow[u, "k=\overline{f}"] & C\times A \arrow[u, "k\times id"] \arrow[ur, "f"'] &
\end{tikzcd}$$

Let us re-draw this diagram by unfolding $B^A\times A$ into the composition of two functors $\mathsf{G}\ (\mathsf{F}\ B)$ where $\mathsf{F}\ X = X^A$ and $\mathsf{G}\ X = X\times A$:

$$k = \overline{f} \Leftrightarrow f = \underbrace{ap\cdot\mathsf{G}\,k}_{\widehat{k}} \qquad \begin{tikzcd}
\mathsf{F}\,B & \mathsf{G}\ (\mathsf{F}\ B) \arrow[r, "ap"] & B \\
C \arrow[u, "k=\overline{f}"] & \mathsf{G}\ C \arrow[u, "\mathsf{G}\,k"] \arrow[ur, "f"'] &
\end{tikzcd} \tag{4.55}$$

As we already know, this establishes the (*curry*/*uncurry*) isomorphism

$$\mathsf{G}\, C \to B \;\cong\; C \to \mathsf{F}\, B \tag{4.56}$$

assuming F and G as defined above.

Note how (4.56) expresses a kind of "shunting rule" at type level: Gs on the input side can be "shunted" to the output if replaced by Fs. This is exactly what *curry* and *uncurry* do typewise. The corollaries of the universal property can also be expressed in terms of F and G:

- Reflection: $\overline{ap} = id$, that is, $ap = \widehat{id}$ – recall (2.85)
- Cancellation: $\widehat{id} \cdot \mathsf{G}\, \overline{f} = f$ – recall (2.84)
- Fusion: $\overline{h} \cdot g = \overline{h \cdot \mathsf{G}\, g}$ — recall (2.86)
- Absorption: $(\mathsf{F}\, g) \cdot \overline{h} = \overline{g \cdot h}$ — recall (2.88)
- Naturality: $h \cdot \widehat{id} = \widehat{id} \cdot \mathsf{G}\, (\mathsf{F}\, h)$
- Functor: $\mathsf{F}\, h = \overline{h \cdot ap}$
- Closed definitions: $\widehat{k} = \widehat{id} \cdot (\mathsf{G}\; k)$ and $\overline{g} = (\mathsf{F}\; g) \cdot \overline{id}$, the latter following from absorption.

Now observe what happens if the functor composition $\mathsf{G} \cdot \mathsf{F}$ is swapped: $\mathsf{F}\, (\mathsf{G}\, X) = (X \times A)^A$. We get the *state monad* out of this construction,

$$(\mathsf{G} \cdot \mathsf{F})\; X = (X \times A)^A = St\; A\; X$$

— recall (4.36). Interestingly, the universal property (4.59) can be expressed also in terms of such a monad structure, as the simple calculation shows:

$$\begin{aligned}
& k = \overline{f} \;\Leftrightarrow\; ap \cdot \mathsf{G}\, k = f \\
\equiv \quad & \quad \{ \text{ see above } \} \\
& k = (\mathsf{F} f) \cdot \overline{id} \;\Leftrightarrow\; f = \widehat{k} \\
\equiv \quad & \quad \{ \text{ swapping variables } k \text{ and } f\text{, to match the starting diagram } \} \\
& f = (\mathsf{F}\, k) \cdot \overline{id} \;\Leftrightarrow\; k = \widehat{f}
\end{aligned}$$

That is,

$$k = \widehat{f} \;\Leftrightarrow\; f = \underbrace{\mathsf{F}\, k \cdot \eta}_{\overline{k}}$$

$$\begin{tikzcd}
\mathsf{G}\, B \arrow[d, "k=\widehat{f}"'] & \mathsf{F}\, (\mathsf{G}\, B) \arrow[d, "\mathsf{F}\, k"'] & B \arrow[l, "\eta"'] \arrow[dl, "f"] \\
C & \mathsf{F}\, C &
\end{tikzcd}$$

for $\eta = \overline{id}$, the unit of the monad $\mathsf{T} = \mathsf{F} \cdot \mathsf{G}$. To complete the definition of the T monad in this way, we recall (4.43)

$$\mu = \mathsf{F}\, \widehat{id} \tag{4.57}$$

with type $(\mathsf{T} \cdot \mathsf{T})\; X \xrightarrow{\mu} \mathsf{T}\; X$, where $id : \mathsf{T}\; X \to \mathsf{T}\; X$.

Adjunctions

The reasoning we have made above for exponentials and the state monad generalizes for any other monad. In general, an isomorphism of shape (4.56) is called an *adjunction* of the two functors F and G, which are said to be *adjoint* to each other. One writes $\mathsf{G} \dashv \mathsf{F}$ and says that G is *left* adjoint and that F is *right* adjoint. Using notation $\lfloor k \rfloor$ and $\lceil k \rceil$ for the generic witnesses of the isomorphism we write

$$\begin{tikzcd}
\mathsf{G}\,C \to B \arrow[rr, bend left, "\lceil - \rceil"] \arrow[rr, phantom, "\cong"] & & C \to \mathsf{F}\,B \arrow[ll, bend left, "\lfloor - \rfloor"]
\end{tikzcd} \tag{4.58}$$

cf.

$$k = \bar{f} \;\Leftrightarrow\; f = \underbrace{\epsilon \cdot \mathsf{L}\,k}_{\widehat{k}} \qquad \begin{tikzcd}
\mathsf{R}\,B & \mathsf{L}\,(\mathsf{R}\,B) \arrow[r, "\epsilon"] & B \\
A \arrow[u, "k=\bar{f}"] & \mathsf{L}\,A \arrow[u, "\mathsf{L}\,k"] \arrow[ur, "f"'] &
\end{tikzcd} \tag{4.59}$$

where *ap* was generalized to ϵ.

Generalizing section 4.12, adjunction (4.58) has the following properties: *reflection*:

$$\lceil \epsilon \rceil = id \tag{4.60}$$

that is, $\epsilon = \lfloor id \rfloor$, *cancellation*:

$$\lfloor id \rfloor \cdot \mathsf{G}\,\lceil f \rceil = f \tag{4.61}$$

fusion:

$$\lceil h \rceil \cdot g = \lceil h \cdot \mathsf{G}\,g \rceil \tag{4.62}$$

absorption:

$$(\mathsf{F}\,g) \cdot \lceil h \rceil = \lceil g \cdot h \rceil \tag{4.63}$$

naturality:

$$h \cdot \lfloor id \rfloor = \lfloor id \rfloor \cdot \mathsf{G}\,(\mathsf{F}\,h) \tag{4.64}$$

functor:

$$\mathsf{F}\,h = \lceil h \cdot \epsilon \rceil \tag{4.65}$$

closed definitions:

$$\lfloor k \rfloor = \lfloor id \rfloor \cdot (\mathsf{G}\,k) \tag{4.66}$$

$$\lceil g \rceil = (\mathsf{F}\,g) \cdot \lceil id \rceil \tag{4.67}$$

(The last follows from absorption.)

From an adjunction (4.58) a monad $\mathsf{T} = \mathsf{F} \cdot \mathsf{G}$ arises defined by $\eta = \lceil id \rceil$ and $\mu = \mathsf{F} \lfloor id \rfloor$. Finally, from all this we can infer the generic version of $f \bullet g$,

$$f \bullet g = \lceil \lfloor f \rfloor \cdot \lfloor g \rfloor \rceil \tag{4.68}$$

by replaying the calculation which lead to (4.44):

$$\begin{array}{cl}
 & f \bullet g \\
= & \qquad \{\ (4.5)\ \} \\
 & \mu \cdot \mathsf{T} f \cdot g \\
= & \qquad \{\ \mathsf{T} = \mathsf{F} \cdot \mathsf{G};\ \mu = \mathsf{F} \lfloor id \rfloor\ \} \\
 & \mathsf{F} \lfloor id \rfloor \cdot (\mathsf{F}\ (\mathsf{G} f)) \cdot g \\
= & \qquad \{\ \text{functor } \mathsf{F}\ \} \\
 & \mathsf{F}\ (\lfloor id \rfloor \cdot \mathsf{G} f) \cdot g \\
= & \qquad \{\ \text{cancellation: } \lfloor id \rfloor \cdot \mathsf{G} f = \lfloor f \rfloor ; g = \lceil \lfloor g \rfloor \rceil\ \} \\
 & \mathsf{F}\ \lfloor f \rfloor \cdot \lceil \lfloor g \rfloor \rceil \\
= & \qquad \{\ \text{absorption: } (\mathsf{F}\ g) \cdot \lceil h \rceil = \lceil g \cdot h \rceil\ \} \\
 & \lceil \lfloor f \rfloor \cdot \lfloor g \rfloor \rceil
\end{array}$$

Finally, let us see another example of a monad arising from one such adjunction (4.58). Recall exercise 2.27, on page 52, where pair / unpair witness an isomorphism similar to that of *curry*/*uncurry*, for $\mathsf{pair}\ (f,g) = \langle f,g \rangle$ and $\mathsf{unpair}\ k = (\pi_1 \cdot k, \pi_2 \cdot k)$. This can be cast into an adjunction as follows

$$\begin{array}{cl}
 & k = \mathsf{pair}\ (f,g)\ \Leftrightarrow\ (\pi_1 \cdot k, \pi_2 \cdot k) = (f,g) \\
\equiv & \qquad \{\ \text{see below}\ \} \\
 & k = \mathsf{pair}\ (f,g)\ \Leftrightarrow\ (\pi_1, \pi_2) \cdot (\mathsf{G}\ k) = (f,g)
\end{array}$$

where $\mathsf{G}\ k = (k,k)$. Note the abuse of notation, on the righthand side, of extending function composition notation to composition of *pairs* of functions, defined in the expected way: $(f,g) \cdot (h,k) = (f \cdot h, g \cdot k)$. Note that, for $f : A \to B$ and $g : C \to D$, the pair (f,g) has type $(A \to B) \times (C \to D)$. However, we shall abuse of notation again and declare the type $(f,g) : (A,C) \to (B,D)$.[8] In the opposite direction, $\mathsf{F}\ (f,g) = f \times g$:

$$\begin{tikzcd}
B \times A & & (B \times A, B \times A) \arrow[r, "(\pi_1,\pi_2)"] & (B,A) \\
C \arrow[u, "k=\mathsf{pair}\ (f,g)"] & & (C,C) \arrow[u, "(k,k)"] \arrow[ur, "(f,g)"'] &
\end{tikzcd} \tag{4.69}$$

[8] Strictly speaking, we are not abusing notation but rather working on a new *category*, that is, another mathematical system where functions and objects always come in pairs. For more on categories see the standard textbook [49].

This is but another way of writing the universal property of products (2.63), since $(f,g) = (h,k) \Leftrightarrow f = h \wedge g = k$ and $\mathsf{pair}\ (f,g) = \langle f,g \rangle$, recall exercise 2.27.

What is, then, the monad behind this *pairing* adjunction? It is the *pairing monad* $(\mathsf{F} \cdot \mathsf{G})\ X = \mathsf{F}\ (\mathsf{G}\ X) = \mathsf{F}\ (X,X) = X \times X$, where $\eta = \langle id, id \rangle$ and $\mu = \pi_1 \times \pi_2$. This monad allows us to work with pairs regarded as 2-dimensional *vectors* (y,x). For instance, the **do**-expression

$$\mathbf{do}\ \{x \leftarrow (2,3); y \leftarrow (4,5); \mathsf{return}\ (x+y)\}$$

yields $(6,8)$ as result in this monad — the *vectorial* sum of vectors $(2,3)$ and $(4,5)$. A simple encoding of this monad in Haskell is:

```
data P a = P (a,a) deriving Show
instance Functor P where
  fmap f (P (a,b)) = P (f a,f b)
instance Monad P where
  x >>= f = (μ · fmap f) x
  return a = P (a,a)
μ :: P (P a) → P a
μ (P (P (a,b),P (c,d))) = P (a,d)
```

***Exercise** 4.13. What is the vectorial operation expressed by the definition*

$$op\ k\ v = \mathbf{do}\ \{x \leftarrow v; \mathsf{return}\ (k \times x)\}$$

in the pairing monad?

□

4.13 ADJOINT CATAMORPHISMS

We conclude this chapter with a general result that links adjunctions (4.59) — and therefore monads — to the catamorphism (3.66) recursion pattern. A very simple example motivates what is to be achieved: the very simple implementation of addition of two numbers n and m,

$$\mathsf{add}\ (0,m) = m \tag{4.70}$$

$$\mathsf{add}\ (n+1,m) = 1 + \mathsf{add}\ (n,m) \tag{4.71}$$

is clearly defined by induction on the first parameter n and yet it is not a catamorphism.[9] So we cannot use catamorphism laws to reason about add, which is sad.

It turns out that add can be converted to a catamorphism once we *curry* it and go higher order, cf:

[9] Neither it is an anamorphism, although it is of course a hylomorphism.

$$
\begin{array}{l}
\overline{\mathsf{add}}\ 0\ m = m \\
\overline{\mathsf{add}}\ (n+1)\ m = 1 + \overline{\mathsf{add}}\ n\ m
\end{array}
$$

that is

$$
\begin{array}{l}
\overline{\mathsf{add}}\ 0 = id \\
\overline{\mathsf{add}}\ (n+1) = (1+) \cdot \overline{\mathsf{add}}\ n
\end{array}
$$

and finally

$$
\overline{\mathsf{add}} = \llparenthesis\, [\underline{id}, ((1+)\cdot)]\, \rrparenthesis \tag{4.72}
$$

Note the interplay between exponentials and catamorphisms. Can it be generalized?

Let adjunction $\mathsf{L} \dashv \mathsf{R}$ be given, where the choice of symbols clearly indicates which functor is the lower adjoint (L) and which is the upper adjoint (R). Let $\mathsf{T} \xleftarrow{\mathsf{in}} \mathsf{F}\ \mathsf{T}$ be an inductive type and $\phi : \mathsf{L}\ \mathsf{F} \to \mathsf{G}\ \mathsf{L}$ be a natural transformation, that is, *free theorem*

$$
\phi \cdot (\mathsf{L}\ \mathsf{F}\ k) = (\mathsf{G}\ \mathsf{L}\ k) \cdot \phi \tag{4.73}
$$

holds, for some functor G. Then:

$$
f \cdot (\mathsf{L}\ \mathsf{in}) = h \cdot \mathsf{G} f \cdot \phi \quad \Leftrightarrow \quad \lceil f \rceil = \llparenthesis\, \lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \rceil\, \rrparenthesis \tag{4.74}
$$

holds. Let us calculate this:

$$
\begin{array}{ll}
 & \lceil f \rceil = \llparenthesis\, \lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \rceil\, \rrparenthesis \\
\equiv & \qquad \{ \text{ cata-universal (3.66) } \} \\
 & \lceil f \rceil \cdot \mathsf{in} = \lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \rceil \cdot \mathsf{F}\ \lceil f \rceil \\
\equiv & \qquad \{ \text{ fusion (4.62) twice } \} \\
 & \lceil f \cdot \mathsf{L}\ \mathsf{in} \rceil = \lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \cdot \mathsf{L}\ \mathsf{F}\ \lceil f \rceil \rceil \\
\equiv & \qquad \{ \text{ isomorphism } \lceil \_ \rceil \text{ ; natural-}\phi \text{ (4.73) } \} \\
 & f \cdot \mathsf{L}\ \mathsf{in} = h \cdot \mathsf{G}\ \epsilon \cdot \mathsf{G}\ \mathsf{L}\ \lceil f \rceil \cdot \phi \\
\equiv & \qquad \{ \text{ functor G; cancellation } \epsilon \cdot \mathsf{L}\ \lceil f \rceil = f \text{ (4.61) } \} \\
 & f \cdot \mathsf{L}\ \mathsf{in} = h \cdot \mathsf{G} f \cdot \phi \\
\Box &
\end{array}
$$

As example of application, let us check how (4.72) arises from this result, where $\mathsf{in} = [\mathsf{zero}, \mathsf{succ}]$, $f = \mathsf{add}$, $\mathsf{F}\ X = 1 + X$ and $\mathsf{L}\ X = X \times A$.[10] Then $\mathsf{L}\ \mathsf{F}\ X = (1+X) \times A$. In general, $(1+X) \times A$ is isomorphic to $A + X \times A$, which suggests $\mathsf{G}\ X = A + X$. Thus one chooses

$$
\begin{array}{l}
\phi : (1+X) \times A \to A + (X \times A) \\
\phi = (\pi_2 + id) \cdot \mathsf{distl}
\end{array}
$$

[10] A is instantiated to $\mathbb{N}_0$ in the current example.

which is an isomorphism. The next step is to calculate $h = [h_1, h_2]$:

$$
\begin{array}{ll}
& \mathsf{add} \cdot ([\mathsf{zero}, \mathsf{succ}] \times id) = [h_1, h_2] \cdot (id + \mathsf{add}) \cdot (\pi_2 + id) \cdot \mathsf{distl} \\
\equiv & \quad \{ \ \mathsf{undistl} = [i_1 \times id, i_2 \times id] \ (2.60); +\text{-absorption} \ \} \\
& \mathsf{add} \cdot ([\mathsf{zero}, \mathsf{succ}] \times id) \cdot [i_1 \times id, i_2 \times id] = [h_1 \cdot \pi_2, h_2 \cdot \mathsf{add}] \\
\equiv & \quad \{ \ +\text{-fusion} ; \text{functor-}\times ; +\text{-eq} \ \} \\
& \left\{ \begin{array}{l} \mathsf{add} \cdot (\mathsf{zero} \times id) = h_1 \cdot \pi_2 \\ \mathsf{add} \cdot (\mathsf{succ} \times id) = h_2 \cdot \mathsf{add} \end{array} \right. \\
\equiv & \quad \{ \ \text{pattern-match with (4.71)} \ \} \\
& \left\{ \begin{array}{l} h_1 = id \\ h_2 = (1+) \end{array} \right.
\end{array}
$$

Then, by (4.74):

$$
\overline{\mathsf{add}} = (\!| \overline{[id, (1+)] \cdot (id + ap) \cdot (\pi_2 + id) \cdot \mathsf{distl}} |\!)
$$

The final step is to simplify the *gene* of catamorphism $\overline{\mathsf{add}}$:

$$
\begin{array}{ll}
& [f, g] = \overline{[id, (1+)] \cdot (id + ap) \cdot (\pi_2 + id) \cdot \mathsf{distl}} \\
\equiv & \quad \{ \ \text{(2.90) and (2.118)} \ \} \\
& [f, g] = \overline{[\pi_2, (1+) \cdot ap] \cdot ap} \cdot [\overline{i_1}, \overline{i_2}] \\
\equiv & \quad \{ \ \text{coproduct laws ; (2.90) twice} \ \} \\
& \left\{ \begin{array}{l} f = \overline{\pi_2} \\ g = \overline{(1+) \cdot ap} \end{array} \right. \\
\equiv & \quad \{ \ \text{(2.103) and (2.89)} \ \} \\
& \left\{ \begin{array}{l} f = \underline{id} \\ g = ((1+) \cdot) \end{array} \right. \\
\square &
\end{array}
$$

Thus we get $\overline{\mathsf{add}} = (\!| [\underline{id}, ((1+)\cdot)] |\!)$ (4.72), as expected.

Expressed by diagrams, it becomes clear that the lefthand side of (4.74) is the G-hylomorphism

$$\begin{tikzcd}
\mathsf{L}\ \mu\mathsf{F} \arrow[d, "f"'] & & \mathsf{G}\ \mathsf{L}\ \mu\mathsf{F} \arrow[d, "\mathsf{G} f"'] & \mathsf{L}\ \mathsf{F}\ \mu\mathsf{F} \arrow[l, "\phi"] \arrow[lll, bend right, "\mathsf{L}\ \mathsf{in}"'] \\
A & & \mathsf{G}\ A \arrow[ll, "h"'] &
\end{tikzcd}$$

$$
f = (\!| h |\!) \cdot [\!(\phi \cdot \mathsf{L}\ \mathsf{out})\!]
$$

which is converted into the adjoint F-catamorphism (righthand side):

$$\begin{tikzcd}
\mu_{\mathsf{F}} \arrow[d, "\lceil f \rceil"'] & & \mathsf{F}\ \mu_{\mathsf{F}} \arrow[ll, "\mathsf{in}"'] \arrow[d, "\mathsf{F}\ \lceil f \rceil"] \\
\mathsf{R}\ A & & \mathsf{F}\ \mathsf{R}\ A \arrow[ll, "\lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \rceil"']
\end{tikzcd}$$

$$\begin{tikzcd}
A & \mathsf{G}\ A \arrow[l, "h"'] & \mathsf{G}\ \mathsf{L}\ \mathsf{R}\ A \arrow[l, "\mathsf{G}\ \epsilon"'] & \mathsf{L}\ \mathsf{F}\ \mathsf{R}\ A \arrow[l, "\phi"']
\end{tikzcd}$$

$$\lceil f \rceil = \llparenthesis\, \lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \rceil \,\rrparenthesis$$

ACCUMULATIONS The adjoint catamorphism construction (4.74) explains a number of useful programming techniques. For instance, mutual recursion (recall section 3.17) can be shown to derive from (4.74) via the *pairing* adjunction (4.69).

Another useful outcome of the same result is the optimization technique known as *adding accummulation parameters* to convert quadratic-time to linear-time programs.[11] This arises as a particular situation in cata-fusion (3.70)

$$\begin{tikzcd}
\mathsf{T} \arrow[d, "\llparenthesis f \rrparenthesis"'] & \mathsf{F}\, \mathsf{T} \arrow[l, "\mathsf{in}"'] \arrow[d, "\mathsf{F}\, \llparenthesis f \rrparenthesis"] \\
A \arrow[d, "k"'] & \mathsf{F}\, A \arrow[l, "f"] \arrow[d, "\mathsf{F}\, k"] \\
A^A & \mathsf{F}\ A^A \arrow[l, "g"]
\end{tikzcd}$$

where $k : A \rightarrow A^A$ such that $k\ a\ e = a$, that is, e is the unit of k. Then $\llparenthesis f \rrparenthesis\ x = \llparenthesis g \rrparenthesis\ x\ e = \widehat{\llparenthesis g \rrparenthesis}\ (x, e)$, cf:

$$\begin{array}{ll}
& k \cdot \llparenthesis f \rrparenthesis = \llparenthesis g \rrparenthesis \\
\equiv & \qquad \{ \text{ go pointwise } \} \\
& k\ (\llparenthesis f \rrparenthesis\ x) = \llparenthesis g \rrparenthesis\ x \\
\Rightarrow & \qquad \{ \text{ add unit element } \} \\
& k\ (\llparenthesis f \rrparenthesis\ x)\ e = \llparenthesis g \rrparenthesis\ x\ e \\
\equiv & \qquad \{ \ k\ a\ e = a; \text{ use uncurried notation } \} \\
& \llparenthesis f \rrparenthesis\ x = \widehat{\llparenthesis g \rrparenthesis}\ (x, e)
\end{array}$$

The second parameter of $\widehat{\llparenthesis g \rrparenthesis}$ is called an *accumulation* parameter. Let us see what the proviso of cata-fusion means in this case:

$$k \cdot f = g \cdot \mathsf{F}\ k$$

[11] The so-called *structural recursion theorem* of [12] is a special case of (4.74). See also footnote 1 of chaper 8 later on.

$$
\begin{array}{ll}
\equiv & \quad \{ \text{ (4.74) letting } k = \overline{m} \text{ and } g = \overline{h \cdot \mathsf{G}\, ap \cdot \phi} \text{ for some } \mathsf{G} \text{ etc } \} \\
 & \overline{m \cdot \mathsf{L} f} = \overline{h \cdot \mathsf{G}\, ap \cdot \phi \cdot \mathsf{L}\, \mathsf{F}\, \overline{m}} \\
\equiv & \quad \{ \text{ curry isomorphism } \} \\
 & m \cdot \mathsf{L} f = h \cdot \mathsf{G}\, ap \cdot \mathsf{G}\, \mathsf{L}\, \overline{m} \cdot \phi \\
\equiv & \quad \{ \text{ functor } \mathsf{G}; ap \cdot \mathsf{L}\, m = m \ \} \\
 & m \cdot \mathsf{L} f = h \cdot \mathsf{G}\, m \cdot \phi
\end{array}
$$

The last equality is a sufficient condition for $\llparenthesis g \rrparenthesis$ be calculated such that $\llparenthesis f \rrparenthesis\ x$ is optimized to $\llparenthesis g \rrparenthesis\ x\ e$.

4.14 BIBLIOGRAPHY NOTES

The use of monads in computer science started with Moggi [58], who had the idea that monads should supply the extra semantic information needed to implement the lambda-calculus theory. Haskell [43] is among the computer languages which make systematic use of monads for implementing effects and imperative constructs in a purely functional style.

Category theorists invented monads in the 1960's to concisely express certain aspects of universal algebra. Functional programmers invented list comprehensions in the 1970's to concisely express certain programs involving lists. Philip Wadler [85] made a great contribution to the field by showing that list comprehensions could be generalised to arbitrary monads and unify with imperative "do"-notation in case of the monad which explains imperative computations.

Monads are nowadays an essential feature of functional programming and are used in fields as diverse as language parsing [36], component-oriented programming [9], strategic programming [45], multimedia [35] and probabilistic programming [20]. Adjunctions play a major role in [32], which gives a full account of adjoint recursive patterns.

Part II

CALCULATING WITH RELATIONS

5

WHEN EVERYTHING BECOMES A RELATION

In the previous chapters, (recursive) functions were taken as a basis for expressing computations, exhibiting powerful laws for calculating programs in a functional programming style.

When writing such programs one of course follows some line of thought concerning *what* the programs should do. *What the program should do* is usually understood as the *specification* of the problem that motivates writing the program in the first place. Specifications can be quite complex in real life situations. In other situations, the complexity of the program that one writes is in strong contrast with the simplicity of the specification. Take the example of sorting, which can be specified as simply as:

> *Yield an ordered permutation of the input.*

Where do you find, in this specification, the orientation (or inspiration) that will guide a programmer towards writing a bi-recursive program like *quicksort*?

The question is, then: are functions *enough* for one to calculate functional programs from given specifications? It is the experience in other fields of mathematics that sometimes it is easier to solve a problem of domain D if one generalizes from D to some wider domain D'. In the field of real numbers, for instance, most of trigonometric identities are easily derived (and memorized) from Euler's formula involving complex exponentials: $e^{i\,x} = cos\ x + i\ (sin\ x)$.

Similarly, it turns out that functional programs often become easier to calculate if one handles them in the wider mathematical domain of binary relations. At school one gets accustomed to the sentence *every function is a special case of a relation*. This chapter puts the usefulness of such a piece of common knowledge into practice.

5.1 FUNCTIONS ARE NOT ENOUGH

Consider the following fragment of a requirement posed by a (fictional) telecommunication company:

> *(...) For each* list of calls *stored in the mobile phone (e.g. numbers dialed, SMS messages, lost calls), the* store *operation should work in a way such that (a) the more recently a* call *is made the more accessible*

> *it is; (b) no number appears twice in a list; (c) only the last 10 entries in each list are stored.*

A tentative, first implementation of the *store* operation could be

$$
\begin{array}{l}
\mathit{store} : \mathit{Call} \rightarrow \mathit{Call}^* \rightarrow \mathit{Call}^* \\
\mathit{store}\ c\ l = c : l
\end{array}
$$

However, such a version of function *store* fails to preserve the *properties* required in the fragment above in case $\mathsf{length}\ l = 10$, or $c \in \mathsf{elems}\ l$, where elems yields the set of all elements of a finite list,

$$\mathsf{elems} = (\!|\, [\mathit{empty}, \mathsf{join}]\, |\!) \tag{5.1}$$

for $\mathit{empty}\ \_ = \{\,\}$ and $\mathsf{join}\ (a,s) = \{a\} \cup s$.

Clearly, the designer would have to *restrict* the application of *store* to input values c, l such that the given properties are preserved. This could be achieved by adding a so-called *"pre-condition"*:

$$
\begin{array}{l}
\mathit{store} : \mathit{Call} \rightarrow \mathit{Call}^* \rightarrow \mathit{Call}^* \\
\mathit{store}\ c\ l = c : l \\
\mathsf{pre\ length}\ l < 10 \wedge \neg\ (c \in \mathsf{elems}\ l)
\end{array}
$$

Such a pre-condition is a predicate telling the range of *acceptable* input values, to be read as a *warning* provided by the designer that the function will not meet the requirements outside such a range of input values.

Thus *store* becomes a *partial function*, that is, a function defined only for some of its inputs. Although this partiality can be regarded as a symptom that the requirements have been partly misunderstood, it turns out that *partial* functions are the rule rather than the exception in mathematics and computing. For example, in the numeric field, we know what $1/2$ means; what about $1/0$? Ruling out this case means that *division* is a partial function. In list processing, given a sequence s, what does $s\ !!\ i$ mean in case $i > \mathsf{length}\ s$? — list indexing is another *partial* operation (as are head, tail and so on).

Partial functions are not new to readers of this text: in section 4.1, the *Maybe* monad was used to "totalize" partial functions. In this chapter we shall adopt another strategy to cope with partiality, and one that has extra merits: it will also cope with computational nondeterminacy and vagueness of software requirements.

It can be shown that the following evolution of *store*,

$$\mathit{store}\ c = (\mathsf{take}\ 10) \cdot (c{:}) \cdot \mathsf{filter}\ (c \neq) \tag{5.2}$$

meets all requirements above with no need for preconditions, the extra components $\mathsf{take}\ 10$ and $\mathsf{filter}\ (c \neq)$ being added to comply with requirements (c) and (b), respectively.

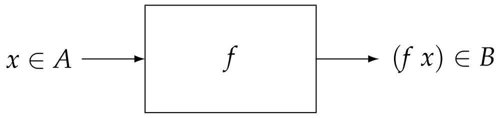
Implementation (5.2) alone should be regarded as example of how functional programs can be built compositionally in a requirement-driven fashion. It does not, however, give any guarantees that the requirements are *indeed* met. How can we ensure this in the compositional way advocated in this book since its beginning? The main purpose of this chapter is to answer such a question.

5.2 FROM FUNCTIONS TO RELATIONS

The way functions are handled and expressed in standard maths books, e.g. in analysis and calculus,

$$y = f\ (x)$$

is indicative that, more important that the *reactive* behaviour of f,



which was the starting point of section 2.1, mathematicians are more interested in expressing the input/output *relationship* of f, that is, the set of all pairs (y, x) such that $y = f\ x$. Such a set of pairs is often referred to as the "graph" of f, which can be plotted two-dimensionally in case types A and B are linearly ordered. (As is the standard case in which A=B=$\mathbb{R}$, the real numbers.)

It turns out that such a *graph* can be regarded as a special case of a *binary relation*. Take for instance the following functional declaration

$$\left\{\begin{array}{l} \mathsf{succ} : \mathbb{N}_0 \to \mathbb{N}_0 \\ \mathsf{succ}\ x = x + 1 \end{array}\right.$$

which expresses the computation rule of the *successor* function. Writing $y = \mathsf{succ}\ n$ establishes the *binary relation* $y = x + 1$. This binary relation "coincides" with succ in the sense that writing

$$\left\{\begin{array}{l} \mathsf{succ} : \mathbb{N}_0 \to \mathbb{N}_0 \\ y\ \mathsf{succ}\ x \ \Leftrightarrow\ y = x + 1 \end{array}\right.$$

means the same as the original definition, while making the i/o relationship explicit. Because there is *only one* y such that $y = x + 1$ we can safely drop both ys from $y\ \mathsf{succ}\ x \ \Leftrightarrow\ y = x + 1$, obtaining the original $\mathsf{succ}\ x = x + 1$.

The new style is, however, more expressive, in the sense that it enables us to declare *genuine* binary relations, for instance

$$\left\{\begin{array}{l} R : \mathbb{N}_0 \to \mathbb{N}_0 \\ y\ R\ x \ \Leftrightarrow\ y \geqslant x + 1 \end{array}\right. \tag{5.3}$$

In this case, not only x and y such that $y = x + 1$ are admissible, but also $y = x + 2$, $y = x + 3$ and so on. It also enables us to express the *converse* of any function — an operation hitherto the privilege of isomorphisms only (2.16):

$$y\ f\ x \ \Leftrightarrow\ x\ f^{\circ}\ y \tag{5.4}$$

Converses of functions are very useful in problem solving, as we shall soon see. For instance, $\mathbb{N}_0 \xleftarrow{\mathsf{succ}^\circ} \mathbb{N}_0$ denotes the *predecessor* relation in $\mathbb{N}_0$. It is not a function because no y such that $y\ \mathsf{succ}^\circ\ 0$ exists — try and solve $0 = y + 1$ in $\mathbb{N}_0$.

The intuitions above should suffice for us to start generalizing what we know about functions, from the preceding chapters, to binary relations. First of all, such relations are denoted by *arrows* exactly in the same way functions are. So,

> we shall write $R : B \leftarrow A$, $R : A \rightarrow B$, $B \xleftarrow{R} A$ or $A \xrightarrow{R} B$ to indicate that relation R relates B-values to A-values.

That is, relations are typed in the same way as functions.

Given binary relation $R : B \leftarrow A$, writing $b\ R\ a$ (read: "b is related to a by R") means the same as $a\ R^\circ\ b$, where R° is said to be the *converse* of R. In terms of grammar, R° corresponds to the *passive voice* — compare e.g.

$$\underbrace{\textit{John}}_{b}\ \underbrace{\textit{loves}}_{R}\ \underbrace{\textit{Mary}}_{a}$$

with

$$\textit{Mary}\ \underbrace{\textit{is loved by}}_{R^\circ}\ \textit{John}$$

That is, $(\textit{loves})^\circ = (\textit{is loved by})$. Another example:

> *Catherine eats the apple*

— $R = (\textit{eats})$, active voice — compared with

> *the apple is eaten by Catherine*

— $R^\circ = (\textit{is eaten by})$, passive voice.

Following a widespread convention, functions are denoted by lowercase characters (e.g. f, g, ϕ) or identifiers starting with a lowercase characters, while uppercase letters are reserved to arbitrary relations. In the case of functions ($R := f$), $b\ f\ a$ means exaclty $b = f\ a$. This is because functions are *univocal*, that is, no two different b and b' are such that $b\ f\ a \wedge b'\ f\ a$. In fact, the following facts hold about *any* function f:

- *Univocality* (or "left" uniqueness) —

$$b\ f\ a \wedge b'\ f\ a \quad \Rightarrow \quad b = b' \tag{5.5}$$

- *Leibniz principle* —

$$a = a' \quad \Rightarrow \quad f\ a = f\ a' \tag{5.6}$$

Clearly, not every relation obeys (5.5), for instance

$$2 < 3 \land 1 < 3 \quad \nRightarrow \quad 2 = 1$$

Relations obeying (5.5) will be referred to as *simple*, according to a terminology to follow shortly.

Implication (5.6) expresses the (philosophically) interesting fact that no function (observation) can be found able to distinguish between two equal objects. This is another fact true about functions which does not generalize to binary relations, as we shall see when we come back to this later.

Recapitulating: we regard *function* $f : A \longrightarrow B$ as the binary relation which relates b to a iff $b = f\ a$. So,

$$b\ f\ a \text{ literally means } b = f\ a \tag{5.7}$$

The purpose of this chapter is to generalize from

$$\boxed{\begin{array}{c} B \xleftarrow{f} A \\ b = f\ a \end{array}} \quad \text{to} \quad \boxed{\begin{array}{c} B \xleftarrow{R} A \\ b\ R\ a \end{array}}$$

5.3 PRE/POST CONDITIONS

It should be noted that relations are used in virtually every body of science and it is hard to think of another way to express human knowledge in philosophy, epistemology and common life, as suggestively illustrated in figure 5.1. This figure is also illustrative of another popular ingredient when using relations — the *arrows* drawn to denote relationships.[1]

In real life, "everything appears to be a relation". This has lead software theorists to invent linguistic layouts for relational specification, leading to so-called *specification languages*. One such language, today historically relevant, is the language of the Vienna Development Method (VDM). In this notation, the relation described in (5.3) will be written:

$$\begin{array}{l} R\ (x : \mathbb{N}_0)\ y : \mathbb{N}_0 \\ \mathsf{post}\ y \geqslant x + 1 \end{array}$$

where the clause prefixed by post is said to be a post-condition. The format also includes pre-conditions, if necessary. Such is the case of the following pre / post -styled specification of the operation that extracts an arbitrary element from a set:

$$\begin{array}{l} Pick\ (x : \mathsf{P}A)\ (r : A, y : \mathsf{P}A) \\ \mathsf{pre}\ x \neq \{\ \} \\ \mathsf{post}\ r \in x \land y = x - \{r\} \end{array} \tag{5.8}$$

[1] Our extensive use of arrows to denote relations in the sequel is therefore rooted on common, informal practice. Unfortunately, mathematicians do not follow such practice and insist on regarding relations just as sets of pairs.

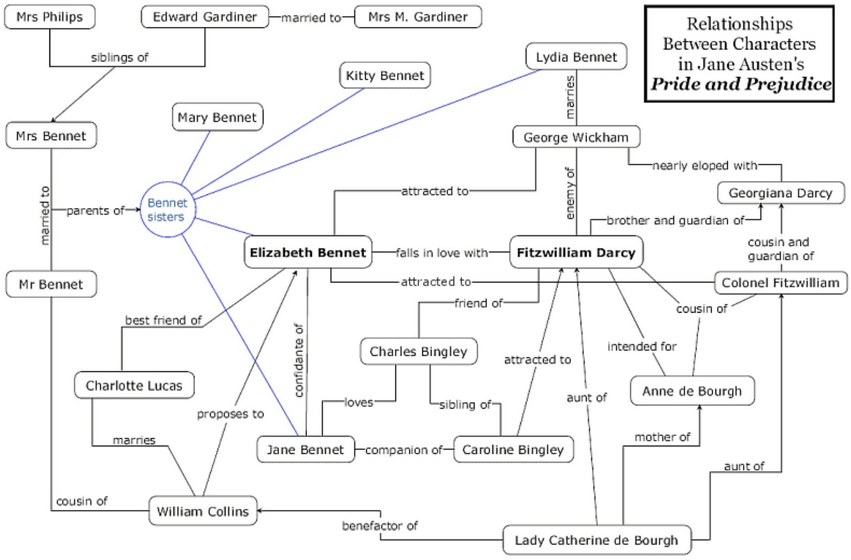


Figure 5.1.: Personal relationships among the main characters of the novel *Pride and Prejudice*, by Jane Austin, 1813. (Source: Wikipedia)

Here $\mathsf{P}A = \{X \mid X \subseteq A\}$ is the set of all subsets of A. Mapping this back to the relational format of (5.3), *Pick* is the relation defined by:

$$\left\{ \begin{array}{l} Pick : \mathsf{P}A \to (A \times \mathsf{P}A) \\ (r,y)\ Pick\ x \ \Leftrightarrow\ x \neq \{\,\} \wedge r \in x \wedge y = x - \{r\} \end{array} \right.$$

Note how $(r,y)\ Pick\ \{\,\} \ \Leftrightarrow\ \mathsf{False}$ for whatever r, y. Here follows the specification of *sorting* written in the pre / post -style,

$$\begin{array}{l} Sort\ (x : A^*)\ y : A^* \\ \mathsf{post}\ (ord\ y) \wedge bag\ y = bag\ x \end{array} \tag{5.9}$$

where *ord* is the predicate defined in section 3.17 and *bag* is the function that extracts the multiset of elements of a finite list.[2] Note how *Sort* defines sorting independently of giving an explicit algorithm. In fact, the pre / post -style provides a way of *hiding* the algorithmic details that any particular functional implementation is bound to include.

Wherever a post-condition is intended to specify a function f, one refers to such a condition as an *implicit specification* of f. Examples: *explicit* definition of the *abs* function

$$\begin{array}{l} abs : \mathbb{Z} \to \mathbb{Z} \\ abs\ i = \mathbf{if}\ i < 0\ \mathbf{then}\ -i\ \mathbf{else}\ i \end{array}$$

followed by an *implicit specification* of the same function:

[2] Recall that *ord* assumes an ordering on type A. For further developments on this specification see exercise 5.17 later on.

$$\begin{array}{l} abs\ (i : \mathbb{Z})\ r : \mathbb{Z} \\ \mathsf{post}\ r \geqslant 0 \wedge (r = i \vee r = -\,i) \end{array}$$

Explicit definition of *max* function:

$$\begin{array}{l} max : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\ max\ (i,j) = \mathbf{if}\ i \leqslant j\ \mathbf{then}\ j\ \mathbf{else}\ i \end{array}$$

Its *implicit specification*:

$$\begin{array}{l} max\ (i : \mathbb{Z}, j : \mathbb{Z})\ r : \mathbb{Z} \\ \mathsf{post}\ r \in \{i,j\} \wedge i \leqslant r \wedge j \leqslant r \end{array}$$

Of a different nature is the following *pre/post*-pair:

$$\begin{array}{l} Sqrt : (i : \mathbb{R})\ r : \mathbb{R} \\ \mathsf{post}\ r^2 = i \end{array}$$

Here the *specifier* is telling the *implementer* that either solution $r = +\sqrt{i}$ or $r = -\sqrt{i}$ will do.[3] Indeed, *square root* is not a function, it is the binary relation:

$$r\ Sqrt\ i \ \Leftrightarrow\ r^2 = i \tag{5.10}$$

We proceed with a thorough study of the concept of a binary relation, by analogy with a similar study carried out about functions in chapter 2.

5.4 RELATIONAL COMPOSITION AND CONVERSE

Such as functions, relations can be combined via composition ($R \cdot S$), defined as follows:

$$\begin{tikzcd} B & A \arrow[l, "R"'] & C \arrow[l, "S"'] \arrow[ll, bend left, "R\cdot S"] \end{tikzcd} \quad b(R \cdot S)c \ \equiv\ \langle \exists\ a\ :\ b\ R\ a\ :\ a\ S\ c \rangle \tag{5.11}$$

Example: $Uncle = Brother \cdot Parent$, expanding to

$$u\ Uncle\ c \ \equiv\ \langle \exists\ p\ ::\ u\ Brother\ p \wedge p\ Parent\ c \rangle$$

An explanation on the $\exists$-notation is on demand: $\exists$ is an instance of a so-called *quantifier*, a main ingredient of formal logic. In this book we follow the so-called *Eindhoven quantifier* notation, whereby expressions of the form

$$\langle \forall\ x\ :\ P\ :\ Q \rangle$$

mean

*"for **all** x in the range P, Q holds"*

[3] This aspect of formal specification is called *vagueness*.

where P and Q are logical expressions involving x; and expressions of the form

$$\langle \exists\, x \;:\; P :\; Q \rangle$$

mean

*"for **some** x in the range P, Q holds".*

Note how the symbols $\exists$ and $\forall$ "flip" letters E (exists) and A (all), respectively. P is known as the *range* of the quantification and Q as the quantified *term*.[4] This logical notation enjoys a well-known set of properties, some of which are given in appendix A.2. As an example, by application of the $\exists$-trading rule (A.2), predicate $\langle \exists\, a \;:\; b\; R\; a :\; a\; S\; c \rangle$ in (5.11) can be written $\langle \exists\, a \;::\; b\; R\; a \wedge a\; S\; c \rangle$.

Note how (5.11) *removes* $\exists$ and bound variable a when applied from right to left. This is an example of conversion from pointwise to pointfree notation, since "point" a also disappears. Indeed, we shall try and avoid lengthy, complex $\forall, \exists$-formulae by converting them to *pointfree* notation, as is the case in (5.11) once relational composition is used.

A simple calculation shows (5.11) to instantiate to (2.6) for the special case where R and S are functions, $R, S := f, g$:

$$\begin{array}{cl}
b(f \cdot g)c & \equiv \;\; \langle \exists\, a \;::\; b\, f\; a \wedge a\; g\; c \rangle \\
\equiv & \qquad \{ \text{ functions are univocal (simple) relations } \} \\
 & \langle \exists\, a \;::\; b = f\; a \wedge a = g\; c \rangle \\
\equiv & \qquad \{ \; \exists\text{-trading rule (A.2) } \} \\
 & \langle \exists\, a \;:\; a = g\; c :\; b = f\; a \rangle \\
\equiv & \qquad \{ \; \exists\text{-``one-point'' rule (A.6) } \} \\
 & b = f\; (g\; c) \\
\Box &
\end{array}$$

Like its functional version (2.8), relation composition is associative:

$$R \cdot (S \cdot P) = (R \cdot S) \cdot P \tag{5.12}$$

Everywhere $T = R \cdot S$ holds, the replacement of T by $R \cdot S$ will be referred to as a "factorization" and that of $R \cdot S$ by T as "fusion". Every relation $B \xleftarrow{R} A$ admits two trivial factorizations,

$$\left\{ \begin{array}{l} R = R \cdot id_A \\ R = id_B \cdot R \end{array} \right. \tag{5.13}$$

where, for every X, id_X is the identity relation relating every element of X with itself (2.9). In other words: the identity (equality) *relation* coincides with the identity *function*.

[4] In particular, Q or P can be universally False or True. Assertions of the form $\langle \forall\, x \;:\; \mathsf{True} :\; Q \rangle$ or $\langle \exists\, x \;:\; \mathsf{True} :\; Q \rangle$ are abbreviated to $\langle \forall\, x \;::\; Q \rangle$ or $\langle \exists\, x \;::\; Q \rangle$, respectively.

In section 2.7 we introduced a very special case of function f — isomorphism — which has a converse f° such that (2.16) holds. A major advantage of generalizing functions to relations is that *every* relation $A \xrightarrow{R} B$ has a converse $A \xleftarrow{R^\circ} B$ defined by

$$b\,R\,a \quad \Leftrightarrow \quad a\,R^\circ\,b \tag{5.14}$$

— the *passive voice* written relationally, as already mentioned. Two important properties of converse follow: it is an involution

$$(R^\circ)^\circ = R \tag{5.15}$$

and it commutes with composition in a contravariant way:

$$(R \cdot S)^\circ = S^\circ \cdot R^\circ \tag{5.16}$$

Converses of functions enjoy a number of properties from which the following is singled out as a way to introduce / remove them from logical expressions:

$$b(f^\circ \cdot R \cdot g)a \quad \equiv \quad (f\ b)R(g\ a) \tag{5.17}$$

For instance, the consequent of implication (5.6) could have been written $a(f^\circ \cdot id \cdot f)a'$, or even simpler as $a(f^\circ \cdot f)a'$, as it takes very little effort to show:

$$\begin{array}{cl}
& a(f^\circ \cdot id \cdot f)a' \\
\equiv & \quad \{\ (5.17)\ \} \\
& (f\ a)id(f\ a') \\
\equiv & \quad \{\ b\ f\ a \equiv b = f\ a\ \} \\
& (f\ a) = id(f\ a') \\
\equiv & \quad \{\ (2.9)\ \} \\
& f\ a = f\ a' \\
\square &
\end{array}$$

***Exercise** 5.1. Let sq $x = x^2$ be the function that computes the square of a real number. Use (5.17) to show that (5.10) reduces to*

$$Sqrt = sq^\circ$$

in relational pointfree notation.

□

***Exercise** 5.2. Give an implicit definition of function $f\ x = x^2 - 1$ in the form of a post-condition not involving subtraction. Then re-write it without variables using (5.17).*

□

5.5 RELATIONAL EQUALITY

Recall that function equality (2.5) is established by extensionality:

$$f = g \quad \textit{iff} \quad \langle \forall\, a \;:\; a \in A : f\ a = g\ a \rangle$$

Also recall that $f = g$ only makes sense iff both functions have the same type, say $A \to B$. Can we do the same for relations? The relational generalization of (2.5) will be

$$R = S \quad \textit{iff} \quad \langle \forall\, a, b \;:\; a \in A \wedge b \in B : \ b\ R\ a \ \Leftrightarrow\ b\ S\ a \rangle \tag{5.18}$$

Since $\Leftrightarrow$ is bi-implication, we can replace the term of the quantification by

$$(b\ R\ a \ \Rightarrow\ b\ S\ a) \quad \wedge \quad (b\ S\ a \Rightarrow b\ R\ a)$$

Now, what does $b\ R\ a \ \Rightarrow\ b\ S\ a$ mean? It simply captures relational *inclusion*,

$$R \subseteq S \quad \textit{iff} \quad \langle \forall\, a, b \;::\; b\ R\ a \ \Rightarrow\ b\ S\ a \rangle \tag{5.19}$$

whose righthand side can also be written

$$\langle \forall\, a, b \;:\; b\ R\ a : \ b\ S\ a \rangle$$

by $\forall$-trading (A.1). Note the same pointwise-pointfree move when one reads (5.19) from right to left: $\forall$, a and b disappear.

Altogether, (5.18) can be written in less symbols as follows:

$$R = S \quad \equiv \quad R \subseteq S \wedge S \subseteq R \tag{5.20}$$

This way of establishing relational equality is usually referred to as *circular inclusion*. Note that relational inclusion (5.19) is a partial order: it is *reflexive*, since

$$R \subseteq R \tag{5.21}$$

holds for every R; it is *transitive*, since for all R, S, T

$$R \subseteq S \wedge S \subseteq T \quad \Rightarrow \quad R \subseteq T \tag{5.22}$$

holds; and it is *antisymmetric*, as established by circular-inclusion (5.20) itself. Circular-inclusion is also jocosely known as the "ping-pong" method for establishing $R = S$: first calculate $R \subseteq S$ ("ping") and then $S \subseteq R$ ("pong"). This can be performed in one go by adopting the following calculation layout:

$$\begin{array}{ll} R & \subseteq \ \ldots \\ & \subseteq \ S \\ & \subseteq \ \ldots \\ & \subseteq \ R \\ & \Box \end{array}$$

This has the advantage of making apparent that not only R and S are the same, but also that every two steps in the circular reasoning are so (just choose a different start and stop point in the "circle").

Circular inclusion (5.20) is not the only way to establish relational equality. A less obvious, but very useful way of calculating the equality of two relations is the method of *indirect equality*:

$$\begin{aligned} R = S \quad &\equiv \quad \langle \forall\, X \;::\; (X \;\subseteq\; R \;\Leftrightarrow\; X \;\subseteq\; S)\rangle && (5.23)\\ &\equiv \quad \langle \forall\, X \;::\; (R \;\subseteq\; X \;\Leftrightarrow\; S \;\subseteq\; X)\rangle && (5.24)\end{aligned}$$

The reader unaware of this way of indirectly setting algebraic equalities will recognize that the same pattern of indirection is used when establishing set equality via the membership relation, cf.

$$A = B \;\equiv\; \langle \forall\, x \;::\; x \in A \;\Leftrightarrow\; x \in B\rangle$$

The typical layout of using any of these rules is the following:

$$\left\{\begin{array}{ll} & X \;\subseteq\; R \\ \equiv & \qquad \{\ \ldots\ \} \\ & X \;\subseteq \ldots \\ \equiv & \qquad \{\ \ldots\ \} \\ & X \;\subseteq\; S \\ :: & \qquad \{\ \text{indirect equality (5.23)}\ \} \\ & R = S \\ \Box & \end{array}\right.$$

This proof method is very powerful and we shall make extensive use of it in the sequel. (The curious reader can have a quick look at section 5.9 for a simple illustration.)

RELATIONAL TYPES. From this point onwards we shall regard the type $B \leftarrow A$ as including not only all functions $f : A \rightarrow B$ but also all relations of the same type, $R : A \rightarrow B$. This is far more than we had before! In particular, type $A \rightarrow B$ includes:

- the *bottom* relation $B \xleftarrow{\;\perp\;} A$, which is such that, for all b, a,

  $$b \bot a \;\equiv\; \text{FALSE}$$

- the *topmost* relation $B \xleftarrow{\;\top\;} A$, which is such that, for all b, a,

  $$b \top a \;\equiv\; \text{TRUE}$$

The former is referred to as the void, or *empty* relation. The latter is known as the universal, or *coexistence* relation. Clearly, for every R,

$$\bot \;\subseteq\; R \;\subseteq\; \top \tag{5.25}$$

and

$$R \cdot \bot = \bot \cdot R = \bot \tag{5.26}$$

hold. By (5.25) and (5.20), writing $R = \bot$ (respectively, $R = \top$) is the same as writing $R \subseteq \bot$ (respectively, $\top \subseteq R$).

A relation $B \xleftarrow{V} A$ is said to be a *vector* if either A or B are the singleton type 1. Relation $1 \xleftarrow{X} A$ is said to be a *row*-vector; clearly, $X \subseteq\, !$. Relation $B \xleftarrow{Z} 1$ is said to be a *column*-vector; clearly, $Z \subseteq\, !^\circ$. [5] A relation of type $1 \xleftarrow{S} 1$ is called a *scalar*.

Last but not least, note that in a relational setting types $B \leftarrow A$ and B^A do not coincide — B^A is the type of all *functions* from A to B, while $B \leftarrow A$ is the type of all *relations* from A to B. Clearly, $B^A \subseteq B \leftarrow A$.

5.6 DIAGRAMS

As happens with functions, the arrow notation adopted for functions makes it possible to express relational formulæ using diagrams. This is a major ingredient of the relational method because it provides a graphical way of picturing relation types and relational constraints.

Paths in diagrams are built by arrow chaining, which corresponds to relational composition $R \cdot S$ (5.11), meaning *"... is R of some S of ..."* in natural language.

Assertions of the form $X \subseteq Y$ where X and Y are relation compositions can be represented graphically by rectangle-shaped diagrams, as is the case in

$$\begin{tikzcd}
\mathit{Descriptor} \arrow[d, "path"'] & \mathit{Handle} \arrow[l, "FT"'] \arrow[d, "\top"] \arrow[dl, phantom, "\subseteq"] \\
\mathit{Path} & \mathit{File} \arrow[l, "FS^\circ"]
\end{tikzcd} \tag{5.27}$$

in the context of modelling a file-system. Relation FS models a *file store* (a table mapping file system paths to the respective files), FT is the *open-file descriptor* table (holding the information about the files that are currently open[6]), function *path* yields the path of a file descriptor and $\top$ is the largest possible relation between file-handles and files, as seen above. The diagram depicts the constraint:

$$path \cdot FT \subseteq FS^\circ \cdot \top \tag{5.28}$$

What does (5.28) mean, then, in predicate logic?

[5] The column and row qualifiers have to do with an analogy with vectors in linear algebra.

[6] Open files are manipulated by the file system via open file descriptor data structures, which hold various relevant metadata (e.g. current position within the file). Such descriptors are identified by file handles which the file system provides to applications that manipulate files. This indirection layer avoids unnecessary coupling between applications and the details of the file system implementation.

FROM DIAGRAMS TO LOGIC. We reason, using definitions (5.19,5.11) and the laws of the predicate calculus given in appendix A.2:

$$\begin{array}{ll}
& path \cdot FT \subseteq FS^\circ \cdot \top \\
\equiv & \quad \{ \text{ `at most' ordering (5.19) } \} \\
& \langle \forall\, p,h \;:\; p(path \cdot FT)h :\; p(FS^\circ \cdot \top)h \rangle \\
\equiv & \quad \{ \text{ composition (5.11) ; } path \text{ is a function } \} \\
& \langle \forall\, p,h \;:\; \langle \exists\, d \;:\; p = path\, d :\; d\, FT\, h \rangle :\; p(FS^\circ \cdot \top)h \rangle \\
\equiv & \quad \{ \text{ quantifier calculus — } \exists\text{-trading (A.2) } \} \\
& \langle \forall\, p,h \;:\; \langle \exists\, d \;:\; d\, FT\, h :\; p = path\, d \rangle :\; p(FS^\circ \cdot \top)h \rangle \\
\equiv & \quad \{ \text{ quantifier calculus — } \forall\text{-nesting (A.7) } \} \\
& \langle \forall\, h \;::\; \langle \forall\, p \;:\; \langle \exists\, d \;:\; d\, FT\, h :\; p = path\, d \rangle :\; p\; (FS^\circ \cdot \top)\; h \rangle \rangle \\
\equiv & \quad \{ \text{ quantifier calculus — splitting rule (A.13) } \} \\
& \langle \forall\, h \;::\; \langle \forall\, d \;:\; d\, FT\, h :\; \langle \forall\, p \;:\; p = path\, d :\; p\; (FS^\circ \cdot \top)\; h \rangle \rangle \rangle \\
\equiv & \quad \{ \text{ quantifier calculus — } \forall\text{-nesting (A.7) } \} \\
& \langle \forall\, d,h \;:\; d\, FT\, h :\; \langle \forall\, p \;:\; p = path\, d :\; p(FS^\circ \cdot \top)h \rangle \rangle \\
\equiv & \quad \{ \text{ quantifier calculus — } \forall\text{-one-point rule (A.5) } \} \\
& \langle \forall\, d,h \;:\; d\, FT\, h :\; (path\, d)(FS^\circ \cdot \top)h \rangle
\end{array}$$

We still have to unfold term $(path\ d)(FS^\circ \cdot \top)h$:

$$\begin{array}{ll}
& (path\, d)(FS^\circ \cdot \top)h \\
\equiv & \quad \{ \text{ composition (5.11) } \} \\
& \langle \exists\, x \;::\; (path\, d)FS^\circ x \;\wedge\; x \top h \rangle \\
\equiv & \quad \{ \text{ converse ; } x \top h \text{ always holds } \} \\
& \langle \exists\, x \;::\; x\, FS\; (path\, d) \wedge \text{TRUE} \rangle \\
\equiv & \quad \{ \text{ trivia } \} \\
& \langle \exists\, x \;::\; x\, FS\; (path\, d) \rangle
\end{array}$$

In summary, $path \cdot FT \subseteq FS^\circ \cdot \top$ unfolds into

$$\langle \forall\, d,h \;:\; d\, FT\, h :\; \langle \exists\, x \;::\; x\, FS\; (path\, d) \rangle \rangle \tag{5.29}$$

Literally:

> *If h is the handle of some open-file descriptor d, then this holds the path of some existing file x.*

In fewer words:

> *Non-existing files cannot be opened (*referential integrity*).*

Thus we see how relation diagrams "hide" logically quantified formulæ capturing properties of the systems one wishes to describe.

Compared with the commutative diagrams of previous chapters, a diagram

$$\begin{tikzcd}
A \arrow[d, "R"'] \arrow[dr, phantom, "\subseteq"] & B \arrow[l, "S"'] \arrow[d, "P"] \\
C & D \arrow[l, "Q"]
\end{tikzcd}$$

is said to be *semi-commutative* because $Q \cdot P \;\subseteq\; R \cdot S$ is not forced to hold, only $R \cdot S \;\subseteq\; Q \cdot P$ is. In case both hold, the $\subseteq$ symbol is dropped, cf. (5.20).

***Exercise** 5.3. Let $a\ S\ n$ mean:* "student a is assigned number n". *Using (5.11) and (5.19), check that assertion*

$$S \cdot \geqslant \;\subseteq\; \top \cdot S \quad \textit{depicted by diagram} \quad \begin{tikzcd}
\mathbb{N}_0 \arrow[d, "S"'] \arrow[dr, phantom, "\subseteq"] & \mathbb{N}_0 \arrow[l, "\geqslant"'] \arrow[d, "S"] \\
A & A \arrow[l, "\top"]
\end{tikzcd}$$

means that numbers are assigned to students in increasing order.
□

5.7 TAXONOMY OF BINARY RELATIONS

The Leibniz principle about functions (5.6) can now be simplified thanks to equivalence (5.19), as shown next:

$$\begin{array}{ll}
& \langle \forall\, a, a' \;::\; a = a' \;\Rightarrow\; f\ a = f\ a' \rangle \\
\equiv & \qquad \{ \text{ introduction of } id\text{; consequent as calculated already } \} \\
& \langle \forall\, a, a' \;::\; a = id\ a' \;\Rightarrow\; a(f^{\circ} \cdot f)a' \rangle \\
\equiv & \qquad \{ \; b\ f\ a \text{ means the same as } b = f\ a \; \} \\
& \langle \forall\, a, a' \;::\; a\ id\ a' \;\Rightarrow\; a(f^{\circ} \cdot f)a' \rangle \\
\equiv & \qquad \{ \text{ (5.19) } \} \\
& id \;\subseteq\; f^{\circ} \cdot f
\end{array} \tag{5.30}$$

A similar calculation will reduce univocality (5.5) to

$$f \cdot f^{\circ} \;\subseteq\; id \tag{5.31}$$

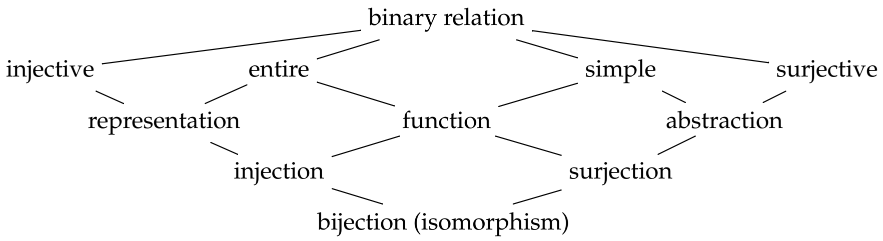


Figure 5.1.: Binary relation taxonomy

Thus a function f is characterized by comparing $f^\circ \cdot f$ and $f \cdot f^\circ$ with the identity.[7]

The exact characterization of functions as special cases of relations is achieved in terms of converse, which is in fact of paramount importance in establishing the whole taxonomy of binary relations depicted in figure 5.1. First, we need to define two important notions: given a relation $B \xleftarrow{R} A$, the *kernel* of R is the relation $A \xleftarrow{\mathsf{ker}\, R} A$ defined by:

$$\mathsf{ker}\, R = R^\circ \cdot R \tag{5.32}$$

Clearly, a' $\mathsf{ker}\, R$ a holds between any two sources a and a' which have (at least) a common target c such that $c\ R\ a'$ and $c\ R\ a$. We can also define its dual, $B \xleftarrow{\mathsf{img}\, R} B$, called the *image* of R, defined by:[8]

$$\mathsf{img}\, R \;=\; R \cdot R^\circ \tag{5.33}$$

From (5.15, 5.16) one immediately draws:

$$\mathsf{ker}\,(R^\circ) \;=\; \mathsf{img}\, R \tag{5.34}$$

$$\mathsf{img}\,(R^\circ) \;=\; \mathsf{ker}\, R \tag{5.35}$$

Kernel and image lead to the four top criteria of the taxonomy of figure 5.1:

	Reflexive	*Coreflexive*
$\mathsf{ker}\, R$	entire R	injective R
$\mathsf{img}\, R$	surjective R	simple R

(5.36)

In words: a relation R is said to be *entire* (or total) iff its kernel is reflexive and to be *simple* (or functional) iff its image is coreflexive. Dually, R is *surjective* iff R° is entire, and R is *injective* iff R° is simple.

[7] As we shall see in section 5.13, relations larger than the identity ($id \subseteq R$) are said to be *reflexive* and relations at most the identity ($R \subseteq id$) are said to be *coreflexive* or *partial identities*.

[8] These operators are relational extensions of two concepts familiar from set theory: the image of a function f, which corresponds to the set of all y such that $\langle \exists\, x\ ::\ y = f\ x \rangle$, and the kernel of f, which is the equivalence relation $b\ \mathsf{ker} f\ a \Leftrightarrow (f\ b) = (f\ a)$. (See exercise 5.8 later on.)

Representing binary relations by Boolean matrices gives us a simple, graphical way of detecting properties such as simplicity, surjectiveness, and so on. Let the enumerated types $A = \{a_1, a_2, a_3, a_4, a_5\}$ and $B = \{b_1, b_2, b_3, b_4, b_5\}$ be given. Two examples of relations of type $A \to B$ are given in figure 5.2 — the leftmost and the rightmost, which we shall refer to as R and S, respectively.[9] The matrix representing R is:

$$\begin{array}{c|ccccc} & a_1 & a_2 & a_3 & a_4 & a_5 \\ \hline b_1 & 0 & 1 & 0 & 0 & 0 \\ b_2 & 1 & 0 & 0 & 0 & 0 \\ b_3 & 0 & 0 & 1 & 1 & 0 \\ b_4 & 0 & 0 & 0 & 0 & 1 \\ b_5 & 0 & 0 & 0 & 0 & 0 \end{array} \tag{5.37}$$

The 1 addressed by b_2 and a_1 means that $b_2 \; R \; a_1$ holds, that between b_1 and a_2 means $b_1 \; R \; a_2$, and so on and so forth. Then, R is:

- *simple* because there is *at most* one 1 in every column
- *entire* because there is *at least* one 1 in every column
- not *injective* because there is *more than* one 1 in some row
- not *surjective* because some row (the last) has no 1s.

So this relation is a *function* that is neither an injection nor a surjection.

Let us now have a look at the matrix that represents $S : A \to B$:

$$\begin{array}{c|ccccc} & a_1 & a_2 & a_3 & a_4 & a_5 \\ \hline b_1 & 0 & 1 & 0 & 0 & 0 \\ b_2 & 1 & 0 & 0 & 0 & 0 \\ b_3 & 0 & 0 & 0 & 1 & 0 \\ b_4 & 0 & 0 & 0 & 0 & 1 \\ b_5 & 0 & 0 & 1 & 0 & 0 \end{array}$$

Now every row and every column has *exactly* one 1 — this tells us that S is not only a function but in fact a bijection. Looking at the matrix that represents $S^\circ : A \leftarrow B$,

$$\begin{array}{c|ccccc} & b_1 & b_2 & b_3 & b_4 & b_5 \\ \hline a_1 & 0 & 1 & 0 & 0 & 0 \\ a_2 & 1 & 0 & 0 & 0 & 0 \\ a_3 & 0 & 0 & 0 & 0 & 1 \\ a_4 & 0 & 0 & 1 & 0 & 0 \\ a_5 & 0 & 0 & 0 & 1 & 0 \end{array}$$

we realize that it also is a function, in fact another bijection. This gives us a rule of thumb for (constructively) checking for bijections (isomorphisms):

$$\textit{A function } f \textit{ is a bijection } \text{iff } \textit{its converse } f^\circ \textit{ is a function } g \tag{5.38}$$

[9] Credits: `http://www.matematikaria.com/unit/injective-surjective-bijective.html`. Note that we enumerate a_1, a_2, ... from the top to the bottom.

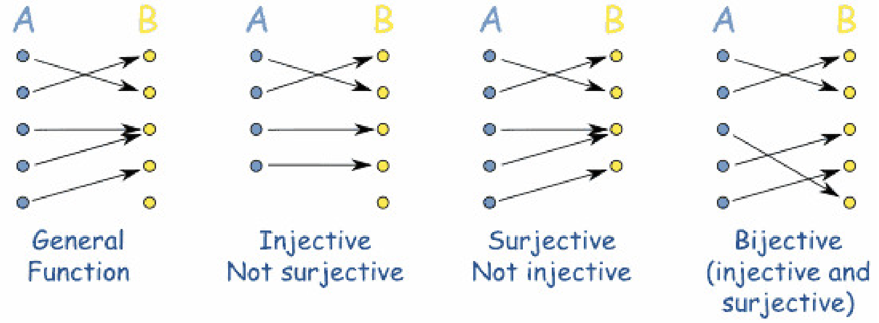


Figure 5.2.: Four binary relations.

Then g is also a bijection since $f^\circ = g \Leftrightarrow f = g^\circ$.[10] Recall how some definitions of isomorphisms given before, e.g. (2.94), are nothing but applications of this rule $f^\circ = g$, once written pointwise with the help of (5.17):

$$f\ b = a \Leftrightarrow b = g\ a$$

Bijections (isomorphisms) are reversible functions — they don't lose any information. By contrast, $! : A \to 1$ (2.58) and indeed all constant functions $\underline{c} : A \to C$ (2.12) lose all the information contained in their inputs, recall (2.14). This property is actually more general,

$$\underline{c} \cdot R \subseteq \underline{c} \tag{5.39}$$

for all suitably typed R.

In the same way $! : A \to 1$ is always a constant function — in fact the *unique* possible function of its type, $f : 1 \to A$ is bound to be a constant function too, for any choice of a target value in non-empty A. Because there are as many such functions as elements if A, functions $\underline{a} : 1 \to A$ are referred to as *points*. These two situations correspond to isomorphisms $1^A \cong 1$ (2.99) and $A^1 \cong A$ (2.100), respectively. Two short-hands are introduced for the constant functions

$$true = \underline{\mathsf{True}} \tag{5.40}$$

$$false = \underline{\mathsf{False}} \tag{5.41}$$

***Exercise** 5.4. Prove (5.38) by completing:*

$$\begin{array}{ll}
& f \text{ and } f^\circ \text{ are functions} \\
\equiv & \quad \{ \ldots \} \\
& (id \subseteq \mathsf{ker}\ f \wedge \mathsf{img}\ f \subseteq id) \wedge (id \subseteq \mathsf{ker}\ (f^\circ) \wedge \mathsf{img}\ (f^\circ) \subseteq id) \\
\equiv & \quad \{ \ldots \} \\
& \vdots \\
\equiv & \quad \{ \ldots \} \\
& f \text{ is a bijection}
\end{array}$$

[10] The interested reader may go back to (2.18,2.19) at this point and check these rules in the light of (5.38).

□

***Exercise** 5.5. Compute, for the relations in figure 5.2, the* kernel *and the* image *of each relation. Why are all these relations* functions*? (NB: note that the types are not all the same.)*

□

***Exercise** 5.6. Recall the definition of a constant function (2.12),*

$$\begin{array}{rcl} \underline{k} & : & A \to K \\ \underline{k}\, a & = & k \end{array}$$

where K is assumed to be non-empty. Using (5.11), show that $\mathsf{ker}\ \underline{k} = \top$ *and compute which relations are defined by the expressions*

$$\underline{b} \cdot \underline{a}^{\circ}, \quad \mathsf{img}\ \underline{k} \tag{5.42}$$

Finally, show that (5.39) holds.

□

***Exercise** 5.7. Resort to (5.34,5.35) and (5.36) to prove the following rules of thumb:*

- *converse of* injective *is* simple *(and vice-versa)* (5.43)
- *converse of* entire *is* surjective *(and vice-versa)* (5.44)

□

***Exercise** 5.8. Given a function* $B \xleftarrow{f} A$ *, calculate the pointwise version*

$$b(\mathsf{ker}\ f)a \quad \equiv \quad f\ b = f\ a \tag{5.45}$$

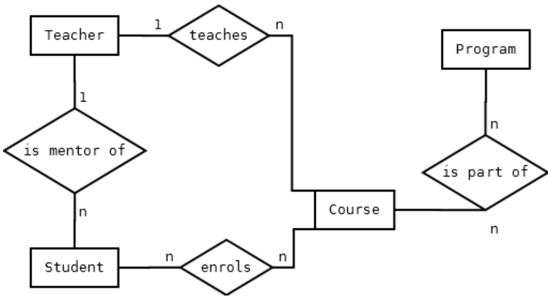
of $\mathsf{ker}\ f$*. What is the outcome of the same exercise for* $\mathsf{img}\ f$*?*

□

ENTITY-RELATIONSHIP DIAGRAMS In the tradition of relational databases, so-called *entity-relationship* (ER) diagrams have become popular as an informal means for capturing the properties of the relationships involved in a particular database design.

Consider the following example of one such diagram:[11]

[11] Credits: `https://dba.stackexchange.com/questions`.



In the case of relation

$$\mathit{Teacher} \xleftarrow{\mathit{is\ mentor\ of}} \mathit{Student}$$

the drawing tells not only that some teacher may mentor more than one student, but also that a given student has exactly one mentor. So *is mentor of* is a *simple* relation (figure 5.1).

The possibility $n = 0$ allows for students with no mentor. Should this possibility be ruled out ($n > 1$), the relation would become also *entire*, i.e. a function. Then

$$t \ \mathit{is\ mentor\ of}\ s$$

could be written

$$t = \mathit{is\ mentor\ of}\ s$$

— recall (5.7) — meaning:

t is *the* mentor of student *s*.

That is, *is mentor of* would become an *attribute* of *Student*. Note how definite article "the" captures the presence of functions in normal speech. "The" means not only determinism (one and only one output) but also definedness (there is always one such output). In the case of *is mentor of* being simple but not entire, we have to say:

t is the *mentor of student s,* if any.

Exercise *5.9. Complete the exercise of declaring in* $A \xrightarrow{R} B$ *notation the other relations of the ER-diagram above and telling which properties in Figure 5.1 are required for such relations.*

□

5.8 FUNCTIONS, RELATIONALLY

Among all binary relations, functions play a central role in relation algebra — as can be seen in figure 5.1. Recapitulating, a *function f* is a binary relation such that

Pointwise	Pointfree	
"Left" Uniqueness		
$b\,f\,a \wedge b'\,f\,a \;\Rightarrow\; b = b'$	$\mathsf{img}\,f \;\subseteq\; id$	(f is simple)
Leibniz principle		
$a = a' \;\Rightarrow\; f\,a = f\,a'$	$id \;\subseteq\; \mathsf{ker}\,f$	(f is entire)

It turns out that *any* function f enjoys the following properties, known as *shunting rules*:

$$f \cdot R \subseteq S \;\equiv\; R \subseteq f^{\circ} \cdot S \tag{5.46}$$

$$R \cdot f^{\circ} \subseteq S \;\equiv\; R \subseteq S \cdot f \tag{5.47}$$

These will prove extremely useful in the sequel. Another very useful fact is the function *equality rule*:

$$f \subseteq g \;\equiv\; f = g \;\equiv\; f \supseteq g \tag{5.48}$$

Rule (5.48) follows immediately from (5.46,5.47) by "cyclic inclusion" (5.20):

$$\begin{array}{ll}
 & f \subseteq g \\
\equiv & \quad \{ \text{ natural-}id\text{ (2.10) } \} \\
 & f \cdot id \subseteq g \\
\equiv & \quad \{ \text{ shunting on } f \text{ (5.46) } \} \\
 & id \subseteq f^{\circ} \cdot g \\
\equiv & \quad \{ \text{ shunting on } g \text{ (5.47) } \} \\
 & id \cdot g^{\circ} \subseteq f^{\circ} \\
\equiv & \quad \{ \text{ converses; identity } \} \\
 & g \subseteq f
\end{array}$$

Then:

$$\begin{array}{ll}
 & f = g \\
\equiv & \quad \{ \text{ cyclic inclusion (5.20) } \} \\
 & f \subseteq g \wedge g \subseteq f \\
\equiv & \quad \{ \text{ above } \} \\
 & f \subseteq g \\
\equiv & \quad \{ \text{ above } \} \\
 & g \subseteq f \\
\Box &
\end{array}$$

***Exercise** 5.10. Infer* $id \subseteq \mathsf{ker}\ f$ *(f is entire) and* $\mathsf{img}\ f \subseteq id$ *(f is simple) from shunting rules (5.46) and (5.47).*

□

***Exercise** 5.11. For $R := f$, the property (5.39) "immediately" coincides with (2.14). Why?*

□

FUNCTION DIVISION. Given two functions $B \xrightarrow{g} C \xleftarrow{f} A$, we can compose f with the converse of g. This turns out to be a very frequent pattern in relation algebra, known as the *division* of f by g:

$$\frac{f}{g} = g^\circ \cdot f \qquad cf. \quad \begin{tikzcd} B \arrow[dr, "g"'] & & A \arrow[ll, "\frac{f}{g}"'] \arrow[dl, "f"] \\ & C & \end{tikzcd} \tag{5.49}$$

That is,

$$b \frac{f}{g} a \Leftrightarrow g\, b = f\, a$$

Think of the sentence:

Mary lives where John was born.

This can be expressed by a division:

$$Mary \frac{birthplace}{residence} John \Leftrightarrow residence\ Mary = birthplace\ John$$

Thus $R = \frac{birthplace}{residence}$ is the relation "... resides in the birthplace of ...". In general,

$b \frac{f}{g} a$ means "the g of b is the f of a".

This combinator enjoys a number of interesting properties, for instance:

$$\frac{f}{id} = f \tag{5.50}$$

$$\left(\frac{f}{g}\right)^\circ = \frac{g}{f} \tag{5.51}$$

$$\frac{f \cdot h}{g \cdot k} = k^\circ \cdot \frac{f}{g} \cdot h \tag{5.52}$$

$$\frac{f}{f} = \mathsf{ker}\, f \tag{5.53}$$

$$a \neq b \Leftrightarrow \frac{\underline{a}}{\underline{b}} = \bot \tag{5.54}$$

Function division is a special case of the more general, and important, concept of relational division, a topic that shall be addressed in section 5.19.

Exercise *5.12. The teams (T) of a football league play games (G) at home or away, and every game takes place in some date:*

$$\begin{tikzcd}
T & G \arrow[l, "home"'] \arrow[r, "away"] \arrow[d, "date"'] & T \\
 & D &
\end{tikzcd}$$

Moreover, (a) No team can play two games on the same date; (b) All teams play against each other but not against themselves; (c) For each home game there is another game away involving the same two teams. Show that

$$id \subseteq \frac{away}{home} \cdot \frac{away}{home} \tag{5.55}$$

captures one of the requirements above — which?
□

Exercise *5.13. Check the properties of function division given above.*
□

5.9 MEET AND JOIN

Like sets, two relations of the same type, say $B \xleftarrow{R,S} A$, can be intersected or joined in the obvious way:

$$b\,(R \cap S)\,a \equiv b\,R\,a \wedge b\,S\,a \tag{5.56}$$

$$b\,(R \cup S)\,a \equiv b\,R\,a \vee b\,S\,a \tag{5.57}$$

$R \cap S$ is usually called *meet* (intersection) and $R \cup S$ is called *join* (union). They lift pointwise conjunction and disjunction, respectively, to the pointfree level. Their meaning is nicely captured by the following *universal* properties:[12]

$$X \subseteq R \cap S \equiv X \subseteq R \wedge X \subseteq S \tag{5.58}$$

$$R \cup S \subseteq X \equiv R \subseteq X \wedge S \subseteq X \tag{5.59}$$

Meet and join have the expected properties, e.g. *associativity*

$$(R \cap S) \cap T = R \cap (S \cap T)$$

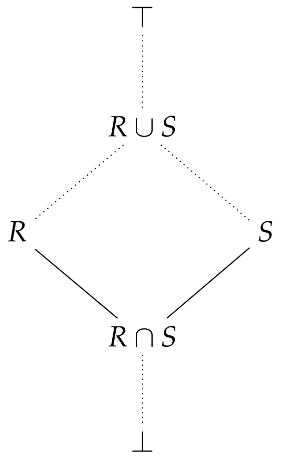
[12] Recall the generic notions of *greatest lower bound* and *least upper bound*, respectively.

proved next by indirect equality (5.23):

$$\begin{array}{ll}
& X \subseteq (R \cap S) \cap T \\
\equiv & \qquad \{ \cap\text{-universal (5.58) twice } \} \\
& (X \subseteq R \wedge X \subseteq S) \wedge X \subseteq T \\
\equiv & \qquad \{ \wedge \text{ is associative } \} \\
& X \subseteq R \wedge (X \subseteq S \wedge X \subseteq T) \\
\equiv & \qquad \{ \cap\text{-universal (5.58) twice } \} \\
& X \subseteq R \cap (S \cap T) \\
:: & \qquad \{ \text{ indirection (5.23) } \} \\
& (R \cap S) \cap T = R \cap (S \cap T)
\end{array}$$

□

In summary, type $B \leftarrow A$ forms a lattice:



"top"

join, lub ("least upper bound")

meet, glb ("greatest lower bound")

"bottom"

DISTRIBUTIVE PROPERTIES. As it will be proved later, *composition* distributes over *union*

$$R \cdot (S \cup T) \;=\; (R \cdot S) \cup (R \cdot T) \tag{5.60}$$

$$(S \cup T) \cdot R \;=\; (S \cdot R) \cup (T \cdot R) \tag{5.61}$$

while distributivity over *intersection* is side-conditioned:

$$(S \cap Q) \cdot R = (S \cdot R) \cap (Q \cdot R) \;\Leftarrow\; \left\{ \begin{array}{c} Q \cdot \mathsf{img}\ R \subseteq Q \\ \vee \\ S \cdot \mathsf{img}\ R \subseteq S \end{array} \right. \tag{5.62}$$

$$R \cdot (Q \cap S) = (R \cdot Q) \cap (R \cdot S) \;\Leftarrow\; \left\{ \begin{array}{c} (\mathsf{ker}\ R) \cdot Q \subseteq Q \\ \vee \\ (\mathsf{ker}\ R) \cdot S \subseteq S \end{array} \right. \tag{5.63}$$

Properties (5.60,5.61) express the *bilinearity* of relation composition with respect to relational join. These, and properties such as e.g.

$$(R \cap S)^\circ = R^\circ \cap S^\circ \tag{5.64}$$

$$(R \cup S)^\circ = R^\circ \cup S^\circ \tag{5.65}$$

will be shown to derive from a general construction that will be explained in section 5.18.

Exercise *5.14. Show that*

$$R \cap \bot = \bot \tag{5.66}$$

$$R \cap \top = R \tag{5.67}$$

$$R \cup \top = \top \tag{5.68}$$

$$R \cup \bot = R \tag{5.69}$$

using neither (5.56) nor (5.57).

□

Exercise *5.15. Prove the* union simplicity *rule:*

$$M \cup N \textit{ is simple} \quad \equiv \quad M, N \textit{ are simple and } M \cdot N^{\circ} \subseteq id \tag{5.70}$$

Using converses, derive from (5.70) the corresponding rule for injective *relations.*

□

Exercise *5.16. Prove the distributive property:*

$$g^{\circ} \cdot (R \cap S) \cdot f \quad = \quad g^{\circ} \cdot R \cdot f \cap g^{\circ} \cdot S \cdot f \tag{5.71}$$

□

Exercise *5.17. Let $bag : A^* \rightarrow \mathbb{N}_0{}^A$ be the function that, given a finite sequence (list), indicates the number of occurrences of its elements, for instance,*

$$bag\ [a, b, a, c]\ a = 2$$

$$bag\ [a, b, a, c]\ b = 1$$

$$bag\ [a, b, a, c]\ c = 1$$

Let $ord : A^* \rightarrow \mathbb{B}$ be the obvious predicate assuming a total order predefined in A. Finally, let $true = \underline{\mathsf{True}}$ (5.40). Having defined

$$S = \frac{bag}{bag} \cap \frac{true}{ord} \tag{5.72}$$

identify the type of S and, going pointwise and simplifying, tell which operation is specified by S.

□

Exercise *5.18. Derive the distributive properties:*

$$k^{\circ} \cdot (f \cup g) = \frac{f}{k} \cup \frac{g}{k} \quad , \quad k^{\circ} \cdot (f \cap g) = \frac{f}{k} \cap \frac{g}{k} \tag{5.73}$$

□

5.10 RELATIONAL THINKING

Binary relations provide a natural way of describing real life situations. Relation algebra can be used to reason about such formal descriptions. This can be achieved using suitable relational combinators (and their laws), in the *pointfree* style.

Let us see a simple example of such a *relational thinking* taking one of the PROPOSITIONES AD ACUENDOS IUUENES ("Problems to Sharpen the Young") proposed by abbot Alcuin of York († 804) as case study. Alcuin states his puzzle in the following way, in Latin:

> XVIII. PROPOSITIO DE HOMINE ET CAPRA ET LVPO. *Homo quidam debebat ultra fluuium transferre lupum, capram, et fasciculum cauli. Et non potuit aliam nauem inuenire, nisi quae duos tantum ex ipsis ferre ualebat. Praeceptum itaque ei fuerat, ut omnia haec ultra illaesa omnino transferret. Dicat, qui potest, quomodo eis illaesis transire potuit?*

Our starting point will be the following (rather free) translation of the above to English:

> XVIII. FOX, GOOSE AND BAG OF BEANS PUZZLE. *A farmer goes to market and purchases a fox, a goose, and a bag of beans. On his way home, the farmer comes to a river bank and hires a boat. But in crossing the river by boat, the farmer could carry only himself and a single one of his purchases - the fox, the goose or the bag of beans. (If left alone, the fox would eat the goose, and the goose would eat the beans.) Can the farmer carry himself and his purchases to the far bank of the river, leaving each purchase intact?*

We wish to describe the essence of this famous puzzle, which is the *guarantee* that

> *under no circumstances does the fox eat the goose or the goose eat the beans.*

Clearly, we need two data types:

$$Being = \{Farmer, Fox, Goose, Beans\}$$
$$Bank = \{Left, Right\}$$

Then we identify a number of relations involving such data:

$$\begin{tikzcd}
Being \arrow[r, "Eats"] & Being \arrow[d, "where"'] & \\
& Bank \arrow[r, "cross"] & Bank
\end{tikzcd} \tag{5.74}$$

Clearly, $cross\ Left = Right$ and $cross\ Right = Left$. So *cross* is its own inverse and therefore a bijection (5.38). Relation *Eats* can be described by the Boolean matrix:

$$Eats \quad = \quad \begin{array}{r|cccc} & Fox & Goose & Beans & Farmer \\ \hline Fox & 0 & 1 & 0 & 0 \\ Goose & 0 & 0 & 1 & 0 \\ Beans & 0 & 0 & 0 & 0 \\ Farmer & 0 & 0 & 0 & 0 \end{array} \qquad (5.75)$$

Relation $where : Being \to Bank$ is necessarily a function because:

- *everyone is somewhere in a bank* (*where* is entire)
- *no one can be in both banks at the same time* (*where* is simple)

Note that there are only two constant functions of type $Being \to Bank$, $\underline{Right}$ and $\underline{Left}$. The puzzle consists in changing from the state $where = \underline{Right}$ to the state $where = \underline{Left}$, for instance, without violating the property that *nobody eats anybody*. How does one record such a property? We need two auxiliary relations capturing, respectively:

- Being at the same bank:

$$SameBank \quad = \quad \mathsf{ker}\ where$$

- Risk of somebody eating somebody else:

$$CanEat \quad = \quad SameBank \cap Eats$$

Then "starvation" is ensured by the *Farmer*'s presence at the same bank:

$$CanEat \quad \subseteq \quad SameBank \cdot \underline{Farmer} \qquad (5.76)$$

By (5.46), this "starvation" property (5.76) converts to:

$$where \cdot CanEat \quad \subseteq \quad where \cdot \underline{Farmer}$$

In this version, (5.76) can be depicted as a diagram

$$\begin{tikzcd} Being \arrow[d, "where"'] \arrow[dr, phantom, "\subseteq"] & Being \arrow[l, "CanEat"'] \arrow[d, "\underline{Farmer}"] \\ Bank & Being \arrow[l, "where"] \end{tikzcd} \qquad (5.77)$$

which "reads" in a nice way:

where `(somebody)` *CanEat* `(somebody else) (that's)` *where* `(the)` *Farmer* `(is).`

Diagram (5.27) given earlier can now be identified as another example of assertion expressed relationally. Diagrams of this kind capture properties of data models that one wishes to hold at any time during the lifetime of the system being described. Such properties are commonly referred to as *invariants* and their preservation by calculation will be the main aim of chapter 7.

***Exercise** 5.19. Calculate the following pointwise version of the "starvation" property (5.77) by introducing quantifiers and simplifying:*

$$\langle \forall\, b', b \;:\; b'\ Eat\ b :\ where\ b' = where\ b \Rightarrow where\ b' = where\ Farmer \rangle$$

□

***Exercise** 5.20. Recalling property (5.39), show that the "starvation" property (5.77) is satisfied by any of the two constant functions that model the start or end states of the Alcuin puzzle.*

□

5.11 MONOTONICITY

As expected, relational composition is monotonic:

$$\frac{\begin{array}{c} R \subseteq S \\ T \subseteq U \end{array}}{(R \cdot T) \subseteq (S \cdot U)} \tag{5.78}$$

Indeed, all relational combinators studied so far are also monotonic, namely

$$R \subseteq S \;\Rightarrow\; R^\circ \subseteq S^\circ \tag{5.79}$$

$$R \subseteq S \wedge U \subseteq V \;\Rightarrow\; R \cap U \subseteq S \cap V \tag{5.80}$$

$$R \subseteq S \wedge U \subseteq V \;\Rightarrow\; R \cup U \subseteq S \cup V \tag{5.81}$$

hold.

Monotonicity and transitivity (5.22) are important properties for reasoning about a given relational inclusion $R \subseteq S$. In particular, the following rules are of help by relying on a "mid-point" relation M, $R \subseteq M \subseteq S$ (analogy with interval arithmetics).

- Rule A — *lowering the upper side*:

$$\begin{array}{cl} & R \subseteq S \\ \Leftarrow & \quad \{\ M \subseteq S \text{ is known ; transitivity of } \subseteq \text{(5.22)}\ \} \\ & R \subseteq M \end{array}$$

  Then proceed with $R \subseteq M$.

- Rule B — *raising the lower side*:

$$\begin{array}{cl} & R \subseteq S \\ \Leftarrow & \quad \{\ R \subseteq M \text{ is known; transitivity of } \subseteq\ \} \\ & M \subseteq S \end{array}$$

  Then proceed with $M \subseteq S$.

The following proof of shunting property (5.46) combines these rules with monotonicity and circular implication:

$$\begin{array}{cl} & R \subseteq f^\circ \cdot S \\ \Leftarrow & \quad \{\ id \subseteq f^\circ \cdot f \text{ ; raising the lower-side }\} \\ & f^\circ \cdot f \cdot R \subseteq f^\circ \cdot S \\ \Leftarrow & \quad \{\ \text{monotonicity of } (f^\circ \cdot)\ \} \\ & f \cdot R \subseteq S \\ \Leftarrow & \quad \{\ f \cdot f^\circ \subseteq id \text{ ; lowering the upper-side }\} \\ & f \cdot R \subseteq f \cdot f^\circ \cdot S \\ \Leftarrow & \quad \{\ \text{monotonicity of } (f \cdot)\ \} \\ & R \subseteq f^\circ \cdot S \end{array}$$

Thus the equivalence in (5.46) is established by circular implication.

Rules A and B should be used only where other proof techniques (notably indirect equality) fail. They assume judicious choice of the mid-point relation M, at each step. The choice of an useless M can drive the proof nowhere.

***Exercise** 5.21. Unconditional distribution laws*

$$\begin{array}{rcl} (P \cap Q) \cdot S & = & (P \cdot S) \cap (Q \cdot S) \\ R \cdot (P \cap Q) & = & (R \cdot P) \cap (R \cdot Q) \end{array}$$

will hold provide one of R or S is simple and the other injective. Tell which, justifying.

□

***Exercise** 5.22. Prove that relational* composition *preserves* all *relational classes in the taxonomy of figure 5.1.*

□

5.12 RULES OF THUMB

Quite often, involved reasoning in logic arguments can be replaced by simple and elegant calculations in relation algebra that arise thanks to smart"rules of thumb". We have already seen two such rules, (5.43) and (5.44). Two others are:

- *smaller* than injective (simple) is injective (simple) (5.82)
- *larger* than entire (surjective) is entire (surjective) (5.83)

Let us see these rules in action in trying to infer what can be said of two functions f and r such that

$$f \cdot r = id$$

holds. On the one hand,

$$\begin{array}{ll} & f \cdot r = id \\ \equiv & \quad \{ \text{ equality of functions } \} \\ & f \cdot r \subseteq id \\ \equiv & \quad \{ \text{ shunting } \} \\ & r \subseteq f^{\circ} \end{array}$$

Since f is simple, f° is injective and so is r because "smaller than injective is injective". On the other hand,

$$\begin{array}{ll} & f \cdot r = id \\ \equiv & \quad \{ \text{ equality of functions } \} \\ & id \subseteq f \cdot r \\ \equiv & \quad \{ \text{ shunting } \} \\ & r^{\circ} \subseteq f \end{array}$$

Since r is entire, r° is surjective and so is f because "larger that surjective is surjective". We conclude that f is surjective and r is injective wherever $f \cdot r = id$ holds. Since both are functions, we furthermore conclude that

f is an *abstraction* and r is a *representation*

— cf. Figure 5.1.

The reason for this terminology can now be explained. Given $f : A \leftarrow C$ and $r : C \leftarrow A$ such that $f \cdot r = id$, that is, for all $a \in A, f\ (r\ a) = a$, think of C as a domain of *concrete* objects and of A as a domain of *abstract* data. For instance, let $A = \mathbb{B}$ and $C = \mathbb{N}_0$. Then define

$$\left\{ \begin{array}{l} r : \mathbb{B} \rightarrow \mathbb{N}_0 \\ r\ b = \textbf{if}\ b\ \textbf{then}\ k\ \textbf{else}\ 0 \end{array} \right.$$

(where k is any natural number different from 0) and

$$\begin{cases} f : \mathbb{B} \leftarrow \mathbb{N}_0 \\ f\ n = \textbf{if}\ n = 0\ \textbf{then}\ \mathsf{False}\ \textbf{else}\ \mathsf{True} \end{cases}$$

Clearly, by the definitions of f and r:

$$\begin{array}{ll} & f\ (r\ b) = \textbf{if}\ (\textbf{if}\ b\ \textbf{then}\ k\ \textbf{else}\ 0) = 0\ \textbf{then}\ \mathsf{False}\ \textbf{else}\ \mathsf{True} \\ \equiv & \qquad \{\ \text{conditional-fusion rule (2.71)}\ \} \\ & f\ (r\ b) = \textbf{if}\ (\textbf{if}\ b\ \textbf{then}\ k = 0\ \textbf{else}\ \mathsf{True})\ \textbf{then}\ \mathsf{False}\ \textbf{else}\ \mathsf{True} \\ \equiv & \qquad \{\ k = 0\ \text{is always false}\ \} \\ & f\ (r\ b) = \textbf{if}\ (\textbf{if}\ b\ \textbf{then}\ \mathsf{False}\ \textbf{else}\ \mathsf{True})\ \textbf{then}\ \mathsf{False}\ \textbf{else}\ \mathsf{True} \\ \equiv & \qquad \{\ \text{pointwise definition of}\ \neg\ b\ \} \\ & f\ (r\ b) = \textbf{if}\ \neg\ b\ \textbf{then}\ \mathsf{False}\ \textbf{else}\ \mathsf{True} \\ \equiv & \qquad \{\ \text{trivial}\ \} \\ & b \end{array}$$

That is, r represents the Booleans True and False by natural numbers while f abstracts from such real numbers back to Booleans. r being injective means r False $\neq r$ True, that is, the Boolean information is not lost in the representation.[13] f being surjective means that any Boolean is representable. Note that $r \cdot f = id$ does not hold: $r\ (f\ 1) = r\ \mathsf{True} = k$ and $k \neq 1$ in general.

The abstraction/representation pair (f, r) just above underlies the way Booleans are handled in programming languages such as C, for instance. Experienced programmers will surely agree that often what is going on in the code they write are processes of representing information using primitive data structures available from the adopted programming language. For instance, representing finite sets by finite lists corresponds to the *abstraction* given by elems (5.1).

Exercise *5.23. Recalling exercise 5.17, complete the definition of*

$$\begin{array}{l} bag\ [\,]\ a = 0 \\ bag\ (h : t)\ a = \textbf{let}\ b = bag\ t\ \textbf{in}\ \textbf{if} \ldots \end{array}$$

Is this function an abstraction or a representation? Justify your answer informally.
□

Exercise *5.24.Show that:*

- *$R \cap S$ is injective (simple) provided one of R or S is so*
- *$R \cup S$ is entire (surjective) provided one of R or S is so.*

□

[13] That is, r causes *no confusion* in the representation process.

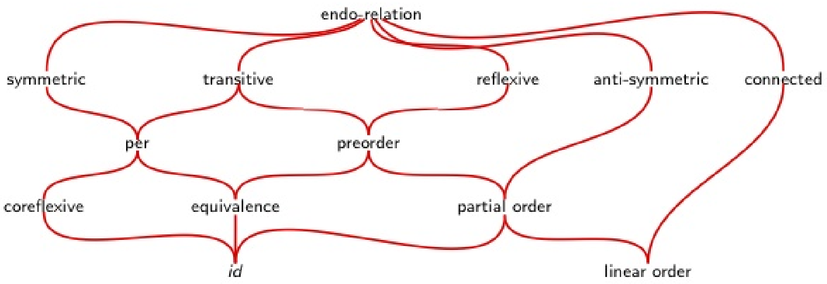


Figure 5.1.: Taxonomy of endorelations.

5.13 ENDO-RELATIONS

Relations in general are of type $A \rightharpoonup B$, for some A and B. In the special case that $A = B$ holds, a relation $R : A \rightharpoonup A$ is said to be an *endo-relation*, or a *graph*. The $A = B$ coincidence gives room for some extra terminology, extending some already given. Besides an endo-relation $A \xleftarrow{R} A$ being

reflexive:	iff $id \subseteq R$	(5.84)
coreflexive:	iff $R \subseteq id$	(5.85)

it can also be:

transitive:	iff $R \cdot R \subseteq R$	(5.86)
symmetric:	iff $R \subseteq R^{\circ} (\equiv R = R^{\circ})$	(5.87)
anti-symmetric:	iff $R \cap R^{\circ} \subseteq id$	(5.88)
irreflexive:	iff $R \cap id = \bot$	(5.89)
connected:	iff $R \cup R^{\circ} = \top$	(5.90)

By combining these criteria, endo-relations $A \xleftarrow{R} A$ can further be classified as in figure 5.1. In summary:

- *Preorders* are reflexive and transitive orders.
  Example: $age\ y \leqslant age\ x$.
- *Partial* orders are anti-symmetric preorders
  Example: $y \subseteq x$ where x and y are sets.
- *Linear* orders are connected partial orders
  Example: $y \leqslant x$ in $\mathbb{N}_0$
- *Equivalences* are symmetric preorders
  Example: $age\ y = age\ x$. [14]

[14] Kernels of functions are always equivalence relations, see exercise 5.25.

- *Pers* are partial equivalences
  Example: $y\ IsBrotherOf\ x$.

Preorders are normally denoted by asymmetric symbols such as e.g. $y \sqsubseteq x$, $y \leqslant x$. In case of a function f such that

$$f \cdot (\sqsubseteq) \ \subseteq\ (\leqslant) \cdot f \tag{5.91}$$

we say that f is monotonic. Indeed, this is equivalent to

$$a \sqsubseteq b \Rightarrow (f\ a) \leqslant (f\ b)$$

once shunting (5.46) takes place, and variables are added and handled via (5.17). Another frequent situation is that of two functions f and g such that

$$f\ \subseteq\ (\leqslant) \cdot g \tag{5.92}$$

This converts to the pointwise

$$\langle \forall\ a\ ::\ f\ a \leqslant g\ a \rangle$$

that is, f is *always at most* g for all possible inputs. The following abbreviation is often used to capture this ordering on functions induced by a pre-order $(\leqslant)$ on their outputs:

$$f\ \stackrel{.}{\leqslant}\ g \quad \mathit{iff} \quad f\ \subseteq\ (\leqslant) \cdot g \tag{5.93}$$

For instance, $f\ \stackrel{.}{\leqslant}\ id$ means $f\ a \leqslant a$ for all inputs a.

CLOSURE OPERATORS Given a partial order $(\leqslant)$, a function f is said to be a *closure* operator iff

$$(\leqslant) \cdot f = f^{\circ} \cdot (\leqslant) \cdot f \tag{5.94}$$

holds. Let us write the same with points — via (5.17) —, for all x, y:

$$y \leqslant f\ x\ \Leftrightarrow\ f\ y \leqslant f\ x \tag{5.95}$$

Clearly, for $(\geqslant) = (\leqslant)^{\circ}$, (5.94) can also be written

$$f^{\circ} \cdot (\geqslant) = f^{\circ} \cdot (\geqslant) \cdot f$$

Any of these alternatives is an elegant way of defining a closure operator f, in so far it can be shown to be equivalent to the conjunction of three facts about f: (a) f is monotonic; (b) $id\ \stackrel{.}{\leqslant}\ f$ and (c) $f = f \cdot f$.

As an example, consider the function that *closes* a finite set of natural numbers by filling in the intermediate numbers, e.g. $f\ \{4, 2, 6\} = \{2, 3, 4, 5, 6\}$. Clearly, $x\ \subseteq\ f\ x$. If you apply f again, you get

$$f\ \{2, 3, 4, 5, 6\} = \{2, 3, 4, 5, 6\}$$

This happens because f is a closure operator.

***Exercise** 5.25. Knowing that property*

$$f \cdot f^{\circ} \cdot f = f \tag{5.96}$$

holds for every function f, prove that $\mathsf{ker}\, f$ *(5.53) is an* equivalence *relation.*
□

Exercise 5.26. *From* $\mathsf{ker}\ ! = \top$ *and (5.96) infer*

$$\top \cdot R \subseteq \top \cdot S \quad \Leftrightarrow \quad R \subseteq \top \cdot S \tag{5.97}$$

Conclude that $(\top\cdot)$ *is a closure operator.*
□

Exercise 5.27. *Generalizing the previous exercise, show that pre/post-composition with functional kernels are* closure *operations:*

$$S \cdot \mathsf{ker}\, f \subseteq R \cdot \mathsf{ker}\, f \quad \equiv \quad S \subseteq R \cdot \mathsf{ker}\, f \tag{5.98}$$

$$\mathsf{ker}\, f \cdot S \subseteq \mathsf{ker}\, f \cdot R \quad \equiv \quad S \subseteq \mathsf{ker}\, f \cdot R \tag{5.99}$$

□

Exercise 5.28. *Consider the relation*

$$b\ R\ a \Leftrightarrow \text{team } b \text{ is playing against team } a$$

Is this relation: reflexive? irreflexive? transitive? anti-symmetric? symmetric? connected?
□

Exercise 5.29. *Expand criteria (5.86) to (5.90) to pointwise notation.*
□

Exercise 5.30. *A relation R is said to be co-transitive or dense iff the following holds:*

$$\langle \forall\, b, a\ :\ b\ R\ a :\ \langle \exists\, c\ :\ b\ R\ c :\ c\ R\ a \rangle \rangle \tag{5.100}$$

Write the formula above in PF notation. Find a relation (e.g. over numbers) which is co-transitive and another which is not.
□

Exercise 5.31. *Check which of the following properties,*

transitive, symmetric, anti-symmetric, connected *hold for the relation Eats (5.75) of the Alcuin puzzle.*

□

***Exercise** 5.32. Show that (5.55) of exercise 5.12 amounts to forcing relation home · away° to be symmetric.*

□

5.14 RELATIONAL PAIRING

Recall from sections 2.8 and 2.9 that functions can be composed in parallel and in alternation, giving rise to so-called *products* and *coproducts*. Does a product diagram like (2.23),

$$\begin{tikzcd}
A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
& C \arrow[ul, "f"] \arrow[u, "\langle f,g \rangle"] \arrow[ur, "g"'] &
\end{tikzcd}$$

make sense when f e g are generalized to relations R and S? We start from definition (2.20),

$$\langle f,g \rangle\, c \overset{\mathrm{def}}{=} (f\,c, g\,c)$$

to try and see what such a generalization could mean. The relational, pointwise expression of function $\langle f,g \rangle$ is

$$y = \langle f,g \rangle\, c$$

which can be rephrased to $(a,b) = \langle f,g \rangle\ c$, knowing that $\langle f,g \rangle$ is of type $C \to A \times B$ in (2.23). We reason:

$$\begin{array}{ll}
& (a,b) = \langle f,g \rangle\ c \\
\equiv & \qquad \{\ \langle f,g \rangle\ c = (f\ c, g\ c);\ \text{equality of pairs}\ \} \\
& \left\{ \begin{array}{l} a = f\ c \\ b = g\ c \end{array} \right. \\
\equiv & \qquad \{\ y = f\ x \Leftrightarrow y\,f\,x\ \} \\
& \left\{ \begin{array}{l} a\,f\,c \\ b\,g\,c \end{array} \right. \\
\Box &
\end{array}$$

By in-lining the conjunction expressed by the braces just above, one gets

$$(a,b)\ \langle f,g \rangle\ c \Leftrightarrow a\,f\,c \wedge b\,g\,c$$

which proposes the generalization:

$$(a,b)\ \langle R,S\rangle\ c\ \Leftrightarrow\ a\ R\ c \wedge b\ S\ c \tag{5.101}$$

Recalling the projections $\pi_1\ (a,b) = a$ and $\pi_2\ (a,b) = b$, let us try and remove variables a, b and c from the above, towards a closed definition of $\langle R,S\rangle$:

$$\begin{array}{cl}
& (a,b)\ \langle R,S\rangle\ c\ \Leftrightarrow\ a\ R\ c \wedge b\ S\ c \\
\equiv & \qquad \{\ \pi_1\ (a,b) = a \text{ and } \pi_2\ (a,b) = b\ \} \\
& (a,b)\ \langle R,S\rangle\ c\ \Leftrightarrow\ \pi_1\ (a,b)\ R\ c \wedge \pi_2\ (a,b)\ S\ c \\
\equiv & \qquad \{\ \text{(5.17) twice}\ \} \\
& (a,b)\ \langle R,S\rangle\ c\ \Leftrightarrow\ (a,b)\ (\pi_1^\circ \cdot R)\ c \wedge (a,b)\ (\pi_2^\circ \cdot S)\ c \\
\equiv & \qquad \{\ \text{(5.56)}\ \} \\
& (a,b)\ \langle R,S\rangle\ c\ \Leftrightarrow\ (a,b)\ (\pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S)\ c \\
\equiv & \qquad \{\ \text{(5.19)}\ \} \\
& \langle R,S\rangle = \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S
\end{array} \tag{5.102}$$

We proceed to investigating what kind of universal property $\langle R,S\rangle$, defined by $\pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S$, satisfies. The strategy is to use indirect equality:

$$\begin{array}{cl}
& X \subseteq \langle R,S\rangle \\
\equiv & \qquad \{\ \text{(5.102)}\ \} \\
& X \subseteq \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S \\
\equiv & \qquad \{\ \text{(5.58)}\ \} \\
& \left\{ \begin{array}{l} X \subseteq \pi_1^\circ \cdot R \\ X \subseteq \pi_2^\circ \cdot S \end{array} \right. \\
\equiv & \qquad \{\ \text{shunting}\ \} \\
& \left\{ \begin{array}{l} \pi_1 \cdot X \subseteq R \\ \pi_2 \cdot X \subseteq S \end{array} \right.
\end{array}$$

In summary, the universal property of $\langle R,S\rangle$ is:

$$X \subseteq \langle R,S\rangle \quad \Leftrightarrow \quad \left\{ \begin{array}{l} \pi_1 \cdot X \subseteq R \\ \pi_2 \cdot X \subseteq S \end{array} \right. \tag{5.103}$$

For functions, $X, R.S := k, f, g$ it can be observed that (5.103) coincides with (2.63). But otherwise, the corollaries derived from (5.103) are different from those that emerge from (2.63). For instance, cancellation becomes:

$$\left\{ \begin{array}{l} \pi_1 \cdot \langle R,S\rangle \subseteq R \\ \pi_2 \cdot \langle R,S\rangle \subseteq S \end{array} \right.$$

This tells us that pairing R with S has the (side) effect of deleting from R all those inputs for which S is undefined (and vice-versa), since output pairs require that *both* relations respond to the input. Thus, for relations, laws such as the $\times$*-fusion* rule (2.26) call for a side-condition:

$$\begin{array}{l} \langle R, S \rangle \cdot T = \langle R \cdot T, S \cdot T \rangle \\ \qquad \Leftarrow \; R \cdot (\mathsf{img}\ T) \subseteq R \vee S \cdot (\mathsf{img}\ T) \subseteq S \end{array} \tag{5.104}$$

Clearly,

$$\langle R, S \rangle \cdot f \;=\; \langle R \cdot f, S \cdot f \rangle \tag{5.105}$$

holds, since $\mathsf{img}\, f \;\subseteq\; id$. Moreover, the *absorption* law (2.27) remains unchanged,

$$(R \times S) \cdot \langle P, Q \rangle = \langle R \cdot P, S \cdot Q \rangle \tag{5.106}$$

where $R \times S$ is defined in the same way as for functions:

$$R \times S = \langle R \cdot \pi_1, S \cdot \pi_2 \rangle \tag{5.107}$$

As generalization of (5.105) and also immediate by monotonicity,

$$\langle R, S \rangle \cdot T = \langle R \cdot T, S \cdot T \rangle$$

holds for T simple.

Because (5.103) is not the universal property of a product, we tend to avoid talking about relational *products* and rather talk about relational *pairing* instead.[15] In spite of the weaker properties, relational pairing has interesting laws, namely

$$\langle R, S \rangle^\circ \cdot \langle X, Y \rangle \;\;=\;\; (R^\circ \cdot X) \cap (S^\circ \cdot Y) \tag{5.108}$$

that will prove quite useful later on.

Exercise *5.33. Derive from (5.108) the following properties:*

$$\frac{f}{g} \cap \frac{h}{k} = \frac{\langle f, h \rangle}{\langle g, k \rangle} \tag{5.109}$$

$$\tag{5.110}$$

$$\mathsf{ker}\ \langle R, S \rangle \;=\; \mathsf{ker}\ R \cap \mathsf{ker}\ S \tag{5.111}$$

$\langle R, id \rangle$ *is always* injective*, for whatever* R

□

Exercise *5.34. Recalling (5.38), prove that* $\mathsf{swap} = \langle \pi_2, \pi_1 \rangle$ *(2.32) is its own converse and therefore a bijection.*

□

[15] Relational products do exist but are not obtained by $\langle R, S \rangle$. For more about this see section 5.23 later on.

***Exercise** 5.35. Derive from the laws studied thus far the following facts about relational pairing:*

$$id \times id = id \tag{5.112}$$

$$(R \times S) \cdot (P \times Q) = (R \cdot P) \times (S \cdot Q) \tag{5.113}$$

□

5.15 RELATIONAL COPRODUCTS

Let us now show that, in contrast with products, coproducts extend perfectly from functions to relations, that is, universal property (2.65) extends to

$$X = [R, S] \quad \Leftrightarrow \quad \left\{ \begin{array}{l} X \cdot i_1 = R \\ X \cdot i_2 = S \end{array} \right. \tag{5.114}$$

where $X : A + B \to C$, $R : A \to C$ and $S : B \to C$ are binary relations. First of all, we need to understand what $[R, S]$ means. Our starting point is $+$-cancellation, recall (2.40):

$$\begin{array}{ll}
& \left\{ \begin{array}{l} [g, h] \cdot i_1 = g \\ [g, h] \cdot i_2 = h \end{array} \right. \\
\equiv & \qquad \{ \text{ equality of functions } \} \\
& \left\{ \begin{array}{l} g \subseteq [g, h] \cdot i_1 \\ h \subseteq [g, h] \cdot i_2 \end{array} \right. \\
\equiv & \qquad \{ \text{ shunting followed by (5.57) } \} \\
& g \cdot i_1^\circ \cup h \cdot i_2^\circ \ \subseteq \ [g, h]
\end{array}$$

On the other hand:

$$\begin{array}{ll}
& \left\{ \begin{array}{l} [g, h] \cdot i_1 = g \\ [g, h] \cdot i_2 = h \end{array} \right. \\
\equiv & \qquad \{ \text{ equality of functions } \} \\
& \left\{ \begin{array}{l} [g, h] \cdot i_1 \subseteq g \\ [g, h] \cdot i_2 \subseteq h \end{array} \right. \\
\Rightarrow & \qquad \{ \text{ monotonicity } \} \\
& \left\{ \begin{array}{l} [g, h] \cdot i_1 \cdot i_1^\circ \subseteq g \cdot i_1^\circ \\ [g, h] \cdot i_2 \cdot i_2^\circ \subseteq h \cdot i_2^\circ \end{array} \right. \\
\Rightarrow & \qquad \{ \text{ monotonicity (5.81) and distribution (5.60) } \} \\
& [g, h] \cdot (i_1 \cdot i_1^\circ \cup i_2 \cdot i_2^\circ) \ \subseteq \ g \cdot i_1^\circ \cup h \cdot i_2^\circ
\end{array}$$

$$
\begin{array}{ll}
\equiv & \{\ \mathsf{img}\ i_1 \cup \mathsf{img}\ i_2 = id, \text{ more about this below }\} \\
 & [g,h] \ \subseteq\ g \cdot i_1^\circ \cup h \cdot i_2^\circ
\end{array}
$$

Altogether, we obtain:

$$[g,h] = g \cdot i_1^\circ \cup h \cdot i_2^\circ$$

Note how this matches with (2.37), once variables are introduced:

$$c\ [g,h]\ x \ \Leftrightarrow\ \langle \exists\, a\ :\ x = i_1\, a:\ c = g\, a\rangle \vee \langle \exists\, b\ :\ x = i_2\, b:\ c = h\, b\rangle$$

Fact

$$\mathsf{img}\ i_1 \cup \mathsf{img}\ i_2 = id \tag{5.115}$$

assumed above is a property stemming from the construction of coproducts,

$$A + B \ \stackrel{\mathrm{def}}{=}\ \{i_1\, a \mid a \in A\} \cup \{i_2\, b \mid b \in B\}$$

since i_1 and i_2 are the *only* constructors of data of type $A + B$. Another property implicit in this construction is:

$$i_1^\circ \cdot i_2 = \bot \tag{5.116}$$

equivalent to its converse $i_2^\circ \cdot i_1 = \bot$. It spells out that, for any $a \in A$ and $b \in B$, $i_1\ a = i_2\ b$ is impossible.[16] In other words, the union is a *disjoint* one.

Let us now generalize the above to relations instead of functions,

$$[R,S] = R \cdot i_1^\circ \cup S \cdot i_2^\circ \tag{5.117}$$

and show that (5.114) holds. First of all,

$$
\begin{array}{ll}
 & X = R \cdot i_1^\circ \cup S \cdot i_2^\circ \\
\Rightarrow & \{\ \text{compose both sides with } i_1 \text{ and simplify; similarly for } i_2\ \} \\
 & X \cdot i_1 = R \wedge X \cdot i_2 = S
\end{array}
$$

The simplifications arise from i_1 and i_2 being injections, so their kernels are identities. On the other hand, $i_1^\circ \cdot i_2 = \bot$ and $i_2^\circ \cdot i_1 = \bot$, as seen above. The converse implication ($\Leftarrow$) holds:

$$
\begin{array}{ll}
 & X = R \cdot i_1^\circ \cup S \cdot i_2^\circ \\
\equiv & \{\ (5.115)\ \} \\
 & X \cdot (\mathsf{img}\ i_1 \cup \mathsf{img}\ i_2) = R \cdot i_1^\circ \cup S \cdot i_2^\circ \\
\equiv & \{\ \text{distribution}\ \} \\
 & X \cdot \mathsf{img}\ i_1 \cup X \cdot \mathsf{img}\ i_2 = R \cdot i_1^\circ \cup S \cdot i_2^\circ \\
\Leftarrow & \{\ \text{Leibniz}\ \} \\
 & X \cdot i_1 \cdot i_1^\circ = R \cdot i_1^\circ \wedge X \cdot i_2 \cdot i_2^\circ = S \cdot i_2^\circ \\
\Leftarrow & \{\ \text{monotonicity}\ \} \\
 & X \cdot i_1 = R \wedge X \cdot i_2 = S
\end{array}
$$

□

[16] Note that in (2.36) this is ensured by always choosing two different tags $t_1 \neq t_2$.

Thus (5.114) holds in general, for relations:

$$\begin{tikzcd}
(B+C) \to A \arrow[rr, bend left, "{[\_,\_]^\circ}"] \arrow[rr, phantom, "\cong"] & & (B \to A) \times (C \to A) \arrow[ll, bend left, "{[\_,\_]}"]
\end{tikzcd} \qquad (5.118)$$

A most useful consequence of this is that all results known for coproducts of functions are valid for relational coproducts. In particular, relational direct sum

$$R + S = [i_1 \cdot R\, , i_2 \cdot S] \qquad (5.119)$$

can be defined satisfying (2.43), (2.44) etc with relations replacing functions. Moreover, the McCarthy conditional (2.70) can be extended to relations in the expected way:[17]

$$p \to R, S \stackrel{\mathrm{def}}{=} [R, S] \cdot p? \qquad (5.120)$$

The property for sums (coproducts) corresponding to (5.108) for products is:

$$[R, S] \cdot [T, U]^\circ = (R \cdot T^\circ) \cup (S \cdot U^\circ) \qquad (5.121)$$

This *divide-and-conquer* rule is essential to *parallelizing* relation composition by so-called *block* decomposition.

Finally, the *exchange law* (2.49) extends to relations,

$$[\langle R, S\rangle\, , \langle T, V\rangle] = \langle [R\, , T], [S\, , V]\rangle \qquad (5.122)$$

cf.

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[d, "R"'] \arrow[rrd, "S"'] & A + B & B \arrow[l, "i_2"'] \arrow[d, "V"] \arrow[lld, "T"] \\
C & C \times D \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & D
\end{tikzcd}$$

For the proof see the following exercise.

Exercise 5.36. *Relying on both (5.114) and (5.105) prove (5.122). Moreover, prove*

$$(R + S)^\circ = R^\circ + S^\circ \qquad (5.123)$$

□

Exercise 5.37. *From (5.117) prove (5.121). Then show that*

$$\mathsf{img}\ [R, S] = \mathsf{img}\ R \cup \mathsf{img}\ S \qquad (5.124)$$

follows immediately from (5.121).

□

[17] Guards $p?$ will be expressed relationally in section 5.21.

Exercise *5.38. Derive the following inequational counterpart of (5.114)*

$$[R,S] \subseteq X \quad \equiv \quad R \subseteq X \cdot i_1 \wedge S \subseteq X \cdot i_2 \tag{5.125}$$

from (5.117) by indirect equality.

□

Exercise *5.39. Two relations R and S are said to be* separated *wherever*

$$R \cdot S^\circ = \bot$$

and co-separated *wherever*

$$R^\circ \cdot S = \bot$$

holds. Show that co-separability is necessary for the coproduct of two injective relations R and S to be injective:

$$id \leqslant [R,S] \quad \Leftrightarrow \quad id \leqslant R \wedge id \leqslant S \wedge R^\circ \cdot S = \bot \tag{5.126}$$

□

Exercise *5.40. Prove:*

$$\frac{f}{g} \times \frac{h}{k} = \frac{f \times h}{g \times k} \tag{5.127}$$

$$\frac{f}{g} + \frac{h}{k} = \frac{f + h}{g + k} \tag{5.128}$$

□

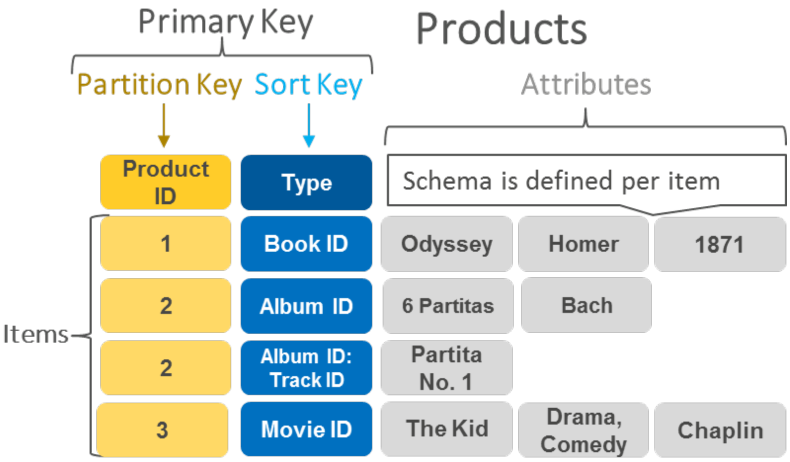
5.16 ON KEY-VALUE DATA MODELS

Simple relations abstract what is currently known as the *key-value-pair* data model in modern databases.[18] In this setting, given a *simple* relation $K \xrightarrow{S} V$, K is regarded as a type of data *keys* and V as a type of data *values*.

By pairing (5.102) such key-value-pairs one obtains more elaborate stores. Conversely, one may use projections to select particular key-attribute relationships from key-value stores. Note that keys and values can be *anything* (that is, of any type) and, in particular, they can be compound, for instance

$$\underbrace{PartitionKey \times SortKey}_{K} \to \underbrace{Type \times \ldots}_{V}$$

[18] For example, Hbase, Amazon DynamoDB, and so on, are examples of database systems that use the key-value pair data model.



Product ID	Type	Schema is defined per item		
1	Book ID	Odyssey	Homer	1871
2	Album ID	6 Partitas	Bach	
2	Album ID: Track ID	Partita No. 1		
3	Movie ID	The Kid	Drama, Comedy	Chaplin

Figure 5.1.: Key-value data model instance.

in the example of figure 5.1.[19]

The example furthermore shows how keys and values can structure themselves even further. In particular, "*schema is defined per item*" indicates that the values may be of coproduct types, something like $Title \times (1 + Author \times (1 + Date \times \ldots))$, for instance. Although the simplicity of the columnar model suggested by the key-value principle is somewhat sacrificed in the example, this shows how expressive *simple* relations involving *product* and *coproduct* types are.

One of the standard variations of the key-value model is to equip keys with time-stamps indicating *when* the pair was inserted or modified in the store, for instance

$$Student \times Course \times Time \rightarrow Result \tag{5.129}$$

telling the possibly different results of students in exams of a particular course. This combination of the key-value model with that of *temporal* (also called *historical*) databases is very powerful.

The relational combinators studied in this book apply naturally to key-value-pair storage processing and offer themselves as a powerful, pointfree high-level language for handling such data in a "noSQL" style.

5.17 WHAT ABOUT RELATIONAL "CURRYING"?

Recall isomorphism (2.95),

$$\begin{tikzcd}
(C^B)^A \arrow[r, bend left, "\mathsf{uncurry}"] \arrow[r, phantom, "\cong"] & C^{A \times B} \arrow[l, bend left, "\mathsf{curry}"]
\end{tikzcd}$$

[19] Credits: `https://aws.amazon.com/nosql/key-value/`.

that is at the core of the way functions are handled in functional programming. Does this isomorphism hold when functions are generalized to relations, something like...

$$A \times B \to C \quad \cong \quad A \to \ldots ?$$

Knowing that the type $A \times B \to C$ of relations is far larger than $C^{A \times B}$, it can be anticipated that the isomorphism will not extend to relations in the same way. In fact, a rather simpler one happens instead, among relations:

$$\begin{tikzcd} A \times B \to C \arrow[rr, bend left, "trans"] \arrow[rr, phantom, "\cong"] & & A \to C \times B \arrow[ll, bend left, "untrans"] \end{tikzcd} \qquad (5.130)$$

This tells us the (obvious, but very useful) fact that relations involving product types can be reshaped in any way we like, leftwards or rightwards.

It is quite convenient to overload the notation used for functions and write $\overline{R}$ to denote *trans* R and $\widehat{R}$ to denote *untrans* R. Then the isomorphism above is captured by universal property,[20]

$$\begin{tikzcd} C \times B & (C \times B) \times B \arrow[r, "\epsilon"] & C \\ A \arrow[u, "\overline{R}"] & A \times B \arrow[u, "\overline{R} \times id"] \arrow[ur, "R"'] & \end{tikzcd}$$

where

$$\overline{R} \;=\; \langle R, \pi_2 \rangle \cdot \pi_1^\circ \qquad \begin{tikzcd} C \times B & \\ A \arrow[u, "\overline{R}"] \arrow[r, "\pi_1^\circ"'] & A \times B \arrow[ul, "\langle R, \pi_2 \rangle"'] \end{tikzcd} \qquad (5.131)$$

that is

$$(c, b)\ \overline{R}\ a \quad \equiv \quad c\ R\ (a, b)$$

Moral: every n-ary relation can be expressed as a binary relation; moreover, where each particular attribute is placed (input/output) is irrelevant.

By *converse duality*, $(\widehat{S})^\circ = \overline{(S^\circ)}$, we obtain the definition of relational "uncurrying":

$$\widehat{S} \quad = \quad \pi_1 \cdot \langle S^\circ, \pi_2 \rangle^\circ$$

Then

$$\epsilon = \widehat{id} = \pi_1 \cdot \langle id, \pi_2 \rangle^\circ .$$

With points:

$$c_2\, \epsilon\, ((c_1, b_1), b_2) \quad \equiv \quad c_2 = c_1 \wedge b_1 = b_2$$

[20] Compare with (2.84).

THE "PAIRING WHEEL" RULE The flexibility offered by (5.130) means that, in relation algebra, the information altogether captured by the three relations M, P and Q in

$$\begin{tikzcd}
& B & \\
& A \arrow[u, "M"'] \arrow[dl, "Q"'] \arrow[dr, "P"] & \\
C & & D
\end{tikzcd} \tag{5.132}$$

can be aggregated in several ways, namely

$$B \xrightarrow{\langle P,Q\rangle\cdot M^\circ} D \times C$$
$$D \xrightarrow{\langle Q,M\rangle\cdot P^\circ} C \times B$$
$$C \xrightarrow{\langle M,P\rangle\cdot Q^\circ} B \times C$$

all isomorphic to each other:

$$\begin{tikzcd}
& B \to D \times C \arrow[dr, bend left, "\alpha"] & \\
C \to B \times D \arrow[ur, bend left, "\alpha"] & & D \to C \times B \arrow[ll, bend left, "\alpha"]
\end{tikzcd}$$

The rotation among relations and types justifies the name "pairing wheel" given to (5.132). Isomorphism α holds in the sense that every entry of one of the aggregates is uniquely represented by another entry in any other aggregate, for instance:

$$\begin{array}{cl}
& (d,c)\ (\langle P,Q\rangle \cdot M^\circ)\ b \\
= & \qquad \{\ \text{composition ; pairing}\ \} \\
& \langle \exists\, a\ :\ d\, P\, a \wedge c\, Q\, a :\ a\, M^\circ\ b \rangle \\
= & \qquad \{\ \text{converse; } \wedge \text{ is associative and commutative}\ \} \\
& \langle \exists\, a\ ::\ (c\, Q\, a \wedge b\, M\, a) \wedge a\, P^\circ\ d \rangle \\
= & \qquad \{\ \text{composition ; pairing}\ \} \\
& (c,b)\ (\langle Q,M\rangle \cdot P^\circ)\ d
\end{array}$$

Thus: $\alpha\ (\langle P,Q\rangle \cdot M^\circ) = (\langle Q,M\rangle) \cdot P^\circ$.

Exercise *5.41. Express α in terms of trans (5.130) and its converse (5.131).*
□

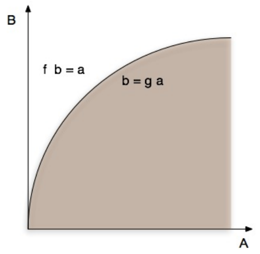


Figure 5.1.: Graphical interpretation of equation (5.133): (a) relation $B \xleftarrow{(\leqslant)\cdot g} A$ is the "area" below function g wrt. $\leqslant$; (b) relation $B \xrightarrow{f^\circ\cdot(\sqsubseteq)} A$ is the "area" above function f wrt. $\sqsubseteq$, to the right (oriented 90°); (c) f and g are such that these areas are the same.

5.18 GALOIS CONNECTIONS

Recall from section 5.13 that a preorder is a reflexive and transitive relation. Given two preorders $\leqslant$ and $\sqsubseteq$, one may relate arguments and results of pairs of suitably typed functions f and g in a particular way,

$$f^\circ\cdot \sqsubseteq \quad = \quad \leqslant \cdot\, g \tag{5.133}$$

as in the diagram:

$$\begin{tikzcd}
A \arrow[d, "f^\circ"'] & A \arrow[l, "\sqsubseteq"'] \arrow[d, "g"] \arrow[dl, phantom, "="] \\
B & B \arrow[l, "\leqslant"]
\end{tikzcd}$$

In this very special situation, f, g are said to be *Galois connected*. We write

$$f \vdash g \tag{5.134}$$

as abbreviation of (5.133) when the two preorders $\sqsubseteq, \leqslant$ are implicit from the context. Another way to represent this is:

$$\begin{tikzcd}
(A,\sqsubseteq) \arrow[r, bend left, "g"] & (B,\leqslant) \arrow[l, bend left, "f"]
\end{tikzcd}$$

Function f (resp. g) is referred to as the *left* (resp. *right*) adjoint of the connection. By introducing variables in both sides of (5.133) via (5.17), we obtain, for all x and y

$$(f\ x) \sqsubseteq y \quad \equiv \quad x \leqslant (g\ y) \tag{5.135}$$

$$(f\,X) \subseteq Y \equiv X \subseteq (g\,Y)$$

Description	f	g	**Obs.**
converse	$(\_)^\circ$	$(\_)^\circ$	
shunting rule	$(h\cdot)$	$(h^\circ\cdot)$	h is a function
"converse" *shunting* rule	$(\cdot h^\circ)$	$(\cdot h)$	h is a function
difference	$(\_ - R)$	$(R \cup\ )$	
implication	$(R \cap \_)$	$(R \Rightarrow \_)$	

Table 3.: Sample of Galois connections in the relational calculus. The general formula given on top is a logical equivalence universally quantified on S and R. It has a left part involving *left adjoint* f and a right part involving *right adjoint* g.

In particular, the two preorders in (5.133) can be the identity id, in which case (5.133) reduces to $f^\circ = g$, that is, f and g are each-other inverses — i.e., isomorphisms. Therefore, the Galois connection concept is a generalization of the concept of isomorphism.[21]

Quite often, the two adjoints are *sections* of binary operators. Recall that, given a binary operator $a\,\theta\,b$, its two sections $(a\theta)$ and (θb) are unary functions f and g such that, respectively:

$$f = (a\theta) \quad \equiv \quad f\,b = a\,\theta\,b \tag{5.136}$$

$$g = (\theta b) \quad \equiv \quad g\,a = a\,\theta\,b \tag{5.137}$$

Galois connections in which the two preorders are relation inclusion ($\leqslant, \sqsubseteq \;:=\; \subseteq, \subseteq$) and whose adjoints are sections of relational combinators are particularly interesting because they express universal properties about such combinators. Table 3 lists some connections that are relevant for this book.

It is remarkably easy to recover known properties of the relation calculus from table 3. For instance, the first row yields

$$X^\circ \subseteq Y \quad \equiv \quad X \subseteq Y^\circ \tag{5.138}$$

since $f = g = (\_)^\circ$ in this case. Thus converse is its own self adjoint. From this we derive

$$R \subseteq S \equiv R^\circ \subseteq S^\circ \tag{5.139}$$

by making $X, Y := R, S^\circ$ and simplifying by involution (5.15). Moreover, the entry marked "*shunting* rule" in the table leads to

$$h \cdot X \subseteq Y \quad \equiv \quad X \subseteq h^\circ \cdot Y$$

[21] Interestingly, every Galois connection is on its turn a special case of an *adjunction*, reall (4.58). Just promote the adjoints f and g in (5.135) to functors, and replace the preorder symbols by arrows. This "syntactic trick" can be taken as a rough sketch of a formal, categorial argument that we shall skip for the time being.

for all h, X and Y. By taking converses, one gets another entry in table 3, namely

$$X \cdot h^{\circ} \subseteq Y \quad \equiv \quad X \subseteq Y \cdot h$$

These are the equivalences (5.46) and (5.47) that we have already met, popularly known as "shunting rules".

The fourth and fifth rows in the table are Galois connections that respectively introduce two new relational operators — relational *difference* $S - R$ and relational *implication* $R \Rightarrow S$ — as a *left adjoint* and an *right adjoint*, respectively:

$$X - R \subseteq Y \quad \equiv \quad X \subseteq Y \cup R \tag{5.140}$$

$$R \cap X \subseteq Y \quad \equiv \quad X \subseteq R \Rightarrow Y \tag{5.141}$$

There are *many* advantages in describing the meaning of relational operators by Galois connections. Further to the systematic tabulation of operators (of which table 3 is just a sample), the concept of a Galois connection is a *generic* one, which offers a rich algebra of *generic* properties, namely:

- both adjoints f and g in a Galois connection are monotonic;
- left adjoint f distributes with join and right-adjoint g distributes with meet, wherever these exist:

$$f(b \sqcup b') \quad = \quad (f\ b) \vee (f\ b') \tag{5.142}$$

$$g(a \wedge a') \quad = \quad (g\ a) \sqcap (g\ a') \tag{5.143}$$

- left adjoint f preserves infima and right-adjoint g preserves suprema, wherever these exist:[22]

$$f\ \bot = \bot \tag{5.144}$$

$$g\ \top = \top \tag{5.145}$$

- two cancellation laws hold,

$$(f \cdot g)a \leqslant a \quad \text{and} \quad b \sqsubseteq (g \cdot f)b \tag{5.146}$$

respectively known as *left-cancellation* and *right-cancellation*.

- Semi-inverse properties:

$$f = f \cdot g \cdot f \tag{5.147}$$

$$g = g \cdot f \cdot g \tag{5.148}$$

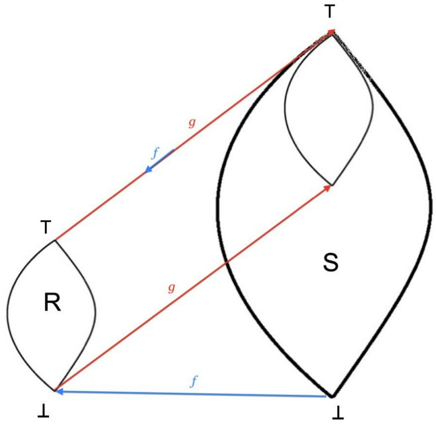


Figure 5.2.: Left-perfect Galois connection $f \vdash g$ involving two lattices S and R.

It may happen that a cancellation law holds up to equality, for instance $f\ (g\ a) = a$, in which case the connection is said to be *perfect* on the particular side. The picture of a left-perfect Galois connection $f \vdash g$ is given in figure 5.2.[23]

Let us take for instance Galois connection (5.140) as example. Following the general rules above, we get *for free*: the monotonicity of $(\_ - R)$,

$$X \subseteq Z \quad \Rightarrow \quad X - R \subseteq Z - R$$

the monotonicity of $(\_ \cup R)$,

$$X \subseteq Z \quad \Rightarrow \quad X \cup R \subseteq Z \cup R$$

the distribution of $(\_ - R)$ over *join*,

$$(X \cup Y) - R \quad = \quad (X - R) \cup (Y - R)$$

the distribution of $(\_ \cup R)$ over *meet*,

$$(X \cap Y) \cup R \quad = \quad (X \cup R) \cap (Y \cup R)$$

the preservation of infima by $(\_ - R)$,

$$\bot - R = \bot$$

the preservation of suprema by $(\_ \cup R)$,

$$\top \cup R = \top$$

left-cancellation $(Y := X - R)$,

$$X \subseteq (X - R) \cup R$$

[22] In these case both orders will form a so-called *lattice* structure.
[23] Adapted from [5].

right-cancellation ($X := Y \cup R$),

$$(Y \cup R) - R \subseteq Y$$

and finally the semi-inverse properties:

$$X - ((X - R) \cup R) = X - R$$
$$((X \cup R) - R) \cup R = X \cup R$$

The reader is invited to extract similar properties from the other connections listed in table 3. Altogether, we get 50 properties out of this table! Such is the power of *generic* concepts in mathematics.

Two such connections were deliberately left out from table 3, which play a central role in relation algebra and will deserve a section of their own — section 5.19.

***Exercise** 5.42. Show that $R - S \subseteq R$, $R - \bot = R$ and $R - R = \bot$ hold.*

□

***Exercise** 5.43. Infer*

$$b(R \Rightarrow S)a \quad \equiv \quad (b\ R\ a) \Rightarrow (b\ S\ a) \tag{5.149}$$

from the Galois connection

$$R \cap X \subseteq Y \quad \equiv \quad X \subseteq (R \Rightarrow Y) \tag{5.150}$$

Suggestion: note that $b\ (R \Rightarrow S)\ a$ can be written $id \subseteq \underline{b}^\circ \cdot (R \Rightarrow S) \cdot \underline{a}$ (check this!). Then proceed with (5.150) and simplify.

□

***Exercise** 5.44. (Lexicographic orders) The* lexicographic chaining *of two relations R and S is defined by:*

$$R\,;S \quad = \quad R \cap (R^\circ \Rightarrow S) \tag{5.151}$$

Show that (5.151) is the same as stating the universal property:

$$X \subseteq (R;S) \equiv X \subseteq R \wedge X \cap R^\circ \subseteq S$$

□

***Exercise** 5.45. Let students in a course have two numeric marks,*

$$\mathbb{N}_0 \xleftarrow{mark1} Student \xrightarrow{mark2} \mathbb{N}_0$$

and define the preorders*:*

$$\leqslant_{mark1} \quad = \quad mark1^\circ \cdot \leqslant \cdot\, mark1$$
$$\leqslant_{mark2} \quad = \quad mark2^\circ \cdot \leqslant \cdot\, mark2$$

Spell out in pointwise notation the meaning of lexicographic ordering

$$\leqslant_{mark1}\,;\leqslant_{mark2}$$

□

NEGATION We define $\neg R = R \Rightarrow \bot$, leading to the pointwise meaning $b\ (\neg R)\ a \ \Leftrightarrow\ \neg\ (b\ R\ a)$. From the Galois connection of $R \Rightarrow S$ one immediately derives that of negation,

$$X \subseteq \neg R \quad \Leftrightarrow \quad \neg X \supseteq R$$

cf.

$$\begin{array}{ll}
& X \subseteq \neg R \\
\equiv & \quad \{ \neg R = R \Rightarrow \bot \} \\
& X \subseteq (R \Rightarrow \bot) \\
\equiv & \quad \{ \text{(5.150)} \} \\
& X \cap R \subseteq \bot \\
\equiv & \quad \{ X \cap R = R \cap X; \text{(5.150)} \} \\
& R \subseteq (X \Rightarrow \bot) \\
\equiv & \quad \{ \neg R = R \Rightarrow \bot \} \\
& R \subseteq \neg X \\
\Box &
\end{array}$$

Note the order reversal, $\subseteq$ on the lefthand side, $\supseteq$ on the other side. Thus the following instances of (5.142,5.143),

$$\begin{aligned}
\neg(R \cup S) &= (\neg R) \cap (\neg S) \qquad (5.152)\\
\neg(R \cap S) &= (\neg R) \cup (\neg S)
\end{aligned}$$

known as *de Morgan laws*.

From the Galois connection above other expected properties analogous to logic negation can be derived, for instance $\neg\top = \bot$. One of the most famous rules for handing negated relations is the so-called *Schröder's rule*:

$$\neg Q \cdot S^\circ \subseteq \neg R \Leftrightarrow R^\circ \cdot \neg Q \subseteq \neg S \qquad (5.153)$$

It can also be shown that

$$R \cup \neg R = \top \qquad (5.154)$$

holds and therefore:

$$\top - R \subseteq R \Rightarrow \bot \qquad (5.155)$$

Exercise *5.46. Show that* $\neg(R^\circ) = (\neg R)^\circ$.

□

Exercise *5.47. Assuming*

$$f^\circ \cdot (R \Rightarrow S) \cdot g \quad = \quad (f^\circ \cdot R \cdot g) \Rightarrow (f^\circ \cdot S \cdot g) \tag{5.156}$$

and (5.155), prove:

$$\underline{c}^\circ \cdot (\top - \underline{c}) = \bot \tag{5.157}$$

□

5.19 RELATION DIVISION

However intimidating it may sound, structuring a calculus in terms of Galois connections turns out to be a great simplification, leading to *rules* that make the reasoning closer to school algebra. Think for instance the rule used at school to reason about whole division of two natural numbers x and y,

$$z \times y \leqslant x \ \equiv \ z \leqslant x \div y \qquad (y > 0) \tag{5.158}$$

assumed universally quantified in all its variables. Pragmatically, it expresses a "shunting" rule which enables one to trade between a whole division in the upper side of an inequality and a multiplication in the lower side. This rule is easily identified as the Galois connection

$$z \underbrace{(\times y)}_{f} \leqslant x \ \Leftrightarrow \ z \leqslant x \underbrace{(\div y)}_{g}.$$

where multiplication is the left adjoint and division is the right adjoint: $(\times y) \vdash (\div y)$, for $y \neq 0$.[24]

As seen in the previous section, many properties of $(\times)$ and $(\div)$ can be inferred from (5.158), for instance the cancellation $(x \div y) \times y \leqslant x$ — just replace z by $x \div y$ and simplify, and so on.

A parallel with relation algebra could be made by trying a rule similar to (5.158),

$$Z \cdot Y \subseteq X \ \equiv \ Z \subseteq X/Y \tag{5.159}$$

which suggests that, like integer multiplication, relational composition has an right adjoint, denoted $X\ /\ Y$. The question is: does such a

[24] This connection is perfect on the lower side since $(z \times y) \div y = z$.

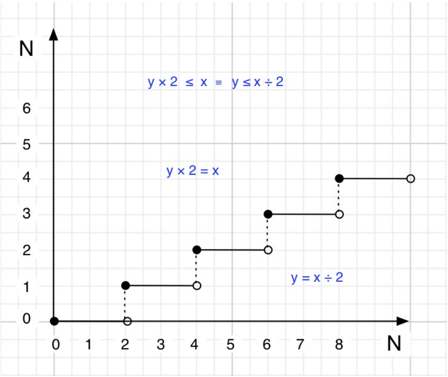


Figure 5.1.: Picturing Galois connection $(\times 2) \vdash (\div 2)$ as in figure 5.1. $f = (\times 2)$ is the left adjoint of $g = (\div 2)$. The area below $g = (\div 2)$ is the same as the area above $f = (\times 2)$. $f = (\times 2)$ is not surjective. $g = (\div 2)$ is not injective.

relation division operator actually exist? Proceeding with the parallel, note that, in the same way

$$z \times y \leqslant x \;\equiv\; z \leqslant x \div y$$

means that $x \div y$ is the largest *number* which multiplied by y approximates x, (5.159) means that X/Y is the largest *relation* Z which, precomposed with Y, approximates X.

What is the pointwise meaning of X/Y? Let us first of all equip (5.159) with a type diagram:

$$Z \cdot Y \subseteq X \;\equiv\; Z \subseteq X/Y$$

$$\begin{tikzcd}
 & A \arrow[dl, "X/Y"'] \\
C & B \arrow[l, "X"] \arrow[u, "Y"']
\end{tikzcd}$$

Then we calculate:[25]

$$c\,(X/Y)\,a$$

$$\equiv \quad \{ \text{ introduce points } C \xleftarrow{\;c\;} 1 \text{ and } A \xleftarrow{\;a\;} 1 \text{ ; (5.17) } \}$$

$$x(\underline{c}^{\circ} \cdot (X/Y) \cdot \underline{a})x$$

$$\equiv \quad \{ \text{ } \forall\text{-one-point (A.5) } \}$$

$$x' = x \;\Rightarrow\; x'(\underline{c}^{\circ} \cdot (X/Y) \cdot \underline{a})x$$

$$\equiv \quad \{ \text{ go pointfree (5.19) } \}$$

$$id \;\subseteq\; \underline{c}^{\circ} \cdot (X/Y) \cdot \underline{a}$$

$$\equiv \quad \{ \text{ shunting rules } \}$$

$$\underline{c} \cdot \underline{a}^{\circ} \;\subseteq\; X/Y$$

$$\equiv \quad \{ \text{ universal property (5.159) } \}$$

[25] Following the strategy suggested in exercise 5.43.

$$
\begin{array}{ll}
 & \underline{c} \cdot \underline{a}^\circ \cdot Y \subseteq X \\
\equiv & \qquad \{ \text{ now shunt } \underline{c} \text{ back to the right } \} \\
 & \underline{a}^\circ \cdot Y \subseteq \underline{c}^\circ \cdot X \\
\equiv & \qquad \{ \text{ back to points via (5.17) } \} \\
 & \langle \forall\, b \ :\ a\, Y\, b :\ c\, X\, b \rangle
\end{array}
$$

In summary:

$$
c\,(X/Y)\,a \;\equiv\; \langle \forall\, b \ :\ a\, Y\, b :\ c\, X\, b \rangle \tag{5.160}
$$

$$\begin{tikzcd}
 & a \arrow[dl, "X/Y"'] \\
c & b \arrow[l, "X"] \arrow[u, "Y"']
\end{tikzcd}$$

In words: in the same way relation *composition* hides an *existential* quantifier (5.11), *relation* division (5.160) hides a *universal* one. Let us feel what (5.160) means through an example: let

$a\ Y\ b$ = passenger a choses flight b
$c\ X\ b$ = company c operates flight b

Then (5.160) yields : whenever a choses a flight b it turns out that b is operated by company c. So:

$c\ (X/Y)\ a$ = company c is the only one trusted by passenger a, that is, *a only flies c*.

Therefore, (5.159) captures, in a rather eloquent way, the duality between universal and existential quantification. It is no wonder, then, that the relational equivalent to $(x \div y) \times y \leqslant x$ above is

$$(X/S) \cdot S \subseteq X$$

This *cancellation* rule, very often used in practice, unfolds to

$$\langle \forall\, b \ :\ a\, S\, b :\ c\, X\, b \rangle \wedge a\, S\, b' \quad\Rightarrow\quad c\, X\, b'$$

i.e. to the well-known device in logic known as *modus ponens*: $((S \rightarrow X) \wedge S) \rightarrow X$.

There is one important difference between (5.158) and (5.159): while multiplication in (5.158) is commutative, and thus writing $z \times y$ or $y \times z$ is the same, writing $Z \cdot Y$ or $Y \cdot Z$ makes a lot of difference because composition is not commutative in general. The dual division operator is obtained by taking converses over (5.159):

$$
\begin{array}{ll}
 & Y \cdot Z \subseteq X \\
\equiv & \qquad \{ \text{ converses } \} \\
 & Z^\circ \cdot Y^\circ \subseteq X^\circ \\
\equiv & \qquad \{ \text{ division (5.159) } \} \\
 & Z^\circ \subseteq X^\circ \,/\, Y^\circ \\
\equiv & \qquad \{ \text{ converses } \} \\
 & Z \subseteq \underbrace{(X^\circ \,/\, Y^\circ)^\circ}_{Y \backslash X}
\end{array}
$$

In summary:

$$X \cdot Z \subseteq Y \;\Leftrightarrow\; Z \subseteq X \setminus Y \tag{5.161}$$

Once variables are added to $X \setminus Y$ we get:

$$a(X \setminus Y)c \quad\equiv\quad \langle \forall\, b \;:\; b\,X\,a :\; b\,Y\,c \rangle \tag{5.162}$$

Thus we are ready to add two more rows to table 3:

$$\boxed{(f\;X) \subseteq Y \equiv X \subseteq (g\;Y)}$$

Description	f	g	**Obs.**
Left-division	$(R\cdot)$	$(R \setminus\;)$	read "R under ..."
Right-division	$(\cdot R)$	$(\;/\,R)$	read "...over R"

As example of left division consider the relation $a \in x$ between a set x and each of its elements a:

$$A \xleftarrow{\;\in\;} \mathsf{P}A \tag{5.163}$$

Then inspect the meaning of relation $\mathsf{P}A \xleftarrow{\in \setminus \in} \mathsf{P}A$ using (5.162):

$$x_1 \;(\in \setminus \in)\; x_2 \qquad\Leftrightarrow\qquad \langle \forall\, a \;:\; a \in x_1 \;:\; a \in x_2 \rangle$$

We conclude that quotient $\mathsf{P}A \xleftarrow{\in \setminus \in} \mathsf{P}A$ expresses the inclusion relation among sets.

Relation division gives rise to a number of combinators in relation algebra that are very useful in problem specification. We review some of these below.

Exercise *5.48. Prove the equalities*

$$\begin{aligned}
R \cdot f &= R / f^{\circ} && (5.164)\\
f \setminus R &= f^{\circ} \cdot R && (5.165)\\
R / \bot &= \top && (5.166)\\
R / id &= R && (5.167)\\
R \setminus (f^{\circ} \cdot S) &= f \cdot R \setminus S && (5.168)\\
R \setminus \top \cdot S &= \,! \cdot R \setminus ! \cdot S && (5.169)\\
R \,/\, (S \cup P) &= R \,/\, S \cap R \,/\, P && (5.170)
\end{aligned}$$

□

Exercise *5.49. On June 23rd, 1991, E.W. Dijkstra wrote one of his famous notes — EWD1102-5 — entitled: "Why preorders are beautiful". The main result of his six page long manuscript is:*

> *A binary relation is a pre-order iff $R = R \,/\, R$ holds.*

The proof of this result becomes even shorter (and perhaps even more beautiful) once expressed in relation algebra. Fill in the ellipses in the following calculation of such a result:

$$
\begin{array}{ll}
 & R = R \,/\, R \\
\equiv & \qquad \{ \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad \} \\
 & \left\{ \begin{array}{l} X \subseteq R \Leftrightarrow X \cdot R \subseteq R \\ X \subseteq R \Leftrightarrow X \cdot R \subseteq R \end{array} \right. \\
\Rightarrow & \qquad \{ \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad \} \\
 & \left\{ \begin{array}{l} id \subseteq R \Leftrightarrow R \subseteq R \\ R \subseteq R \Leftrightarrow R \cdot R \subseteq R \end{array} \right. \\
\equiv & \qquad \{ \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad \} \\
 & \left\{ \begin{array}{l} id \subseteq R \\ R \cdot R \subseteq R \end{array} \right. \\
\equiv & \qquad \{ \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad \} \\
 & \left\{ \begin{array}{l} id \subseteq R \wedge (R \,/\, R) \cdot R \subseteq R \\ R \subseteq R \,/\, R \end{array} \right. \\
\Rightarrow & \qquad \{ \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad \} \\
 & \left\{ \begin{array}{l} R \,/\, R \subseteq R \\ R \subseteq R \,/\, R \end{array} \right. \\
\equiv & \qquad \{ \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad \} \\
 & R = R \,/\, R \\
\Box &
\end{array}
$$

That is,

$$
R = R \,/\, R \quad \equiv \quad \left\{ \begin{array}{l} id \subseteq R \\ R \cdot R \subseteq R \end{array} \right.
$$

□

SYMMETRIC DIVISION Given two arbitrary relations R and S typed as in the diagram below, define the *symmetric division* $\frac{S}{R}$ of S by R by:

$$
b \, \frac{S}{R} \, c \;\equiv\; \langle \forall\, a \;::\; a \, R \, b \Leftrightarrow a \, S \, c \rangle \qquad
\begin{tikzcd}
B \arrow[dr, "R"'] & & C \arrow[ll, "\frac{S}{R}"'] \arrow[dl, "S"] \\
 & A &
\end{tikzcd}
\qquad (5.171)
$$

That is, $b \, \frac{S}{R} \, c$ means that b and c are related to exactly the same outputs (in A) by R and by S. Another way of writing (5.171) is $b \, \frac{S}{R} \, c \equiv \{a \mid a \, R \, b\} = \{a \mid a \, S \, c\}$ which is the same as

$$
b \, \frac{S}{R} \, c \quad \equiv \quad \Lambda R \, b = \Lambda S \, c \qquad (5.172)
$$

where Λ is the *power transpose* operator[26] which maps a relation $Q : Y \leftarrow X$ to the set valued function $\Lambda Q : X \to \mathsf{P}\ Y$ such that $\Lambda Q\ x = \{y \mid y\ Q\ x\}$. Another way to define $\frac{S}{R}$ is

$$\frac{S}{R} \quad = \quad R \setminus S \cap R^{\circ} / S^{\circ} \tag{5.173}$$

which factors symmetric division into the two asymmetric divisions $R \setminus S$ (5.161) and R / S (5.159) already studied above. Moreover, for $R, S := f, g$, definition (5.173) instantiates to $\frac{f}{g}$ as defined by (5.49). By (5.161, 5.159), (5.173) is equivalent to the universal property:

$$X \subseteq \frac{S}{R} \quad \equiv \quad R \cdot X \subseteq S \wedge S \cdot X^{\circ} \subseteq R \tag{5.174}$$

From the definitions above a number of standard properties arise:

$$\left(\frac{S}{R}\right)^{\circ} \quad = \quad \frac{R}{S} \tag{5.175}$$

$$\frac{S}{R} \cdot \frac{Q}{S} \quad \subseteq \quad \frac{Q}{R} \tag{5.176}$$

$$f^{\circ} \cdot \frac{S}{R} \cdot g \quad = \quad \frac{S \cdot g}{R \cdot f} \tag{5.177}$$

$$id \quad \subseteq \quad \frac{R}{R} \tag{5.178}$$

Thus $\frac{R}{R}$ is always an *equivalence relation*, for any given R. Furthermore,

$$R = \frac{R}{R} \quad \equiv \quad R \text{ is an equivalence relation} \tag{5.179}$$

holds.Also note that, even in the case of functions, (5.176) remains an inclusion,

$$\frac{f}{g} \cdot \frac{h}{f} \subseteq \frac{h}{g} \tag{5.180}$$

since:

$$\begin{array}{cl}
& \frac{f}{g} \cdot \frac{h}{f} \subseteq \frac{h}{g} \\
\Leftarrow & \quad \{ \text{ factor } \frac{id}{g} \text{ out } \} \\
& f \cdot \frac{h}{f} \subseteq h \\
\Leftarrow & \quad \{ \text{ factor } h \text{ out } \} \\
& f \cdot \frac{id}{f} \subseteq id \\
\equiv & \quad \{ \text{ shunting rule (5.47) } \} \\
& f \subseteq f \\
\equiv & \quad \{ \text{ trivial } \} \\
& true
\end{array}$$

□

[26] See section 5.24 for more details about this operator.

From (5.180) it follows that $\frac{f}{f}$ is always transitive. By (5.175) it is symmetric and by (5.30) it is reflexive. Thus $\frac{f}{f}$ is an *equivalence relation*.

RELATION SHRINKING Given relations $R : A \leftarrow B$ and $S : A \leftarrow A$, define $R \restriction S : A \leftarrow B$, pronounced "$R$ shrunk by S", by

$$X \subseteq R \restriction S \;\equiv\; X \subseteq R \wedge X \cdot R^\circ \subseteq S \tag{5.181}$$

cf. diagram:

$$\begin{tikzcd}
& B \arrow[dl, "R\restriction S"'] \arrow[d, "R"] \\
A & A \arrow[l, "S"]
\end{tikzcd}$$

This states that $R \restriction S$ is the largest part of R such that, if it yields an output for an input x, it must be a maximum, with respect to S, among all possible outputs of x by R. By indirect equality, (5.181) is equivalent to the closed definition:

$$R \restriction S \;=\; R \cap S / R^\circ \tag{5.182}$$

(5.181) can be regarded as a Galois connection between the set of all *subrelations* of R and the set of *optimization criteria* (S) on its outputs.

Combinator $R \restriction S$ also makes sense when R and S are finite, relational data structures (e.g. tables in a database). Consider, for instance, the following example of $R \restriction S$ in a *data-processing* context: given

Examiner	*Mark*	*Student*
Smith	10	*John*
Smith	11	*Mary*
Smith	15	*Arthur*
Wood	12	*John*
Wood	11	*Mary*
Wood	15	*Arthur*

and wishing to "choose the best mark" for each student, project over *Mark*, *Student* and optimize over the $\geqslant$ ordering on *Mark*:

$$\left(\begin{array}{c|c} \mathit{Mark} & \mathit{Student} \\ \hline 10 & \mathit{John} \\ 11 & \mathit{Mary} \\ 12 & \mathit{John} \\ 15 & \mathit{Arthur} \end{array}\right) \restriction \geqslant \;=\; \left(\begin{array}{c|c} \mathit{Mark} & \mathit{Student} \\ \hline 11 & \mathit{Mary} \\ 12 & \mathit{John} \\ 15 & \mathit{Arthur} \end{array}\right)$$

Relational shrinking can be used in many other contexts. Consider, for instance, a sensor recording temperatures (T), $T \xleftarrow{S} \mathbb{N}_0$, where data in $\mathbb{N}_0$ are "time stamps". Suppose one wishes to filter out repeated temperatures, keeping the first occurrences only. This can be specified by:

$$T \xleftarrow{nub\ S} \mathbb{N}_0 \;=\; (S^\circ \restriction \leqslant)^\circ$$

That is, *nub* is the function that removes all duplicates while keeping the first instances.

Among the properties of shrinking [60] we single out the two *fusion* rules:

$$\begin{aligned}(S \cdot f) \upharpoonright R &= (S \upharpoonright R) \cdot f \qquad (5.183)\\ (f \cdot S) \upharpoonright R &= f \cdot (S \upharpoonright (f^\circ \cdot R \cdot f)) \qquad (5.184)\end{aligned}$$

Some more basic properties are: "chaotic optimization",

$$R \upharpoonright \top = R \qquad (5.185)$$

"impossible optimization"

$$R \upharpoonright \bot = \bot \qquad (5.186)$$

and "brute force" determinization:

$$R \upharpoonright id = \text{largest deterministic fragment of } R \qquad (5.187)$$

$R \upharpoonright id$ is the extreme case of the fact which follows:

$$R \upharpoonright S \text{ is simple} \Leftarrow S \text{ is anti-symmetric} \qquad (5.188)$$

Thus anti-symmetric criteria always lead to determinism, possibly at the sacrifice of totality. Also, for R simple:

$$R \upharpoonright S = R \quad \equiv \quad \mathsf{img}\ R \subseteq S \qquad (5.189)$$

Thus (for functions):

$$f \upharpoonright S = f \quad \Leftarrow \quad S \text{ is reflexive} \qquad (5.190)$$

The distribution of shrinking by join,

$$(R \cup S) \upharpoonright Q = (R \upharpoonright Q) \cap Q/S^\circ \cup (S \upharpoonright Q) \cap Q/R^\circ \qquad (5.191)$$

has a number of corollaries, namely a *conditional rule*,

$$(p \rightarrow R\ ,\ T) \upharpoonright S = p \rightarrow (R \upharpoonright S)\ ,\ (p \upharpoonright S) \qquad (5.192)$$

the *distribution* over alternatives (5.114),

$$[R\ ,S] \upharpoonright U = [R \upharpoonright U\ ,S \upharpoonright U] \qquad (5.193)$$

and the "*function competition*" rule:

$$(f \cup g) \upharpoonright S = (f \cap S \cdot g) \cup (g \cap S \cdot f) \qquad (5.194)$$

(Recall that $S/g^\circ = S \cdot g$.)

Putting universal properties (5.174,5.181) together we get, by indirect equality,

$$\begin{aligned}\frac{R}{g} &= g^\circ \cdot (R \upharpoonright id) \qquad (5.195)\\ \frac{f}{R} &= (R \upharpoonright id)^\circ \cdot f \qquad (5.196)\end{aligned}$$

capturing a relationship between shrinking and symmetric division: knowing that $R \upharpoonright id$ is the deterministic fragment of R, we see how the *vagueness* of arbitrary R replacing either f or g in $\frac{f}{g}$ is forced to shrink.

***Exercise** 5.50. Use shrinking and other relational combinators to calculate, from a relation of type (5.129), the relation of type Student × Course → Result that tells the final results of all exams. (**NB**: assume Time = $\mathbb{N}_0$ ordered by $(\leqslant)$.)*
□

RELATION OVERRIDING Another operator enabled by relation division is the relational *overriding* combinator,

$$R \dagger S \;=\; S \cup R \cap \bot / S^\circ \tag{5.197}$$

which yields the relation which contains the whole of S and that part of R where S is undefined — read $R \dagger S$ as "R overridden by S".

It is easy to show that $\bot \dagger S = S$, $R \dagger \bot = R$ and $R \dagger R = R$ hold. From (5.197) we derive, by indirect equality, the universal property:

$$X \subseteq R \dagger S \;\;\equiv\;\; X \;\subseteq\; R \cup S \;\wedge\; (X - S) \cdot S^\circ = \bot \tag{5.198}$$

The following property establishes a relationship between overriding and the McCarthy conditional:

$$p \rightarrow g\,,\, f \;=\; f \dagger (g \cdot \Phi_p) \tag{5.199}$$

Notation Φ_p is explained in the next section.

Below we show how to use relation restriction and overriding in specifying the operation that, in the Alcuin puzzle — recall (5.74)

$$\begin{tikzcd}
Being \arrow[r, "Eats"] & Being \arrow[d, "where"'] & \\
& Bank \arrow[r, "cross"] & Bank
\end{tikzcd}$$

— specifies the move of *Being*s to the other bank:

$$carry\ who\ where = where \dagger (cross \cdot where \cdot \Phi_{\in\, who})$$

By (5.199) this simplifies to a McCarthy conditional:

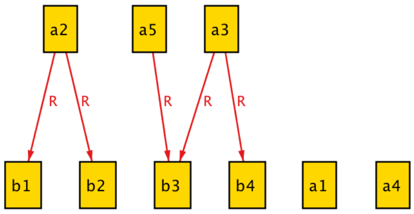
$$carry\ who\ where = (\in who) \rightarrow cross \cdot where\ ,\ where \tag{5.200}$$

In pointwise notation, *carry* is the function:

```
$carry\ who\ where\ b =$
  **if** $b \in who$ **then** $cross\ m$ **else** $m$
    **where** $m = where\ b$
```

Note the type $carry : \mathsf{P}Being \rightarrow Bank^{Being} \rightarrow Bank^{Being}$.

***Exercise** 5.51. Let $R : A \rightarrow B$ be the relation depicted below, where $A = \{a_1, a_2, a_3, a_4, a_5\}$ and $B = \{b_1, b_2, b_3, b_4\}$:*



Represent the following relation overridings in the form of Boolean $(0,1)$ *matrices:*

$$P = \top \dagger R = \begin{array}{c|c|c|c|c|c|}
 & a_1 & a_2 & a_3 & a_4 & a_5 \\
\hline
b_1 & & & & & \\
\hline
b_2 & & & & & \\
\hline
b_3 & & & & & \\
\hline
b_4 & & & & & \\
\hline
\end{array} \qquad Q = R \dagger (\underline{b_4} \cdot \underline{a_2}^{\circ}) = \begin{array}{c|c|c|c|c|c|}
 & a_1 & a_2 & a_3 & a_4 & a_5 \\
\hline
b_1 & & & & & \\
\hline
b_2 & & & & & \\
\hline
b_3 & & & & & \\
\hline
b_4 & & & & & \\
\hline
\end{array}$$

Tell which are entire, simple or surjective.

□

Exercise *5.52. Show that*

$$R \dagger f = f$$

holds, arising from (5.198,5.140) — where f is a function, of course.

□

5.20 PREDICATES ALSO BECOME RELATIONS

Recall from (5.49) the notation $\frac{f}{g} = g^{\circ} \cdot f$ and define, given a predicate $p : A \to \mathbb{B}$, the relation $\Phi_p : A \to A$ as follows:[27]

$$\Phi_p = id \cap \frac{true}{p} \tag{5.201}$$

By (5.49), Φ_p is the *coreflexive* relation which represents predicate p as a binary relation,

$$y \, \Phi_p \, x \;\Leftrightarrow\; y = x \wedge p \, x \tag{5.202}$$

as can be easily checked. From (5.201) one gets the limit situations:[28]

$$\Phi_{true} = id \tag{5.203}$$

$$\Phi_{false} = \bot \tag{5.204}$$

[27] Recall that *true* is the *constant* function yielding True for every argument (5.40).

[28] $\Phi_{false} = \bot$ arises from (5.54) since True $\neq$ False.

Moreover,

$$\Phi_{p \wedge q} = \Phi_p \cap \Phi_q \tag{5.205}$$

$$\Phi_{p \vee q} = \Phi_p \cup \Phi_q \tag{5.206}$$

$$\Phi_{\neg p} = id - \Phi_p \tag{5.207}$$

follow immediately from (5.202) and from (5.39) one infers $\frac{true}{p} \cdot R \subseteq \frac{true}{p}$ for any R. In particular, $\frac{true}{p} \cdot \top = \frac{true}{p}$ since $\frac{true}{p} \subseteq \frac{true}{p} \cdot \top$ always holds. Then, by distributive property (5.62):

$$\Phi_p \cdot \top = \frac{true}{p} \tag{5.208}$$

Moreover, the following two properties hold:

$$R \cdot \Phi_p = R \cap \top \cdot \Phi_p \tag{5.209}$$

$$\Phi_q \cdot R = R \cap \Phi_q \cdot \top \tag{5.210}$$

We check (5.210):[29]

$$\begin{array}{ll}
 & \Phi_q \cdot R \\
= & \quad \{ \text{ (5.109) ; (5.201) } \} \\
 & \frac{\langle id, true \rangle}{\langle id, q \rangle} \cdot R \\
= & \quad \{ \text{ (5.104) } \} \\
 & \langle id, q \rangle^\circ \cdot \langle R, true \rangle \\
= & \quad \{ \text{ (5.108) } \} \\
 & R \cap \frac{true}{q} \\
= & \quad \{ \text{ (5.208) } \} \\
 & R \cap \Phi_q \cdot \top \\
\Box &
\end{array}$$

Note the meaning of (5.209) and (5.210):

$$b\ (R \cdot \Phi_p)\ a \ \Leftrightarrow\ b\ R\ a \wedge (p\ a)$$
$$b\ (\Phi_q \cdot R)\ a \ \Leftrightarrow\ b\ R\ a \wedge (q\ b)$$

So (5.209) — resp. (5.210) — restricts R to inputs satisfying p — resp. outputs satisfying q.

A notable property of coreflexive relations is that their composition coincides with their meet:

$$\Phi_q \cdot \Phi_p = \Phi_q \cap \Phi_p \tag{5.211}$$

[29] The other is obtained from (5.210) by taking converses.

In consequence, composing a coreflexive with itself yields that very same coreflexive: $\Phi_p \cdot \Phi_p = \Phi_p$. (5.211) follows from (5.209,5.210):

$$\begin{array}{cl}
& \Phi_q \cdot \Phi_p \\
= & \qquad \{ \; R = R \cap R \; \} \\
& \Phi_q \cdot \Phi_p \cap \Phi_p \cdot \Phi_p \\
= & \qquad \{ \; \text{(5.209,5.210)} \; \} \\
& \Phi_q \cap \top \cdot \Phi_p \cap \Phi_p \cap \Phi_q \cdot \top \\
= & \qquad \{ \; \text{since } \Phi_p \subseteq \top \cdot \Phi_p \text{ and } \Phi_q \subseteq \Phi_q \cdot \top \; \} \\
& \Phi_q \cap \Phi_p \\
\Box &
\end{array}$$

EQUALIZERS The definition of Φ_p (5.189) can be regarded as a particular case of an *equalizer*: given two functions $B \xleftarrow{f,g} A$, the equalizer of f and g is the relation $eq\ (f,g) = id \cap \frac{f}{g}$. By indirect equality,

$$X \subseteq eq\ (f,g) \Leftrightarrow X \subseteq id \wedge g \cdot X \subseteq f$$

That is, $eq\ (f,g)$ is the largest coreflexive X that restricts g so that f and g yield the same outputs.

Clearly, $eq\ (f,f) = id$. Note that an equalizer can be empty, cf. e.g. $eq\ (true,false) = \bot$.

***Exercise** 5.53. Based on (5.71) show that*

$$g^\circ \cdot \Phi_p \cdot f \quad = \quad \frac{f}{g} \cap \frac{true}{p \cdot g} \tag{5.212}$$

holds.[30]

□

5.21 GUARDS, COREFLEXIVES AND THE MCCARTHY CONDITIONAL

From the definition of a McCarthy conditional (2.70) we obtain $p? = p \rightarrow i_1\ ,\ i_2$ and then $p? = i_2 \dagger i_1 \cdot \Phi_p$ by (5.199). A third way to express the guard $p?$ is

$$p? = i_1 \cdot \Phi_p \cup i_2 \cap (\bot \;/\; (i_1 \cdot \Phi_p)^\circ) \tag{5.213}$$

by (5.197), which simplifies to:

$$p? = [\Phi_p\,, \Phi_{\neg\, p}]^\circ \tag{5.214}$$

[30] Both sides of the equality mean $g\ b = f\ a \wedge p\ (g\ b)$.

To prove (5.214) note that $\bot \,/\, (i_1 \cdot \Phi_p)^\circ = \bot \,/\, \Phi_p$ follows immediately by the laws of $S \,/\, R$ and shunting. Then, $\bot \,/\, \Phi_p = \top \cdot \Phi_{\neg\, p}$. Here one only needs to check:

$$
\begin{array}{ll}
& \bot \,/\, \Phi_p \subseteq \top \cdot \Phi_{\neg\, p} \\
\equiv & \qquad \{\ \frac{\neg \cdot p}{true} = \frac{p}{false}\ \} \\
& \bot \,/\, \Phi_p \subseteq \dfrac{p}{false} \\
\equiv & \qquad \{\ \text{going pointwise}\ \} \\
& \langle \forall\, y, x \ :\ y\ (\bot \,/\, \Phi_p)\ x :\ p\ x = \mathsf{False} \rangle \\
\equiv & \qquad \{\ (5.161)\ ;\ (5.202)\ \} \\
& \langle \forall\, y, x \ :\ p\ x \Rightarrow \mathsf{False} :\ p\ x = \mathsf{False} \rangle \\
\equiv & \qquad \{\ \text{trivial}\ \} \\
& true \\
\Box &
\end{array}
$$

Finally, back to (5.213):

$$
\begin{array}{ll}
& p? = i_1 \cdot \Phi_p \cup i_2 \cap \top \cdot \Phi_{\neg\, p} \\
\equiv & \qquad \{\ (5.209)\ ;\ \text{converses}\ \} \\
& (p?)^\circ = \Phi_p \cdot i_1^\circ \cup \Phi_{\neg\, p} \cdot i_2^\circ \\
\equiv & \qquad \{\ (5.121)\ \} \\
& p? = [\Phi_p\,, \Phi_{\neg\, p}]^\circ \\
\Box &
\end{array}
$$

***Exercise** 5.54. From (5.214) infer*

$$
p \rightarrow R\ ,\ S \quad = \quad R \cap \frac{p}{true} \cup S \cap \frac{p}{false} \tag{5.215}
$$

and therefore $p \rightarrow R\ ,\ S \ \subseteq\ R \cup S$. Furthermore, derive (2.78) from (5.215) knowing that $true \cup false = \top$.

□

DOMAIN AND RANGE Suppose one computes $\mathsf{ker}\ \langle R, id \rangle$ instead of $\mathsf{ker}\ R$. Since $\mathsf{ker}\ \langle R, id \rangle = \mathsf{ker}\ R \cap id$ (5.111), coreflexive relation is obtained. This is called the *domain* of R, written:

$$
\delta\, R = \mathsf{ker}\ \langle R, id \rangle \tag{5.216}
$$

Since[31]

$$
\top \cdot R \cap id = R^\circ \cdot R \cap id \tag{5.217}
$$

[31] (5.217) follows from $id \cap \top \cdot R \subseteq R^\circ \cdot R$ which can be easily checked pointwise.

domain can be also defined by

$$\delta\, R = \top \cdot R \cap id \tag{5.218}$$

Dually, one can define the *range* of R as the domain of its converse:

$$\rho\, R = \delta\, R^{\circ} = \mathsf{img}\ R \cap id \tag{5.219}$$

For functions, range and image coincide, since $\mathsf{img}\, f \subseteq id$ for any f. For injective relations, domain and kernel coincide, since $\mathsf{ker}\ R \subseteq id$ in such situations. These two operators can be shown to be characterized by two Galois connections, as follows:

<table>
<tr><td colspan="4">$(f\ X) \subseteq Y \equiv X \subseteq (g\ Y)$</td></tr>
<tr><th>Description</th><th>f</th><th>g</th><th>Obs.</th></tr>
<tr><td>domain</td><td>δ</td><td>$(\top\cdot)$</td><td>left $\subseteq$ restricted to coreflexives</td></tr>
<tr><td>range</td><td>ρ</td><td>$(\cdot\top)$</td><td>left $\subseteq$ restricted to coreflexives</td></tr>
</table>

Let us show that indeed

$$\delta\, X \subseteq Y \equiv X \subseteq \top \cdot Y \tag{5.220}$$

$$\rho\, R \subseteq Y \equiv R \subseteq Y \cdot \top \tag{5.221}$$

hold, where variable Y ranges over coreflexive relations only. We only derive (5.220), from which (5.221) is obtained taking converses. We rely on the definition just given and on previously defined connections:

$$\begin{array}{cl}
 & \delta\, X \subseteq Y \\
\equiv & \quad \{\ (5.218)\ \} \\
 & \top \cdot X \cap id \subseteq Y \\
\equiv & \quad \{\ \text{two Galois connections}\ \} \\
 & X \subseteq \top \setminus (id \Rightarrow Y) \\
\equiv & \quad \{\ \top \setminus (id \Rightarrow Y) = \top \cdot Y, \text{see below}\ \} \\
 & X \subseteq \top \cdot Y \\
\Box &
\end{array}$$

To justify the hint above, first note that $\top \cdot Y \subseteq id \Rightarrow Y$, for Y coreflexive — recall (5.201) and (5.208). Then:

$$\begin{array}{cl}
 & \top \setminus (id \Rightarrow Y) \subseteq \top \cdot Y \\
\Leftarrow & \quad \{\ \text{monotonicity ; rule “raise-the-lower-side”}\ \} \\
 & \top \setminus (\top \cdot Y) \subseteq \top \cdot Y \\
\equiv & \quad \{\ (5.169)\ ; f \cdot f^{\circ} \cdot f = f \text{ for } f := !\ \text{(twice)}\ \} \\
 & ! \setminus ! \cdot Y \subseteq \top \cdot Y \\
\equiv & \quad \{\ f \setminus R = f^{\circ} \cdot R\ ; \top = \mathsf{ker}\ !\ \} \\
 & \top \cdot Y \subseteq \top \cdot Y \\
\Box &
\end{array}$$

Note the left-cancellation rule of the δ connection:

$$R \subseteq \top \cdot \delta\, R \qquad (5.222)$$

From this the following domain/range elimination rules follow:

$$\begin{array}{rcll} \top \cdot \delta\, R &=& \top \cdot R & (5.223)\\ \rho\, R \cdot \top &=& R \cdot \top & (5.224)\\ \delta\, R \subseteq \delta\, S &\equiv& R \subseteq \top \cdot S & (5.225) \end{array}$$

Proof of (5.223):

$$\begin{array}{cl} & \top \cdot \delta\, R = \top \cdot R \\ \equiv & \qquad \{\ \text{circular inclusion}\ \} \\ & \top \cdot \delta\, R \subseteq \top \cdot R \wedge \top \cdot R \subseteq \top \cdot \delta\, R \\ \equiv & \qquad \{\ (5.97)\ \text{twice}\ \} \\ & \delta\, R \subseteq \top \cdot R \wedge R \subseteq \top \cdot \delta\, R \\ \equiv & \qquad \{\ \text{cancelation (5.222)}\ \} \\ & \delta\, R \subseteq \top \cdot R \\ \equiv & \qquad \{\ \delta\, R = \top \cdot R \cap id\ (5.218)\ \} \\ & true \\ \Box & \end{array}$$

Rule (5.224) follows by dualization (converses) and (5.225) follows from (5.220) and (5.223). More facts about domain and range:

$$\begin{array}{rcll} \delta\,(R \cdot S) &=& \delta\,(\delta\, R \cdot S) & (5.226)\\ \rho\,(R \cdot S) &=& \rho\,(R \cdot \rho\, S) & (5.227)\\ R &=& R \cdot \delta\, R & (5.228)\\ R &=& \rho\, R \cdot R & (5.229) \end{array}$$

Last but not least: given predicate q and function f,

$$\Phi_{q \cdot f} = \delta\,(\Phi_q \cdot f) \qquad (5.230)$$

holds. Proof:

$$\begin{array}{cl} & \Phi_{q \cdot f} \\ = & \qquad \{\ (5.201)\ \} \\ & id \cap \dfrac{true}{q \cdot f} \\ = & \qquad \{\ \text{since } \frac{f}{f} \text{ is reflexive (5.30)}\ \} \\ & id \cap \dfrac{f}{f} \cap \dfrac{true \cdot f}{q \cdot f} \\ = & \qquad \{\ (5.109)\,;\ \text{products}\ \} \end{array}$$

$$
\begin{array}{ll}
& id \cap \dfrac{\langle id, true\rangle \cdot f}{\langle id, q\rangle \cdot f} \\
= & \quad \{ \ (5.49) \ ; (5.109) \ \} \\
& id \cap f^\circ \cdot (id \cap \dfrac{true}{q}) \cdot f \\
= & \quad \{ \ (5.201) \ \} \\
& id \cap f^\circ \cdot \Phi_q \cdot f \\
= & \quad \{ \ \delta\, R = id \cap R^\circ \cdot R \ \} \\
& \delta\, (\Phi_q \cdot f)
\end{array}
$$

□

***Exercise** 5.55. Recalling (5.209), (5.210) and other properties of relation algebra, show that: (a) (5.220) and (5.221) can be re-written with R replacing $\top$; (b) $\Phi \subseteq \Psi \equiv\ ! \cdot \Phi \subseteq\ ! \cdot \Psi$ holds.*[32]

□

5.22 DIFUNCTIONALS

A relation R is said to be *difunctional* or *regular* wherever $R \cdot R^\circ \cdot R = R$ holds, which amounts to $R \cdot R^\circ \cdot R \ \subseteq \ R$ since the converse inclusion always holds.

The class of difunctional relations is vast. $\top$ and $\bot$ are difunctional, and so are all coreflexive relations, as is easy to check. It also includes all simple relations, since $R \cdot R^\circ = \mathsf{img}\ R \ \subseteq \ id$ wherever R is simple. Moreover, divisions of functions are difunctional because every symmetric division is so, as is easy to check by application of laws (5.176) and (5.175):

$$
\begin{array}{ll}
& \dfrac{f}{g} \cdot \left(\dfrac{f}{g}\right)^\circ \cdot \dfrac{f}{g} \subseteq \dfrac{f}{g} \\
\Leftarrow & \quad \{ \ (5.51) \ ; (5.180) \ \} \\
& \dfrac{f}{g} \cdot \dfrac{f}{f} \subseteq \dfrac{f}{g} \\
\Leftarrow & \quad \{ \ (5.180) \ \} \\
& \dfrac{f}{g} \subseteq \dfrac{f}{g}
\end{array}
$$

□

For $g = id$ above we get that any function f being difunctional can be expressed by $f \cdot \frac{f}{f} = f$.

[32] Thus coreflexives can be represented by *vectors* and vice-versa.

Recall that an equivalence relation can always be represented by the kernel of some function, typically by $R = \frac{\Lambda R}{\Lambda R}$. So equivalence relations are difunctional. The following rule is of pratical relevance:

$$(R \text{ transitive } \Leftrightarrow R \text{ difunctional}) \;\Leftarrow\; \left\{ \begin{array}{l} R \text{ symmetric} \\ R \text{ reflexive} \end{array} \right. \tag{5.231}$$

Proof ($\Rightarrow$):

$$\begin{array}{ll}
 & R \text{ difunctional} \\
\equiv & \quad \{ \text{ definition of difunctional } \} \\
 & R \cdot R^\circ \cdot R \subseteq R \\
\equiv & \quad \{ \ R \text{ assumed symmetric } \} \\
 & R \cdot R \cdot R \subseteq R \\
\Leftarrow & \quad \{ \ R \text{ assumed transitive } \} \\
 & R \cdot R \cdot R \subseteq R \cdot R \\
\Leftarrow & \quad \{ \text{ monotonicity of } (R \cdot) \ \} \\
 & R \cdot R \subseteq R \\
\equiv & \quad \{ \ R \text{ assumed transitive } \} \\
 & \mathit{true} \\
\Box &
\end{array}$$

Proof ($\Leftarrow$):

$$\begin{array}{ll}
 & R \text{ transitive} \\
\equiv & \quad \{ \text{ definition } \} \\
 & R \cdot R \subseteq R \\
\equiv & \quad \{ \ R \text{ assumed difunctional } \} \\
 & R \cdot R \subseteq R \cdot R^\circ \cdot R \\
\Leftarrow & \quad \{ \text{ monotonicity of } R \cdot \_ \cdot R \ \} \\
 & id \subseteq R^\circ \\
\equiv & \quad \{ \ R \text{ assumed reflexive } \} \\
 & \mathit{true} \\
\Box &
\end{array}$$

If above we take the proof ($\Rightarrow$) alone we get that a symmetric and transitive relation is difunctional. Thus:

> *Every partial equivalence relation (Per, recall figure 5.1) is difunctional.*

If we take the other part of the proof ($\Leftarrow$) alone we get:

> *Every reflexive and difunctional relation is transitive.*

Moreover:

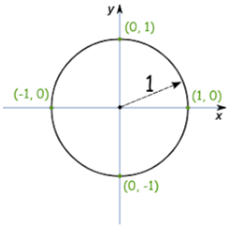
> *A difunctional relation that is* reflexive *and* symmetric *necessarily is an* equivalence *relation.*

Difunctional relations are also called *regular*, *rational* or *uniform*. First, some intuition about what "regularity" means: a regular (difunctional) relation is such that, wherever two inputs have a common image, then they have *exactly the same* set of images. In other words, the image sets of two different inputs are either disjoint or the same. As a counterexample, take the following relation, represented as a matrix with inputs taken from set $\{a_1, .., a_5\}$ and outputs delivered into set $\{b_1, .., b_5\}$:

$$\begin{array}{c|ccccc} R & a_1 & a_2 & a_3 & a_4 & a_5 \\ \hline b_1 & 0 & 0 & 1 & 0 & 1 \\ b_2 & 0 & 0 & 0 & 0 & 0 \\ b_3 & 0 & 1 & 0 & 0 & 0 \\ b_4 & 0 & 1 & 0 & 1 & 0 \\ b_5 & 0 & 0 & 0 & 1 & 0 \end{array} \tag{5.232}$$

Concerning inputs a_3 and a_5, regularity holds; but sets $\{b_3, b_4\}$ and $\{b_4, b_5\}$ — the images of a_2 and a_4, respectively — are neither disjoint nor the same: so R isn't regular. It would become so if e.g. b_4 were dropped from both image sets or one of b_3 or b_5 were replaced for the other in the corresponding image set.

Exercise *5.56. The unit circle*



can be described as the relation $\mathbb{R} \xleftarrow{R} \mathbb{R}$ *such that* $y\ R\ x \Leftrightarrow y^2 + x^2 = 1$, *that is*

$$R = sq^\circ \cdot (1-) \cdot sq \tag{5.233}$$

in pointfree notation, where $\left\{ \begin{array}{l} sq : \mathbb{R} \to \mathbb{R} \\ sq\ x = x^2 \end{array} \right.$ *and* $\left\{ \begin{array}{l} (1-) : \mathbb{R} \to \mathbb{R} \\ (1-)\ x = 1 - x \end{array} \right.$
It can be easily check that R is neither entire, nor simple, nor injective, nor surjective. Show that it is difuncional.

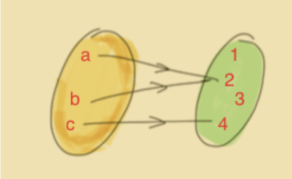
□

5.23 OTHER ORDERINGS ON RELATIONS

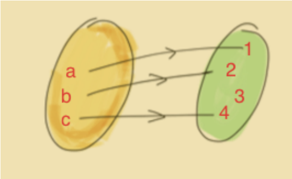
THE INJECTIVITY PREORDER The kernel relation $\ker R = R^\circ \cdot R$ *measures* the level of *injectivity* of R according to the preorder

$$R \leqslant S \quad \equiv \quad \ker S \subseteq \ker R \tag{5.234}$$

telling that R is *less injective* or *more defined* (entire) than S. For instance:



$$\leqslant$$



This ordering is surprisingly useful in formal specification because of its properties. For instance, it is upper-bounded by relation *pairing*, recall (5.103):

$$\langle R,S\rangle \leqslant X \quad \equiv \quad R \leqslant X \wedge S \leqslant X \tag{5.235}$$

Cancellation of (5.235) means that *pairing* always *increases injectivity*:

$$R \leqslant \langle R,S\rangle \quad \text{and} \quad S \leqslant \langle R,S\rangle . \tag{5.236}$$

(5.236) unfolds to $\mathsf{ker}\ \langle R,S\rangle \subseteq (\mathsf{ker}\ R) \cap (\mathsf{ker}\ S)$, confirming (5.111). The following injectivity *shunting law* arises as a Galois connection:

$$R \cdot g \leqslant S \quad \equiv \quad R \leqslant S \cdot g^{\circ} \tag{5.237}$$

Restricted to *functions*, $(\leqslant)$ is *universally* bounded by

$$! \leqslant f \leqslant id$$

where (recall) $1 \xleftarrow{\;!\;} A$ is the unique function of its type, where 1 is the singleton type. Moreover,

- A function is *injective* iff $id \leqslant f$. Thus $\langle f, id\rangle$ is always *injective* (5.236).

- Two functions f e g are said to be *complementary* wherever $id \leqslant \langle f,g\rangle$.

- Any relation R can be factored into the composition $f \cdot g^{\circ}$ of two complementary functions f and g.[33]

For instance, the *projections* $\pi_1\ (a,b = a)$, $\pi_2\ (a,b = b)$ are complementary since $\langle \pi_1, \pi_2\rangle = id$ (2.32).

As illustration of the use of this ordering in formal specification, suppose one writes

$$room \leqslant \langle lect, slot\rangle$$

in the context of the data model

$$\begin{tikzcd}
Teacher & Class \arrow[l, "lect"'] \arrow[r, "room"] \arrow[d, "slot"'] & Room \\
 & TD &
\end{tikzcd}$$

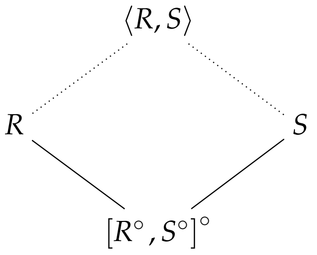
[33] This remarkable factorization is known as a *tabulation* of R [12].

where TD abbreviates time and date. What are we telling about this model by writing $room \leqslant \langle lect, slot \rangle$? We unfold this constraint in the expected way:

$$\begin{array}{ll}
 & room \leqslant \langle lect, slot \rangle \\
\equiv & \qquad \{ \text{ (5.234) } \} \\
 & \mathsf{ker}\ \langle lect, slot \rangle \subseteq \mathsf{ker}\ room \\
\equiv & \qquad \{ \text{ (5.111) ; (5.53) } \} \\
 & \frac{lect}{lect} \cap \frac{slot}{slot} \subseteq \frac{room}{room} \\
\equiv & \qquad \{ \text{ going pointwise, for all } c_1, c_2 \in Class \} \\
 & (lect\ c_1 = lect\ c_2 \wedge slot\ c_1 = slot\ c_2) \Rightarrow (room\ c_1 = room\ c_2)
\end{array}$$

This $room \leqslant \langle lect, slot \rangle$ constrains the model in the sense of imposing that a given lecturer cannot be in two different rooms at the same time. c_1 and c_2 are classes shared by different courses, possibly of different degrees. In the standard terminology of database theory this is called a *functional dependency*, see exercises 5.58 and 5.59 in the sequel.

Interestingly, the injectivity preorder not only has least upper bounds but also greatest lower bounds,



"least upper bound"

"greatest lower bound"

that is,

$$X \leqslant [R^\circ, S^\circ]^\circ \quad \Leftrightarrow \quad X \leqslant R \wedge X \leqslant S \tag{5.238}$$

as the calculation shows:

$$\begin{array}{ll}
 & X \leqslant [R^\circ, S^\circ]^\circ \\
\equiv & \qquad \{ \text{ injectivity preorder ; } \mathsf{ker}\ R^\circ = \mathsf{img}\ R \} \\
 & \mathsf{img}\ [R^\circ, S^\circ] \subseteq \mathsf{ker}\ X \\
\equiv & \qquad \{ \text{ (5.124) } \} \\
 & R^\circ \cdot R \cup S^\circ \cdot S \subseteq \mathsf{ker}\ X \\
\equiv & \qquad \{ \text{ kernel; } \cdot \cup \cdot\text{-universal } \} \\
 & \mathsf{ker}\ R \subseteq \mathsf{ker}\ X \wedge \mathsf{ker}\ S \subseteq \mathsf{ker}\ X \\
\equiv & \qquad \{ \text{ injectivity preorder (twice) } \} \\
 & X \leqslant R \wedge X \leqslant S
\end{array}$$

□

Note the meaning of the glb of R and S,

$$x\,[R^\circ, S^\circ]^\circ\, a \;\Leftrightarrow\; \langle \exists\, b \;:\; x = i_1\, b : \; b\, R\, a \rangle \vee \langle \exists\, c \;:\; x = i_2\, c : \; c\, R\, a \rangle$$

since $[R^\circ, S^\circ]^\circ = i_1 \cdot R \cup i_2 \cdot S$. This is the most injective relation that is less injective than R and S because it just "collates" the outputs of both relations without confusing them.[34]

Exercise *5.57. The Peano algebra* $\mathbb{N}_0 \xleftarrow{\mathsf{in}} 1 + \mathbb{N}_0 \;=\; [\underline{0}, \mathsf{succ}]$ *is an isomorphism*[35]*, and therefore injective. Check what (5.126) means in this case.*
□

Exercise *5.58. An SQL-like relational operator is* projection*,*

$$\pi_{g,f} R \overset{\mathrm{def}}{=} g \cdot R \cdot f^\circ \qquad \begin{tikzcd} B \arrow[d, "g"'] & A \arrow[l, "R"'] \arrow[d, "f"] \\ C & D \arrow[l, "\pi_{g,f} R"] \end{tikzcd} \tag{5.239}$$

whose set-theoretic meaning is[36]

$$\pi_{g,f} R \;=\; \{(g\, b, f\, a) \mid b \in B \wedge a \in A \wedge b\, R\, a\} \tag{5.240}$$

Functions f and g are often referred to as attributes *of R. Derive (5.240) from (5.239).*
□

Exercise *5.59. A relation R is said to satisfy* functional dependency *(FD) $g \to f$, written $g \overset{R}{\twoheadrightarrow} f$ wherever projection $\pi_{f,g} R$ (5.239) is* simple.

1. *Recalling (5.234), prove the equivalence:*

$$g \overset{R}{\twoheadrightarrow} f \;\equiv\; f \leqslant g \cdot R^\circ \tag{5.241}$$

2. *Show that $g \overset{R}{\twoheadrightarrow} f$ trivially holds wherever g is injective and R is simple, for all (suitably typed) f.*

[34] It turns out that universal property $X = [R^\circ, S^\circ]^\circ \;\Leftrightarrow\; i_1^\circ \cdot X = R \wedge i_2^\circ \cdot X = S$ holds, as is easy to derive from (5.114). So $[R^\circ, S^\circ]^\circ$ is the *categorial* product for relations:

$$\begin{tikzcd} A \to (B + C) \arrow[rr, bend left] & \cong & (A \to B) \times (A \to C) \arrow[ll, bend left] \end{tikzcd}$$

That is, among relations, the product is obtained as the converse dual of the coproduct. This is called a *biproduct* [48].

[35] Recall section 3.1.

[36] Note that any relation $R : B \leftarrow A$ defines the set of pairs $\{(b, a) \mid b\; R\; a\}$. Predicate $b\; R\; a$ describes R *intensionally*. The set $\{(b, a) \mid b\; R\; a\}$ is the *extension* of R.

3. *Prove the* composition rule *of FDs:*

$$h \overset{S\cdot R}{\twoheadleftarrow} g \quad \Leftarrow \quad h \overset{S}{\twoheadleftarrow} f \quad \wedge \quad f \overset{R}{\twoheadleftarrow} g \tag{5.242}$$

□

Exercise *5.60. Let R and S be the two relations depicted as follows:*

$$\begin{tikzcd}
C & W \arrow[l, "R"'] \arrow[r, "S"] & \mathbb{N}_0 \\
 & \texttt{"Armstrong"} \arrow[r, mapsto] \arrow[dl, mapsto] & 9 \\
\texttt{'A'} & \texttt{"Albert"} \arrow[l, mapsto] \arrow[r, mapsto] & 6 \\
\texttt{'M'} & \texttt{"Minho"} \arrow[l, mapsto] \arrow[r, mapsto] & 5 \\
\texttt{'B'} & \texttt{"Braga"} \arrow[l, mapsto] \arrow[ur, mapsto] &
\end{tikzcd}$$

Check the assertions:

1. $R \leqslant S$
2. $S \leqslant R$
3. *Both hold*
4. *None holds.*

□

Exercise *5.61. As follow up to exercise 5.9,*

- *specify the relation R between students and teachers such that t R s means: t* is the mentor of *s* and also teaches one of her/his courses.
- *Specify the property:* mentors of students necessarily are among their teachers.

□

THE DEFINITION PREORDER The injectivity preorder works perfectly for functions, which are entire relations. For non-entire R it behaves in a mixed way, measuring not only injectivity but also definition (entireness). It is useful to order relations with respect to how defined they are:

$$R \preceq S \quad \equiv \quad \delta\, R \subseteq \delta\, S \tag{5.243}$$

From $\top = \mathsf{ker}\ !$ one draws another version of (5.243), $R \preceq S \ \equiv\ ! \cdot R \subseteq\ ! \cdot S$. The following Galois connections

$$R \cup S \preceq T \quad \equiv \quad R \preceq T \wedge S \preceq T \tag{5.244}$$

$$R \cdot f^{\circ} \preceq S \quad \equiv \quad R \preceq S \cdot f \tag{5.245}$$

are easy to prove. Recalling (5.225), (5.243) can also be written

$$\delta\, R \ \subseteq\ \delta\, S \equiv R \subseteq \top \cdot S \tag{5.246}$$

THE REFINEMENT ORDER Standard programming theory relies on a notion of program *refinement*. As a rule, the starting point in any software design is a so-called *specification*, which indicates the expected behaviour of the program to be developed with no indication of *how* outputs are computed from the inputs. So, "vagueness" is a chief ingredient of a good specification, giving freedom to the programmer to choose a particular algorithmic solution.

Relation algebra captures this by ordering relations with respect to the degree in which they are closer to implementations:

$$S \vdash R \;\equiv\; \delta\, S \subseteq \delta\, R \wedge R \cdot \delta\, S \subseteq S \tag{5.247}$$

cf.

$$\begin{tikzcd}
A \arrow[dr, "R"'] & & A \arrow[ll, "\delta\, S"'] \arrow[dl, "S"] \\
& B &
\end{tikzcd}$$

$S \vdash R$ is read: "S is refined by R". In the limit situation, R is a function f, and then

$$S \vdash f \quad\Leftrightarrow\quad \delta\, S \subseteq f^{\circ} \cdot S \tag{5.248}$$

by shunting (5.46). This is a limit in the sense that f can be neither more defined nor more deterministic.

As maxima of the refinement ordering, functions are regarded as implementations *"par excellence"*. Note how (5.248) captures *implicit specification S* being refined by some function f — recall section 5.3. Back to points and thanks to (5.17) we obtain, in classical "VDM-speak":

$$\forall a.\ \text{pre-}S(a) \Rightarrow \text{post-}S(f\ a, a)$$

In case S is entire, (5.248) simplifies to $S \vdash f \;\Leftrightarrow\; f \subseteq S$. As example of this particular case we go back to section 5.3 and prove that *abs*, explicitly defined by $abs\ i = \textbf{if}\ i < 0\ \textbf{then}\ -i\ \textbf{else}\ i$, meets the implicit specification given there, here encoded by $S = \frac{true}{geq0} \cap (id \cup sym)$ where $geq0\ x = x \geqslant 0$ and $sym\ x = -x$. The explicit version below uses a McCarthy conditional, for $lt0\ x = x < 0$. By exercise 5.54 term $id \cup sym$ in S can be ignored:

$$\begin{array}{cl}
& lt0 \rightarrow sym\ ,\ id \subseteq \dfrac{true}{geq0} \\
\equiv & \quad\{\ \text{shunting (5.46)}\ \} \\
& geq0 \cdot (lt0 \rightarrow sym\ ,\ id) \subseteq true \\
\equiv & \quad\{\ \text{fusion (2.71)}\ \} \\
& lt0 \rightarrow geq0 \cdot sym\ ,\ geq0 \subseteq true \\
\equiv & \quad\{\ -x \geqslant 0 \Leftrightarrow x \leqslant 0 = leq0\ x\ \} \\
& lt0 \rightarrow leq0\ ,\ geq0 \subseteq true
\end{array}$$

$$\equiv \qquad \{\ x < 0 \Rightarrow x \leqslant 0 \text{ and } \neg\,(x < 0)\ \Leftrightarrow\ x \geqslant 0\ \}$$

$$lt0 \rightarrow true\ ,\ true\ \subseteq\ true$$

$$\equiv \qquad \{\ p \rightarrow f\ ,\ f = f \text{ (exercise 5.54) }\}$$

$$true$$

□

Finally note that an equivalent way of stating (5.247) without using the domain operator is:

$$S \vdash R \quad \equiv \quad \top \cdot S \cap \top \cdot R \cap (R \cup S) = R \tag{5.249}$$

Exercise *5.62. Prove (5.249.*

□

5.24 BACK TO FUNCTIONS

In this chapter we have argued that one needs *relations* in order to reason about *functions*. The inverse perspective — that relations can be represented as functions — also makes sense and it is, in many places, the approach that is followed.

Indeed, relations can be *transposed* back to functions without losing information. There are two transposes of interest. One is complete in the sense that it allows us to see *any* relation as a function. The other is specific, in the sense that it only applies to (the very important class of) simple relations (vulg. *partial* functions).

POWER TRANSPOSE Let $A \xrightarrow{R} B$ be a relation and define the function

$$\begin{aligned} &\Lambda R : A \rightarrow \mathsf{P}\ B \\ &\Lambda R\ a = \{b \mid b\ R\ a\} \end{aligned}$$

which is such that:

$$\Lambda R = f \quad \equiv \quad \langle \forall\ b, a\ ::\ b\ R\ a\ \Leftrightarrow\ b \in f\ a \rangle \tag{5.250}$$

That is:

$$f = \Lambda R \ \Leftrightarrow\ \in \cdot f = R \tag{5.251}$$

cf.

$$\begin{tikzcd} A \rightarrow \mathsf{P}\ B \arrow[r, bend left, "(\in\cdot)"] \arrow[r, phantom, "\cong"] & A \rightarrow B \arrow[l, bend left, "\Lambda"] \end{tikzcd}$$

In words: any *relation* can be faithfully represented by a set-valued *function*.

For instance, moving the variables of (5.172) outwards by use of (5.17), we obtain the following *power transpose cancellation* rule:[37]

$$\frac{\Lambda S}{\Lambda R} = \frac{S}{R} \tag{5.252}$$

Read from right to left, this shows a way of converting arbitrary symmetric divisions into function divisions.

“MAYBE” TRANSPOSE Let $A \xrightarrow{S} B$ be a *simple* relation. Define the function

$$\Gamma S : A \to B + 1$$

such that:

$$\Gamma S = f \quad \Leftrightarrow \quad \langle \forall\, b, a \; :: \; b\, S\, a \; \Leftrightarrow \; (i_1\, b) = f\, a \rangle$$

That is:

$$f = \Gamma S \;\; \Leftrightarrow \;\; S = i_1^\circ \cdot f \tag{5.253}$$

cf.

$$\begin{tikzcd}(B+1)^A \arrow[r, bend left, "(i_1^\circ\cdot)"] \arrow[r, phantom, "\cong"] & A \to B \arrow[l, bend left, "\Gamma"]\end{tikzcd}$$

In words: simple *relations* can always be represented by “Maybe”, or “pointer”-valued *functions*. Recall section 4.1, where the *Maybe* monad was used to “totalize” partial functions. Isomorphism (5.253) explains why such a totalization maske sense. For finite relations, and assuming these represented extensionally as lists of pairs, the function $lookup :: Eq\ a \Rightarrow a \to [(a,b)] \to Maybe\ b$ in Haskell implements the “Maybe”-transpose.

5.25 BIBLIOGRAPHY NOTES

Chronologically, relational notation emerged — earlier than predicate logic itself — in the work of Augustus De Morgan (1806-71) on binary relations [50]. Later, Peirce (1839-1914) invented quantifier notation to explain De Morgan’s algebra of relations (see e.g. [50] for details). De Morgan’s pioneering work was ill fated: the language[38] invented to explain his calculus of relations became eventually more popular than the calculus itself. Alfred Tarski (1901-83), who had a life-long struggle with quantified notation [21, 29], revived relation algebra. Together

[37] This rule is nothing but another way of stating exercise 4.48 proposed in [12]. Note that ΛR is always a function.

[38] Meanwhile named FOL, first order logic.

with Steve Givant he wrote a book (published posthumously) on *set theory without variables* [82].

Meanwhile, category theory was born, stressing the role of *arrows* and *diagrams* and on the arrow language of diagrams, which is inherently *pointfree*. The category of sets and functions immediately provided a basis for pointfree functional reasoning, but this was by and large ignored by John Backus (1924-2007) in his FP algebra of programs [8] which is APL-flavoured. (But there is far more in it than such a flavour, of course!) Anyway, Backus' landmark FP paper was among the first to show how relevant such reasoning style is to computing.

A bridge between the two pointfree schools, the relational and the categorial, was eventually established by Freyd and Ščedrov [24] in their proposal of the concept of an *allegory*. This gave birth to *typed* relation algebra and relation (semi-commutative) diagrams like those adopted in the current book for *relational thinking*. The pointfree algebra of programming (AoP) as it is understood today, stems directly from [24]. Its has reached higher education thanks to textbook [12] written by Bird and Moor.

In his book on *relational mathematics* [80], Gunther Schmidt makes extensive use of matrix displays, notation, concepts and operations in relation algebra. Winter [86] generalizes relation algebra to so-called Goguen categories.

In the early 1990s, the Groningen-Eindhoven MPC group led by Backhouse [1, 5] contributed decisively to the AoP by structuring relation algebra in terms of Galois connections. This elegant approach has been very influential in the way (typed) relation algebra was perceived afterwards, for instance in the way relation shrinking was introduced in the algebra [60, 73]. Galois connections are also the "Swiss knife" of [60].

Most of the current chapter was inspired by [5].

6

THEOREMS FOR FREE BY CALCULATION

6.1 INTRODUCTION

As already stressed in previous chapters, type polymorphism remains one of the most useful and interesting ingredients of functional programming. For example, the two functions

$$\begin{array}{l}
\mathit{countBits} : \mathbb{B}^* \to \mathbb{N}_0 \\
\mathit{countBits}\ [\,] = 0 \\
\mathit{countBits}\ (b : bs) = 1 + \mathit{countBits}\ bs
\end{array}$$

and

$$\begin{array}{l}
\mathit{countNats} : {\mathbb{N}_0}^* \to \mathbb{N}_0 \\
\mathit{countNats}\ [\,] = 0 \\
\mathit{countNats}\ (b : bs) = 1 + \mathit{countNats}\ bs
\end{array}$$

are both subsumed by a single, *generic* (that is, parametric) program:

$$\begin{array}{l}
\mathit{count} : (\forall\ A)\ A^* \to \mathbb{N}_0 \\
\mathit{count}\ [\,] = 0 \\
\mathit{count}\ (a : as) = 1 + \mathit{count}\ as
\end{array}$$

Written as a catamorphism

$$(\!|\, \mathsf{in}_{\mathbb{N}_0} \cdot (id + \pi_2)\,|\!)$$

and thus even dispensing with a name, it becomes clear why this function is generic: nothing in

$$\mathsf{in}_{\mathbb{N}_0} \cdot (id + \pi_2)$$

is susceptible to the *type* of the elements that are being counted up!

This form of polymorphism, known as *parametric polymorphism*, is attractive because

- one writes less code (*specific* solution = *generic* solution + *customization*);
- it is intellectually rewarding, as it brings elegance and economy in programming;

- and, last but not least[1],

> "(...) *from the* type *of a polymorphic function we can derive a* theorem *that it satisfies.* (...) *How useful are the theorems so generated? Only time and experience will tell* (...)"

Recall that section 2.12 already addresses these theorems, also called *natural properties*. However, the full spread of naturality is not explored there. In particular, it does not address higher-order (exponential) types.

It turns out that the "free theorems" involving such types are easy to derive in relation algebra. The current chapter is devoted to such a generic derivation and includes a number of examples showing how vast the application of *free theorems* is.

6.2 POLYMORPHIC TYPE SIGNATURES

In any typed functional language, when declaring a polymorphic function one is bound to use the same generic format,

$$f \; : \; t$$

known as the function's *signature*: f is the name of the function and t is a functional type written according to the following "grammar" of types:

$$\begin{array}{lll} t & ::= & t' \rightarrow t'' \\ t & ::= & \mathsf{F}(t_1, \ldots, t_n) \qquad \text{F is a } \textit{type} \text{ constructor} \\ t & ::= & v \qquad \text{a type } \textit{variable}\text{, source of polymorphism.} \end{array}$$

What does it mean for $f : t$ to be *parametrically* polymorphic? We shall see shortly that what matters in this respect is the formal structure of type t. Let

- V be the set of type variables involved in type expression t;
- $\{R_v\}_{v \in V}$ be a V-indexed family of relations (f_v in case R_v is a function);
- R_t be a relation defined inductively as follows:

$$R_{t:=v} = R_v \tag{6.1}$$

$$R_{t:=\mathsf{F}(t_1,\ldots,t_n)} = \mathsf{F}(R_{t_1}, \ldots, R_{t_n}) \tag{6.2}$$

$$R_{t:=t' \rightarrow t''} = R_{t'} \rightarrow R_{t''} \tag{6.3}$$

Two questions arise: what does F in the right handside of (6.2) mean? What kind of relation is $R_{t'} \rightarrow R_{t''}$ in (6.3)?

First of all, and to answer the first question, we need the concept of *relator*, which extends that of a *functor* (introduced in section 3.8) to relations.

[1] Quoting *Theorems for free!*, by Philip Wadler [85].

6.3 RELATORS

A functor G is said to be a *relator* wherever, given a relation R from A to B, $\mathsf{G}\,R$ extends R to G-structures: it is a relation from $\mathsf{G}\,A$ to $\mathsf{G}\,B$

$$\begin{tikzcd} A \arrow[r, dotted, no head] \arrow[d, "R"'] & \mathsf{G}\,A \arrow[d, "\mathsf{G}\,R"] \\ B \arrow[r, dotted, no head] & \mathsf{G}\,B \end{tikzcd} \tag{6.4}$$

which obeys the properties of a functor,

$$\mathsf{G}\,id = id \tag{6.5}$$

$$\mathsf{G}\,(R \cdot S) = (\mathsf{G}\,R) \cdot (\mathsf{G}\,S) \tag{6.6}$$

— recall (3.54) and (3.55) — plus the properties:

$$R \subseteq S \;\Rightarrow\; \mathsf{G}\,R \subseteq \mathsf{G}\,S \tag{6.7}$$

$$\mathsf{G}\,(R^\circ) = (\mathsf{G}\,R)^\circ \tag{6.8}$$

That is, a relator is a functor that is monotonic and commutes with converse. For instance, the "Maybe" functor $\mathsf{G}\;X = 1 + X$ is an example of relator:

$$\begin{tikzcd} A \arrow[r, dotted, no head] \arrow[d, "R"'] & \mathsf{G}\,A = 1 + A \arrow[d, "\mathsf{G}\,R = id + R"] \\ B \arrow[r, dotted, no head] & \mathsf{G}\,B = 1 + B \end{tikzcd}$$

It is monotonic since $\mathsf{G}\;R = id + R$ only involves monotonic operators and commutes with converse via (5.123). Let us unfold $\mathsf{G}\,R = id + R$:

$$\begin{array}{ll} & y(id + R)x \\ \equiv & \quad \{ \text{ unfolding the sum, cf. } id + R = [i_1 \cdot id\,, i_2 \cdot R] \text{ (5.119) } \} \\ & y(i_1 \cdot i_1^\circ \cup i_2 \cdot R \cdot i_2^\circ)x \\ \equiv & \quad \{ \text{ relational union (5.57); image } \} \\ & y(\mathsf{img}\ i_1)x \vee y(i_2 \cdot R \cdot i_2^\circ)x \\ \equiv & \quad \{ \text{ let } NIL \text{ denote } \underline{\text{the}} \text{ sole inhabitant of the singleton type } \} \\ & y = x = i_1 NIL \vee \langle \exists\, b, a \;:\; y = i_2\, b \wedge x = i_2\, a :\; b\ R\ a \rangle \end{array}$$

In words: two "pointer-values" x and y are $\mathsf{G}\;R$-related iff they are both null or they are both defined and hold R-related data.

Finite lists also form a relator, $\mathsf{G}\;X = X^*$. Given $B \xleftarrow{R} A$, relator $B^\star \xleftarrow{R^\star} A^\star$ is the relation

$$\begin{array}{lll} s'(R^\star)s & \Leftrightarrow & \mathsf{length}\ s' = \mathsf{length}\ s\ \wedge \\ & & \langle \forall\, i \;:\; 0 \leqslant i < \mathsf{length}\ s :\; (s' \,!!\, i)\ R\ (s \,!!\, i) \rangle \end{array} \tag{6.9}$$

***Exercise** 6.1. Check properties (6.7) and (6.8) for the list relator defined above.*

□

6.4 A RELATION ON FUNCTIONS

The next step needed to postulate free theorems requires a formal understanding of the arrow operator written on the right handside of (6.3).

This is achieved by defining the so-called "Reynolds arrow" relational operator, which establishes a relation on two functions f and g parametric on two other arbitrary relations R and S:

$$f(R \leftarrow S)g \quad \equiv \quad f \cdot S \subseteq R \cdot g \qquad \begin{tikzcd} A \arrow[d, "f"'] \arrow[rd, phantom, "\subseteq"] & B \arrow[l, "S"'] \arrow[d, "g"] \\ C & D \arrow[l, "R"] \end{tikzcd} \tag{6.10}$$

The typing rule is:

$$\frac{A \xleftarrow{S} B \qquad C \xleftarrow{R} D}{C^A \xleftarrow{R \leftarrow S} D^B}$$

This is a powerful operator that satisfies many properties, for instance:

$$\begin{aligned} id \leftarrow id \quad &= \quad id && (6.11)\\ (R \leftarrow S)^\circ \quad &= \quad R^\circ \leftarrow S^\circ && (6.12)\\ R \leftarrow S \quad \subseteq \quad V \leftarrow U \quad &\Leftarrow \quad R \subseteq V \wedge U \subseteq S && (6.13)\\ (R \leftarrow V) \cdot (S \leftarrow U) \quad &\subseteq \quad (R \cdot S) \leftarrow (V \cdot U) && (6.14)\\ (f \leftarrow g^\circ)h \quad &= \quad f \cdot h \cdot g && (6.15)\\ k(f \leftarrow g)h \quad &\equiv \quad k \cdot g = f \cdot h && (6.16) \end{aligned}$$

From property (6.13) we learn that the combinator is monotonic on the left hand side — and thus facts

$$\begin{aligned} S \leftarrow R \quad &\subseteq \quad (S \cup V) \leftarrow R && (6.17)\\ \top \leftarrow S \quad &= \quad \top && (6.18) \end{aligned}$$

hold [2] — and anti-monotonic on the right hand side — and thus property

$$R \leftarrow \bot \quad = \quad \top \tag{6.19}$$

and the two distributive laws which follow:

$$\begin{aligned} S \leftarrow (R_1 \cup R_2) \quad &= \quad (S \leftarrow R_1) \cap (S \leftarrow R_2) && (6.20)\\ (S_1 \cap S_2) \leftarrow R \quad &= \quad (S_1 \leftarrow R) \cap (S_2 \leftarrow R) && (6.21) \end{aligned}$$

It should be stressed that (6.14) expresses *fusion* only, not *fission*.

[2] Cf. $f \cdot S \cdot g^\circ \subseteq \top \Leftrightarrow$ TRUE concerning (6.18).

SUPREMA AND INFIMA Suppose relation R in (6.10) is a complete lattice $(\leqslant)$, that is, it has suprema and infima. What kind of relationship is established between two functions f and g such that

$$f\ ((\leqslant) \leftarrow S)\ g$$

holds? We reason:

$$\begin{array}{cl}
& f\ ((\leqslant) \leftarrow S)\ g \\
\equiv & \quad \{\ (6.10)\ \} \\
& f \cdot S\ \subseteq\ (\leqslant) \cdot g \\
\equiv & \quad \{\ \text{shunting (5.46)}\ \} \\
& S\ \subseteq\ f^{\circ} \cdot (\leqslant) \cdot g \\
\equiv & \quad \{\ \text{go pointwise — (5.17), etc}\ \} \\
& \langle \forall\, a, b\ :\ a\, S\, b : f\ a \leqslant g\ b \rangle \\
\equiv & \quad \{\ \text{quantifier calculus}\ \} \\
& \langle \forall\, b\ ::\ \langle \forall\, a\ :\ a\, S\, b : f\ a \leqslant g\ b \rangle \rangle \\
\equiv & \quad \{\ \text{universal law of suprema}\ \} \\
& \langle \forall\, b\ ::\ \langle \bigvee\ a\ :\ a\, S\, b : f\ a \rangle \leqslant g\ b \rangle \\
\equiv & \quad \{\ \text{introduce supremum, for all } b \text{ (see below)}\ \} \\
& g\ b = \langle \bigvee\ a\ :\ a\, S\, b : f\ a \rangle
\end{array}$$

In summary:

$$f\ ((\leqslant) \leftarrow S)\ g \quad \equiv \quad g\ b = \langle \bigvee\ a\ :\ a\, S\, b : f\ a \rangle \qquad (6.22)$$

In words: *$g\ b$ is the largest of all $(f\ a)$ such that $a\ S\ b$ holds.*

Pattern $(\leqslant) \leftarrow \ldots$ turns up quite often in relation algebra. Consider, for instance, a Galois connection $\alpha \vdash \gamma$ (5.134), that is,

$$\begin{array}{cl}
& \alpha^{\circ} \cdot (\sqsubseteq) = (\leqslant) \cdot \gamma \\
\equiv & \quad \{\ \text{ping pong}\ \} \\
& \alpha^{\circ} \cdot (\sqsubseteq)\ \subseteq\ (\leqslant) \cdot \gamma\ \wedge\ \gamma^{\circ} \cdot (\geqslant)\ \subseteq\ (\sqsupseteq) \cdot \alpha
\end{array}$$

Following the same strategy as just above, we obtain pointwise definitions for the two adjoints of the connection:[3]

$$\gamma\ x = \langle \bigvee\ y\ :\ \alpha\ y \sqsubseteq x :\ y \rangle \qquad (6.23)$$

$$\alpha\ y = \langle \bigsqcap\ x\ :\ y \leqslant \gamma\ x :\ x \rangle \qquad (6.24)$$

[3] Similarly, introducing infimum, for all a: $f\ a = \langle \bigwedge\ b\ :\ a\, S\, b :\ g\ b \rangle$.

6.5 FREE THEOREM OF TYPE t

We are now ready to establish the *free theorem* (FT) of type t, which is the following remarkably simple result:[4]

> *Given any function $\theta : t$, and V as above, then*
>
> $$\theta \, R_t \, \theta$$
>
> *holds, for any relational instantiation of type variables in V.*
> □

Note that this theorem

- is a result about t;
- holds *independently* of the actual definition of θ.

So, it holds about any polymorphic function of type t.

6.6 EXAMPLES

Let us see the simplest of all examples, where the target function is the identity:

$$\theta = id : a \leftarrow a$$

We first calculate $R_{t=a \leftarrow a}$:

$$\begin{array}{cl} & R_{a \leftarrow a} \\ \equiv & \quad \{ \text{ rule } \ R_{t=t' \leftarrow t''} \ = \ R_{t'} \leftarrow R_{t''} \ \} \\ & R_a \leftarrow R_a \end{array}$$

Then we derive the free theorem itself (R_a is abbreviated to R):

$$\begin{array}{cl} & id(R \leftarrow R)id \\ \equiv & \quad \{ \ (6.10) \ \} \\ & id \cdot R \subseteq R \cdot id \end{array}$$

In case R is a function f, the FT theorem boils down to id's *natural* property, $id \cdot f \ = \ f \cdot id$ — recall (2.10) — that can be read alternatively as stating that id is the *unit* of composition.

As a second example, consider $\theta = reverse : a^\star \leftarrow a^\star$, and first calculate $R_{t=a^\star \leftarrow a^\star}$:

$$\begin{array}{cl} & R_{a^\star \leftarrow a^\star} \\ \equiv & \quad \{ \text{ rule } \ R_{t=t' \leftarrow t''} \ = \ R_{t'} \leftarrow R_{t''} \ \} \\ & R_{a^\star} \leftarrow R_{a^\star} \\ \equiv & \quad \{ \text{ rule } \ R_{t=\mathsf{F}(t_1,\ldots,t_n)} \ = \ \mathsf{F}(R_{t_1}, \ldots, R_{t_n}) \ \} \\ & R_a{}^\star \leftarrow R_a{}^\star \end{array}$$

[4] This result is due to J. Reynolds [78], advertised by P. Wadler [85] and re-written by Backhouse [3] in the pointfree style adopted in this book.

where $s\ R^\star s'$ is given by (6.9). Next we calculate the FT itself (R_a abbreviated to R):

$$
\begin{array}{ll}
 & reverse(R^\star \leftarrow R^\star)reverse \\
\equiv & \qquad \{\ \text{definition}\ \ f(R \leftarrow S)g \ \ \equiv\ \ f \cdot S \subseteq R \cdot g\ \ \} \\
 & reverse \cdot R^\star \subseteq R^\star \cdot reverse
\end{array}
$$

In case R is a function r, this FT theorem boils down to *reverse*'s *natural* property,

$$reverse \cdot r^\star \ = \ r^\star \cdot reverse$$

that is, $reverse\ [\ r\ a \mid a \leftarrow l\ ]\ =\ [\ r\ b \mid b \leftarrow reverse\ l\ ]$. For the general case, we obtain:

$$
\begin{array}{ll}
 & reverse \cdot R^\star \subseteq R^\star \cdot reverse \\
\equiv & \qquad \{\ \text{shunting rule (5.46)}\ \} \\
 & R^\star \subseteq reverse^\circ \cdot R^\star \cdot reverse \\
\equiv & \qquad \{\ \text{going pointwise (5.19, 5.17)}\ \} \\
 & \langle \forall\ s, r\ ::\ s\ R^\star r \Rightarrow (reverse\ s)R^\star(reverse\ r)\rangle
\end{array}
$$

An instance of this pointwise version of *reverse*-FT will state that, for example, *reverse* will respect element-wise orderings ($R := <$):[5]

$$
\begin{array}{c}
length\ s = length\ r \wedge \langle \forall\ i\ :\ i \in inds\ s :\ (s\, !!\, i) < (r\, !!\, i)\rangle \\
\Downarrow \\
length(reverse\ s) = length(reverse\ r) \\
\wedge \\
\langle \forall\ j\ :\ j \in inds\ s :\ (reverse\ s\, !!\, j) < (reverse\ r\, !!\, j)\rangle
\end{array}
$$

(Guess other instances.)

As a third example, also involving finite lists, let us calculate the FT of

$$sort : a^\star \leftarrow a^\star \leftarrow (Bool \leftarrow (a \times a))$$

where the first parameter stands for the chosen ordering relation, expressed by a binary predicate:

$$
\begin{array}{ll}
 & sort(R_{(a^\star \leftarrow a^\star) \leftarrow (Bool \leftarrow (a \times a))})sort \\
\equiv & \qquad \{\ \text{(6.2, 6.1, 6.3); abbreviate}\ R_a := R\ \} \\
 & sort((R^\star \leftarrow R^\star) \leftarrow (R_{Bool} \leftarrow (R \times R)))sort \\
\equiv & \qquad \{\ R_{t:=Bool} = id\ \text{(constant relator) — cf. exercise 6.11}\ \} \\
 & sort((R^\star \leftarrow R^\star) \leftarrow (id \leftarrow (R \times R)))sort \\
\equiv & \qquad \{\ \text{(6.10)}\ \}
\end{array}
$$

[5] Let *inds s* denote the set $\{0, \ldots, \text{length}\ s - 1\}$.

$$
\begin{array}{ll}
& sort \cdot (id \leftarrow (R \times R)) \quad \subseteq \quad (R^\star \leftarrow R^\star) \cdot sort \\
\equiv & \quad \{ \text{ shunting (5.46) } \} \\
& (id \leftarrow (R \times R)) \quad \subseteq \quad sort^\circ \cdot (R^\star \leftarrow R^\star) \cdot sort \\
\equiv & \quad \{ \text{ introduce variables } f \text{ and } g \text{ (5.19, 5.17) } \} \\
& f(id \leftarrow (R \times R))g \quad \Rightarrow \quad (sort\ f)(R^\star \leftarrow R^\star)(sort\ g) \\
\equiv & \quad \{ \text{ (6.10) twice } \} \\
& f \cdot (R \times R) \subseteq g \quad \Rightarrow \quad (sort\ f) \cdot R^\star \subseteq R^\star \cdot (sort\ g)
\end{array}
$$

Case $R := r$:

$$
\begin{array}{ll}
& f \cdot (r \times r) = g \quad \Rightarrow \quad (sort\ f) \cdot r^\star = r^\star \cdot (sort\ g) \\
\equiv & \quad \{ \text{ introduce variables } \} \\
& \left\langle \begin{array}{c} \forall\, a, b \ :: \\ f(r\ a, r\ b) = g(a, b) \end{array} \right\rangle \ \Rightarrow \ \left\langle \begin{array}{c} \forall\, l \ :: \\ (sort\ f)(r^\star\ l) = r^\star(sort\ g\ l) \end{array} \right\rangle
\end{array}
$$

Denoting predicates f, g by infix orderings $\leqslant, \preceq$:

$$
\left\langle \begin{array}{c} \forall\, a, b \ :: \\ r\ a \leqslant r\ b \equiv a \preceq b \end{array} \right\rangle \ \Rightarrow \ \left\langle \begin{array}{c} \forall\, l \ :: \\ sort\ (\leqslant)(r^\star\ l) = r^\star(sort\ (\preceq)\ l) \end{array} \right\rangle
$$

That is, for r monotonic and injective,

$$sort\ (\leqslant)\ [\, r\ a \mid a \leftarrow l \,]$$

is always the same list as

$$[\, r\ a \mid a \leftarrow sort\ (\preceq)\ l \,]$$

Exercise *6.2. Let C be a nonempty data domain and let and c* $\in C$*. Let* $\underline{c}$ *be the* "everywhere *c" function* $\underline{c} : A \rightarrow C$ *(2.12). Show that the free theorem of* $\underline{c}$ *reduces to*

$$\langle \forall\, R \ :: \ R \subseteq \top \rangle \tag{6.25}$$

□

Exercise *6.3. Calculate the free theorem associated with the projections*

$$\begin{tikzcd} A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \end{tikzcd}$$

and instantiate it to (a) functions; (b) coreflexives. Introduce variables and derive the corresponding pointwise expressions.

□

Exercise *6.4. As follow-up to exercise 6.2, consider higher order function* $\underline{(\_)} : a \rightarrow b \rightarrow a$ *such that, given any* x *of type* a*, produces the constant function* $\underline{x}$*. Show that the equalities*

$$\underline{f\,x} = f \cdot \underline{x} \tag{6.26}$$

$$\underline{x} \cdot f = \underline{x} \tag{6.27}$$

$$\underline{x}^{\circ} \cdot \underline{x} = \top \tag{6.28}$$

arise as corollaries of the free theorem *of* $\underline{(\_)}$.[6]

□

Exercise *6.5. The following is a well-known Haskell function*

$$\mathsf{filter} :: \forall\, a \cdot (a \rightarrow \mathbb{B}) \rightarrow [a] \rightarrow [a]$$

Calculate the free theorem associated with its type

$$filter : a^{\star} \leftarrow a^{\star} \leftarrow (\mathbb{B} \leftarrow a)$$

and instantiate it to the case where all relations are functions.

□

Exercise *6.6. In many sorting problems, data are sorted according to a given* ranking *function which computes each datum's numeric rank (e.g. students marks, credits, etc). In this context one may parameterize sorting with an extra parameter* f *ranking data into a fixed numeric datatype, e.g. the integers:* $serial : (a \rightarrow \mathbb{N}_0) \rightarrow a^{\star} \rightarrow a^{\star}$*. Calculate the FT of* $serial$*.*

□

Exercise *6.7. Consider the following function from Haskell's Prelude:*

$$findIndices :: (a \rightarrow \mathbb{B}) \rightarrow [a] \rightarrow [\mathbb{Z}]$$
$$findIndices\ p\ xs = [i \mid (x,i) \leftarrow \mathsf{zip}\ xs\ [0\,..],p\ x]$$

which yields the indices of elements in a sequence `xs` *which satisfy* `p`*.*

For instance, $findIndices\ (<0)\ [1,-2,3,0,-5] = [1,4]$*. Calculate the FT of this function.*

□

[6] Note that (6.27) is property (2.14) assumed in chapter 2.

***Exercise** 6.8. Wherever two equally typed functions f, g are such that $f\ a \leqslant g\ a$, for all a, we say that f is* pointwise at most g *and write $f \stackrel{.}{\leqslant} g$,*

$$f \stackrel{.}{\leqslant} g \quad = \quad f \subseteq (\leqslant) \cdot g \qquad \textit{cf. diagram}$$

$$\begin{tikzcd}
& A \arrow[dl, "f"'] \arrow[dr, "g"] \arrow[d, phantom, "\subseteq"] & \\
B & {} & B \arrow[ll, "\leqslant"]
\end{tikzcd}$$

recall (5.93). Show that implication

$$f \stackrel{.}{\leqslant} g \quad \Rightarrow \quad (\mathsf{map}\, f) \stackrel{.}{\leqslant}^{\star} (\mathsf{map}\, g) \tag{6.29}$$

follows from the FT *of the function* $\mathsf{map} : (a \to b) \to a^* \to b^*$.

□

***Exercise** 6.9. Infer the FT of the following function, written in Haskell syntax,*

$$\begin{array}{l}
\mathbf{while} :: (a \to \mathbb{B}) \to (a \to a) \to (a \to b) \to a \to b \\
\mathbf{while}\ p\ f\ g\ x = \mathbf{if}\ \neg\ (p\ x)\ \mathbf{then}\ g\ x\ \mathbf{else}\ \mathbf{while}\ p\ f\ g\ (f\ x)
\end{array}$$

which implements a generic `while`-loop. Derive its corollary for functions.

□

6.7 CATAMORPHISM LAWS AS FREE THEOREMS

Recall from section 3.13 the concept of a catamorphism over a parametric type $\mathsf{T}\ a$:

$$\begin{tikzcd}
\mathsf{T}\ a \arrow[d, "\llparenthesis g \rrparenthesis"'] & \mathsf{B}\,(a, \mathsf{T}\ a) \arrow[l, "in_{\mathsf{T}\ a}"'] \arrow[d, "\mathsf{B}\,(id, \llparenthesis g \rrparenthesis)"] \\
b & \mathsf{B}\,(a, b) \arrow[l, "g"]
\end{tikzcd}$$

So $\llparenthesis \_ \rrparenthesis$ has generic type

$$\llparenthesis \_ \rrparenthesis : b \leftarrow \mathsf{T}\ a \leftarrow (b \leftarrow \mathsf{B}\,(a, b))$$

where $\mathsf{T}\ a \cong \mathsf{B}\,(a, \mathsf{T}\ a)$. Then the free theorem of $\llparenthesis \_ \rrparenthesis$ is

$$\llparenthesis \_ \rrparenthesis \cdot (R_b \leftarrow \mathsf{B}\,(R_a, R_b)) \quad \subseteq \quad (R_b \leftarrow \mathsf{T} R_a) \cdot \llparenthesis \_ \rrparenthesis$$

This unfolds into (R_a, R_b abbreviated to R, S):

$$\llparenthesis \_ \rrparenthesis \cdot (S \leftarrow \mathsf{B}\,(R, S)) \quad \subseteq \quad (S \leftarrow \mathsf{T}\ R) \cdot \llparenthesis \_ \rrparenthesis$$

$$\begin{array}{ll} \equiv & \{ \text{ shunting (5.46) } \} \\ & (S \leftarrow \mathsf{B}\,(R,S)) \quad \subseteq \quad (\!|\_|\!)^{\circ}(S \leftarrow \mathsf{T}\,R) \cdot (\!|\_|\!) \\ \equiv & \{ \text{ introduce variables } f \text{ and } g \text{ (5.19, 5.17) } \} \\ & f(S \leftarrow \mathsf{B}\,(R,S))g \quad \Rightarrow \quad (\!| f |\!)(S \leftarrow \mathsf{T}\,R)(\!| g |\!) \\ \equiv & \{ \text{ definition } \; f(R \leftarrow S)g \;\; \equiv \;\; f \cdot S \subseteq R \cdot g \;\; \} \\ & f \cdot \mathsf{B}\,(R,S) \subseteq S \cdot g \quad \Rightarrow \quad (\!| f |\!) \cdot \mathsf{T}\,R \subseteq S \cdot (\!| g |\!) \end{array}$$

From the calculated free theorem of the catamorphism combinator,

$$f \cdot \mathsf{B}\,(R,S) \subseteq S \cdot g \quad \Rightarrow \quad (\!| f |\!) \cdot \mathsf{T}\,R \subseteq S \cdot (\!| g |\!) \tag{6.30}$$

we can infer:

- $(\!|\_|\!)$-*fusion* $(R,S := id, s)$:

  $$f \cdot \mathsf{B}\,(id,s) = s \cdot g \quad \Rightarrow \quad (\!| f |\!) = s \cdot (\!| g |\!)$$

  — recall (3.70), for $\mathsf{F}\,f = \mathsf{B}\;(id,f)$;

- $(\!|\_|\!)$-*absorption* $(R,S := r, id)$:

  $$f \cdot \mathsf{B}\,(r,id) = g \quad \Rightarrow \quad (\!| f |\!) \cdot \mathsf{T}\,r = (\!| g |\!)$$

  whereby, substituting $g := f \cdot \mathsf{B}\,(r,id)$, we get:

  $$(\!| f |\!) \cdot \mathsf{T}\,r = (\!| f \cdot \mathsf{B}\,(r,id) |\!)$$

  — recall (3.76).

***Exercise** 6.10. Let*

$$iprod = (\!| [\underline{1}\,, (\times)] |\!)$$

be the function that multiplies all natural numbers in a given list, and even be the predicate which tests natural numbers for evenness. Finally, let

$$exists = (\!| [\underline{\text{FALSE}}\,, (\vee)] |\!)$$

be the function that implements existential quantification over a list of Booleans.

From (6.30) infer

$$even \cdot iprod \quad = \quad exists \cdot even^{\star}$$

meaning that the product $n_1 \times n_2 \times \ldots \times n_m$ *is even if and only if some* n_i *is so.*

□

***Exercise** 6.11. Show that the* identity *relator* Id, *which is such that* $\;\mathsf{Id}\,R \;\; = \;\; R$ *and the* constant *relator* K *(for a given data type K) which is such that* $\;\mathsf{K}\,R \;\; = \;\; id_K$

are indeed relators.

□

***Exercise** 6.12.Show that product*

$$\begin{tikzcd}
A \arrow[d, "R"'] & C \arrow[d, "S"'] \arrow[r, dotted, no head] & \mathsf{G}(A,C) = A \times C \arrow[d, "\mathsf{G}(R,S)=R\times S"] \\
B & D \arrow[r, dotted, no head] & \mathsf{G}(B,D) = B \times D
\end{tikzcd}$$

is a (binary) relator.

□

6.8 BIBLIOGRAPHY NOTES

The free theorem of a polymorphic function is a result due to computer scientist John Reynolds [78]. It became popular under the "theorems for free" heading coined by Phil Wadler [85]. The original pointwise setting of this result was re-written in the pointfree style in [3] thanks to the *relation on functions* combinator (6.10) first introduced by Roland Backhouse in [4].

More recently, Janis Voigtlaender devoted a whole research project to free theorems, showing their usefulness in several areas of computer science [55]. One outcome of this project was an automatic generator of free theorems for types written in Haskell syntax. This is (was?) available from Janis Voigtlaender's home page:

`http://www-ps.iai.uni-bonn.de/ft`

The relators used in the calculational style followed in this book are implemented in this automatic generator by so-called structural functor *lifting*.

7

CONTRACT-ORIENTED PROGRAMMING

The chapters of the first part of this book rely on a type-polymorphic notion of computation, captured by the omnipresent use of the arrow notation

$$B \xleftarrow{f} A$$

where A and B are *types*.

The generalization from functions to relations carried out in the previous two chapters has preserved the same principle — all relational combinators are typed in the same way. There is thus an implicit assumption of *static type checking* in the overall approach — types are checked at "compile time". Expressions which don't type are automatically excluded.

However, examples such as the Alcuin puzzle show that this is insufficient. Why? Because the types involved are most often "too large": the whole purpose of the puzzle is to consider only the inhabitants of type $Bank^{Being}$ — functions that describe all possible configurations in the puzzle — that satisfy the "starvation property", recall (5.76). Moreover, the *carry* _ operation (5.200) *should* preserve this property — something we didn't at all check in the previous chapter!

Let us generalize the situation in this puzzle to that of a function $f : A \rightarrow A$ and a predicate $p : A \rightarrow \mathbb{B}$ that should be preserved by f. Predicates such as p have become known as *invariants* by software theorists. The preservation requirement is captured by:

$$\langle \forall\, a \; : \; p\, a : \; p\ (f\ a) \rangle$$

Note how the type A is now divided in two parts — a "good one", $\{a \mid a \in A \wedge p\ a\}$ and a "bad one", $\{a \mid a \in A \wedge \neg\ (p\ a)\}$. By identifying p as an invariant, the programmer is *obliged* to ensure a "good" output $f\ a$ wherever a "good" input is passed to f. For "bad" inputs nothing is requested.

The situation above can be generalized to some $f : A \rightarrow B$ where B is subject to some invariant $q : B \rightarrow \mathbb{B}$. So f is *obliged* to ensure "good" outputs satisfying q. It may well be the case that the only way for f to ensure "good" outputs is to restrict its inputs by some precondition $p : A \rightarrow \mathbb{B}$. Thus the proof obligation above generalizes to:

$$\langle \forall\, a \; : \; p\, a : \; q\ (f\ a) \rangle \tag{7.1}$$

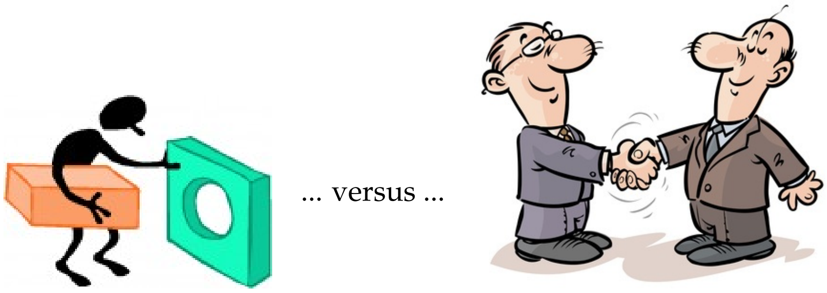


Figure 7.1.: The contract-oriented programming metaphor.

One might tentatively try and express this requirement by writing

$$p \xrightarrow{f} q$$

where predicates p and q take the place of the original types A and B, respectively. This is what we shall do, calling assertion $p \xrightarrow{f} q$ a *contract*. Note how are are back to the function-as-a-contract view of section 2.1 but in a wider setting:

> f *commits itself* to producing a "good" B-value (wrt. q) provided it is supplied with a "suitable" A-value (wrt. p).

The main difference compared to section 2.1 is that the well-typing of $p \xrightarrow{f} q$ cannot be mechanically ascertained at "compile time" — it has to be validated by a formal proof — the proof obligation (7.1) mentioned above. This kind of type checking is often referred to as "extended type checking".

In real life software design data type invariants can be arbitrarily complex — think of all legal restrictions imposed on the organized societies of today! The increasing "softwarization" of our times forces us to think that, as in the regular functioning of such organized *societies*, programs should interact with each other via *formal contracts* establishing what they rely upon or guarantee among themselves. This is the only way to ensure *safety* and *security* essential to reliable, mechanized operations.

This chapter will use relation algebra to describe such contracts and develop a simple theory about them, enabling compositionality as before. Relations (including functions) will play a double role — they will not only describe computations but also the data structures involved in such computations, in a unified and elegant way.

7.1 CONTRACTS

It should be routine work for the reader to check that

$$f \cdot \Phi_p \subseteq \Phi_q \cdot f \tag{7.2}$$

means exactly the same as (7.1) above, and that it can be expressed by

$$f\ (\Phi_q \leftarrow \Phi_p)\ f \tag{7.3}$$

in the arrow-notation of (6.10). In software design terminology, this is known as a (functional) *contract*, and we shall write

$$p \xrightarrow{f} q \tag{7.4}$$

to denote it — a notation that generalizes the type $A \rightarrow B$ of f, as already observed. Thanks to (5.210), (7.2) can also be written:

$$f \cdot \Phi_p \ \subseteq\ \Phi_q \cdot \top \tag{7.5}$$

Predicates p and q in contract $p \xrightarrow{f} q$ shall be referred to as the contract's *precondition* and *postcondition*, respectively. Contracts compose sequentially, see the following exercise.

WEAKEST PRE-CONDITIONS Note that more than one (*pre*) condition p may ensure (*post*) condition q on the outputs of f. Indeed, contract $\mathit{false} \xrightarrow{f} q$ always holds, but it is useless — pre-condition *false* is *"unacceptably strong"*.

Clearly, the weaker p the better. The question is, then: is there a *weakest* such p? We calculate:

$$\begin{array}{ll}
 & f \cdot \Phi_p \ \subseteq\ \Phi_q \cdot f \\
\equiv & \qquad \{\ \text{recall (5.210)}\ \} \\
 & f \cdot \Phi_p \ \subseteq\ \Phi_q \cdot \top \\
\equiv & \qquad \{\ \text{shunting (5.46); (5.208)}\ \} \\
 & \Phi_p \ \subseteq\ f^\circ \cdot \dfrac{\mathit{true}}{q} \\
\equiv & \qquad \{\ \text{(5.52)}\ \} \\
 & \Phi_p \ \subseteq\ \dfrac{\mathit{true}}{q \cdot f} \\
\equiv & \qquad \{\ \Phi_p \ \subseteq\ id\ ;\text{(5.58)}\ \} \\
 & \Phi_p \ \subseteq\ id \cap \dfrac{\mathit{true}}{q \cdot f} \\
\equiv & \qquad \{\ \text{(5.201)}\ \} \\
 & \Phi_p \ \subseteq\ \Phi_{(q \cdot f)}
\end{array}$$

We conclude that $q \cdot f$ is such a *weakest* pre-condition. Notation $wp\ (f, q) = q \cdot f$ is often used to denote a *weakest* pre-condition (WP). This is the weakest constraint on inputs for outputs by f to fall within q. The special situation of a weakest precondition is nicely captured by the universal property:

$$f \cdot \Phi_p = \Phi_q \cdot f \quad \equiv \quad p = q \cdot f \tag{7.6}$$

where $p = wp\ (f, q)$ could be written instead of $p = q \cdot f$, as seen above. Property (7.6) enables a "logic-free" calculation of weakest preconditions, as we shall soon see: given f and post-condition q, there always exists a unique (weakest) precondition p such that $\Phi_q \cdot f$ can be replaced by $f \cdot \Phi_p$. Moreover:

$$\frac{f}{f} \cdot \Phi_p = \Phi_p \cdot \frac{f}{f} \quad \Leftarrow \quad p \leqslant f \tag{7.7}$$

where $\leqslant$ denotes the injectivity preorder (5.234) on functions.[1]

Exercise *7.1. Calculate the weakest pre-condition $wp\ (f, q)$ for the following function / post-condition pairs:*

- $f\ x = x^2 + 1\ ,\ q\ y = y \leqslant 10$ *(in* $\mathbb{R}$*)*
- $f = \ \mathbb{N}_0 \xrightarrow{\mathsf{succ}} \mathbb{N}_0\ \ ,\ q = even$
- $f\ x = x^2 + 1\ ,\ q\ y = y \leqslant 0$ *(in* $\mathbb{R}$*)*

□

COMPOSITIONALITY FOR FREE The fact that functional contracts compose with each other, that is,

$$q \xleftarrow{h \cdot f} p \ \text{ holds provided } \ r \xleftarrow{f} p \ \text{ and } \ q \xleftarrow{h} r \ \text{ hold} \tag{7.8}$$

does not need to be proved: it is a corollary of the *free theorem* (section 6.5) of composition itself, which unfolds to

$$\left\{ \begin{array}{l} f \cdot R \ \subseteq \ S \cdot g \\ h \cdot S \ \subseteq \ Q \cdot k \end{array} \right. \Rightarrow (h \cdot f) \cdot R \ \subseteq \ Q \cdot (k \cdot g) \tag{7.9}$$

for suitably typed f, g, h, k, R, S, Q. To get (7.8) from (7.9) consider the substitutions $g, k := f, h$ in (7.9), which lead to:

$$\left\{ \begin{array}{l} S \xleftarrow{f} R \\ Q \xleftarrow{h} S \end{array} \right. \Rightarrow \ Q \xleftarrow{h \cdot f} R \tag{7.10}$$

Further substituting $R, S, Q := \Phi_p, \Phi_r, \Phi_q$ immediately yields (7.8).

Interestingly, if we regard relations R, S, Q in (7.10) as preorders, then (7.10) will tell that the composition of monotonic functions h and f is monotonic. The free theorem (7.9) captures even more basic properties of composition: for the substitution $R, S, Q := r, id, id$ one gets

$$\left\{ \begin{array}{l} f \cdot r = g \\ h = k \end{array} \right. \Rightarrow (h \cdot f) \cdot r = k \cdot g$$

and then, immediately:

$$(h \cdot f) \cdot r = h \cdot (f \cdot r)$$

This is the associative law of composition, recall (2.8).

[1] The interested reader will find the proofs of (7.6) and (7.7) in reference [72].

INVARIANTS In case $p = q$ in a contract (7.4), that is, in case of $q \xrightarrow{f} q$ holding, we say that q is an *invariant* of f, meaning that the "truth value" of q remains unchanged by execution of f. More generally, invariant q is *preserved* by function f provided contract $p \xrightarrow{f} q$ holds and $p \Rightarrow q$, that is, $\Phi_p \subseteq \Phi_q$.

Some pre-conditions are weaker than others wrt. invariant preservation. We shall say that w is the *weakest* pre-condition for f to preserve *invariant* q wherever $wp\ (f, q) = w \wedge q$, where $\Phi_{p \wedge q} = \Phi_p \cdot \Phi_q$.

Recalling the Alcuin puzzle, let us define the *starvation* invariant as a predicate on the state of the puzzle, passing the *where* function as a parameter w:

$$starving\ w \quad = \quad w \cdot CanEat \ \subseteq\ w \cdot \underline{Farmer}$$

Then the *contract*

$$starving \xrightarrow{carry\ b} starving$$

would mean that the function *carry b* — that should transfer the beings in b to the other bank of the river — always preserves the invariant:

$$wp\ (carry\ b, starving) = starving.$$

Things are not that easy, however: there is a need for a *pre-condition* ensuring that b includes the farmer together with a good choice of the being to carry!

Let us see some simpler examples first.

7.2 LIBRARY LOAN EXAMPLE

Consider the following relational data model of a library involving books and users that can borrow its books:

$$\begin{tikzcd}
 & ISBN & Name & \\
Title & Book \arrow[l, "title"'] \arrow[u, "isbn"'] \arrow[r, "R"] \arrow[d, "Auth"'] & User \arrow[u, "name"'] \arrow[r, "addr"'] \arrow[d, "card"] & Address \\
 & Author & Id &
\end{tikzcd} \tag{7.11}$$

All arrows denote attributes (functions) but two — *Auth* and R. The former is a relation because a book can have more that one author.[2] The latter is the most interesting relation of the model, $u\ R\ b$ meaning "book b currently on loan to library user u". Quite a few invariants are required in this model, for instance:

- *the same book cannot be not on loan to more than one user;*

[2] Its power transpose (5.250) — $\Lambda Auth : Book \rightarrow \mathsf{P} Author$ — gives the *set* of authors of a book.

- *no book exists with no authors;*
- *no two different users have the same card Id;*
- *books with the same ISBN should have the same title and the same authors.*

Such properties (invariants) are easy to encode:

- no book on loan to more than one user:

  $Book \xrightarrow{R} User$ is *simple*
- no book without an author:

  $Book \xrightarrow{Auth} Author$ is *entire*
- no two users with the same card Id:

  $User \xrightarrow{card} Id$ is *injective*
- *ISBN* is a *key* attribute:

  $ISBN \xrightarrow{title \cdot isbn^\circ} Title$ and $ISBN \xrightarrow{\Lambda Auth \cdot isbn^\circ} \mathsf{P}\ Author$ are *simple* relations.

Since all other arrows are functions, they are simple and entire.

Let us now spell out such invariants in terms of relational assertions (note the role of the injectivity preorder):

- no book on loan to more than one user:

  $id \leqslant R^\circ$

  equivalent to $\mathsf{img}\ R \subseteq id$;
- no book without an author:

  $$id \subseteq \mathsf{ker}\ Auth$$
- no two users with the same card Id:

  $$id \leqslant card$$

  equivalent to $\mathsf{ker}\ card \subseteq id$.
- *ISBN* is a *key* attribute:

  $$title \leqslant isbn \ \wedge\ \Lambda Auth \leqslant isbn$$

  equivalent to $\frac{isbn}{isbn} \subseteq \frac{title}{title}$ and $\frac{isbn}{isbn} \subseteq \frac{Auth}{Auth}$, respectively.[3]

[3] Note the use of (5.172) in the second case.

Below we focus on the first invariant, *no book on loan to more than one user*. To bring life to our model, let us think of two operations on $User \xleftarrow{R} Book$, one that *returns* books to the library and another that *records* new borrowings:

$$(\mathsf{return}\ S)\ R = R - S \tag{7.12}$$

$$(borrow\ S)\ R = S \cup R \tag{7.13}$$

Note that parameter S is of type $User \xleftarrow{R} Book$, indicating which users borrow/return which books. Clearly, these operations only change the *books-on-loan* relation R, which is conditioned by invariant

$$inv\ R \ = \ \mathsf{img}\ R \subseteq id \tag{7.14}$$

The question is, then: are the following "types"

$$inv \xleftarrow{\mathsf{return}\ S} inv \tag{7.15}$$

$$inv \xleftarrow{borrow\ S} inv \tag{7.16}$$

valid? Let us check (7.15):

$$\begin{array}{ll}
& inv\ (\mathsf{return}\ S\ R) \\
\equiv & \quad \{ \text{ inline definitions } \} \\
& \mathsf{img}\ (R - S) \subseteq id \\
\Leftarrow & \quad \{ \text{ since } \mathsf{img} \text{ is monotonic } \} \\
& \mathsf{img}\ R \subseteq id \\
\equiv & \quad \{ \text{ definition } \} \\
& inv\ R \\
\Box &
\end{array}$$

So, for all R, $inv\ R \Rightarrow inv\ (\mathsf{return}\ S\ R)$ holds — invariant *inv* is preserved.

At this point note that (7.15) was checked only as a *warming-up* exercise — we don't actually need to worry about it! Why?

> As $R - S$ is smaller than R (exercise 5.42) and *"smaller than injective is injective"* (5.82), it is immediate that *inv* (7.14) is preserved.

To see this better, we unfold and draw definition (7.14) in the form of a diagram:

$$inv\ R \ = \ \begin{tikzcd} Book \arrow[d, "R"'] & User \arrow[l, "R^\circ"'] \arrow[d, "id"] \arrow[dl, phantom, "\subseteq"] \\ User & User \arrow[l, "id"] \end{tikzcd}$$

As R occurs only in the lower-path of the diagram, it can always get smaller.

This "rule of thumb" does not work for *borrow* S because, in general, $R \subseteq borrow\ S\ R$. This time R gets bigger, not smaller, and we do have to check the contract:

$$
\begin{array}{ll}
& inv\ (borrow\ S\ R) \\
\equiv & \qquad \{\ \text{inline definitions}\ \} \\
& \mathsf{img}\ (S \cup R) \subseteq id \\
\equiv & \qquad \{\ \text{exercise 5.15}\ \} \\
& \mathsf{img}\ R \subseteq id \wedge \mathsf{img}\ S \subseteq id \wedge S \cdot R^{\circ} \subseteq id \\
\equiv & \qquad \{\ \text{definition of } inv\ \} \\
& inv\ R \wedge \underbrace{\mathsf{img}\ S \subseteq id \wedge S \cdot R^{\circ} \subseteq id}_{wp\ (borrow\ S, inv)}
\end{array}
$$

Thus the complete definition of the *borrow* operation becomes, in the notation of section 5.3:

$$
\begin{array}{l}
Borrow\ (S, R : Book \rightharpoonup User)\ R' : Book \rightharpoonup User \\
\mathsf{pre}\ S \cdot S^{\circ} \subseteq id \wedge S \cdot R^{\circ} \subseteq id \\
\mathsf{post}\ R' = R \cup S
\end{array}
$$

Why have we written *Borrow* instead of *borrow* as before? This is because *borrow* has become a *simple* relation

$$Borrow = borrow \cdot \Phi_{\mathsf{pre}}$$

It is no longer a function since its (weakest) precondition is not the predicate *true*. (Recall that lowercase identifiers are reserved to functions only.) This precondition was to be expected, as spelt out by rendering $S \cdot R^{\circ} \subseteq id$ in pointwise notation: for all users u, u',

$$\langle \exists\ b\ :\ u\ S\ b :\ u'\ R\ b \rangle \Rightarrow u = u'$$

should hold. So, after the operation takes place, the result state $R' = R \cup S$ won't have the same book on loan twice to different users. (Of course, the same must happen about S itself, which is the same predicate for $R = S$.) Interestingly, the weakest precondition is not ruling out the situation in which $u\ S\ b$ and $u\ R\ b$ hold, for some book b and user u. Not only this does not harm the model but also it corresponds to a kind of renewal of a previous borrowing.

EVOLUTION The library loan model (7.11) given above is not realistic in the following sense — it only "gives life" to the borrowing relation R. In a sense, it assumes that all books have been bought and all users are registered.

How do we improve the model so that new books can be acquired and new users can join the library? Does this evolution entail a complete revision of (7.11)? Not at all. What we have to do is to *add* two new relations, say M and N, the first recording the books currently available in the library and the second the users currently registered for loaning:

$$\begin{tikzcd}
 & ISBN & & & Name & \\
Title & Book \arrow[l, "title"'] \arrow[u, "isbn"'] \arrow[d, "Auth"] & \#B \arrow[l, "M"'] \arrow[r, "R"] & \#U \arrow[r, "N"] & User \arrow[u, "name"'] \arrow[r, "addr"'] \arrow[d, "card"'] & Address \\
 & Author & & & Id &
\end{tikzcd}$$

Two new datatypes have been added: $\#U$ (unique identifier of each user) and $\#B$ (key identifying each book). Relations M and N have to be simple. The operations defined thus far stay the same, provided $\#B$ replaces *Book* and $\#U$ replaces *User* — advantages of a polymorphic notation. New operations can be added for

- acquiring new books — will change relation M only;
- registering new users — will change relation N only;
- cancelling users' registrations — will change relation N only.

There is, however, something that has not been considered: think of a starting state where $M = \bot$ and $N = \bot$, that is, the library has no users, no books yet. Then necessarily $R = \bot$. In general, users cannot borrow books that don't exist,

$$\delta\, R \ \subseteq\ \delta\, M$$

and not-registered users cannot borrow books at all:

$$\rho\, R \ \subseteq\ \delta\, N$$

Invariants of this kind capture so-called *referential integrity* constraints. They can be written with less symbols, cf.

$$R \ \subseteq\ \top \cdot M$$

and

$$R \ \subseteq\ N^\circ \cdot \top$$

respectively. Using the "thumb" rules as above, it is clear that, with respect to *referential integrity*:

- returning books is no problem, because R is only on the lower side of both inclusions;
- *borrow*ing books calls for new contracts — R is on the lower side and it increases!

- registering new users and buying new books are no problem, because M and N are on the upper side only;
- unregistering users calls for a contract because N is on the upper side and decreases — users must return all books before unregistering!

7.3 MOBILE PHONE EXAMPLE

In this example we go back to the *store* operation on a mobile phone list of calls specified by (5.2). Of the three invariants we select (b), the one requiring no duplicate calls in the list. Recall, in Haskell, the function $(!!) :: [a] \rightarrow \mathbb{Z} \rightarrow a$. This tells how a finite list s is converted into a partial function $(s!!)$ of type $\mathbb{Z} \rightarrow a$. In fact, the partiality extends to the negative numbers[4] and so we should regard $(s!!)$ as a *simple* relation[5] even if restricted to the type $a \leftarrow \mathbb{N}_0$, as we shall do below.

The no-duplicates requirement requests $(s!!)$ to be injective: in case $s\,!!\,i$ and $s\,!!\,j$ are defined, $i \neq j \Rightarrow s\,!!\,i \neq s\,!!\,j$. Let $L = (s!!)$. Then we can re-specify the operations of *store* in terms of L, as follows:[6]

$$\begin{array}{l} \mathit{inv}\ L = \mathit{id} \leqslant L \\ \mathsf{filter}\ (c \neq)\ L = L - \underline{c} \\ c : L = [\underline{c}, L] \cdot \mathsf{in}^{\circ} \end{array}$$

where $\mathsf{in} = [\underline{0}, \mathsf{succ}]$ — the Peano algebra which builds up natural numbers.[7] By (5.121) the definition of $c : L$ can also be written $\underline{c} \cdot \underline{0}^{\circ} \cup L \cdot \mathsf{succ}^{\circ}$, explicitly telling that c is placed in position 0 while L is shifted one position up to make room for the new element. We calculate:

$$\begin{array}{cl} & \mathit{inv}\ (c : (\mathsf{filter}\ (c \neq)\ L) \\ \equiv & \quad \{\ \mathit{inv}\ L = \mathit{id} \leqslant L, \text{ using the injectivity preorder }\} \\ & \mathit{id} \leqslant c : (\mathsf{filter}\ (c \neq)\ L) \\ \equiv & \quad \{\ \text{in-line definitions }\} \\ & \mathit{id} \leqslant [\underline{c}, L - \underline{c}] \cdot \mathsf{in}^{\circ} \\ \equiv & \quad \{\ \text{Galois connection (5.237) }\} \\ & \mathsf{in} \leqslant [\underline{c}, L - \underline{c}] \\ \equiv & \quad \{\ \text{(5.126) ; } \mathsf{in} \text{ is as injective as } \mathit{id}\ \} \\ & \mathit{id} \leqslant \underline{c} \wedge \mathit{id} \leqslant L - \underline{c} \wedge \underline{c}^{\circ} \cdot (L - \underline{c}) \ \subseteq\ \bot \\ \Leftarrow & \quad \{\ \text{constant function } \mathbb{N}_0 \xleftarrow{\underline{c}} 1 \text{ is injective; } L \subseteq \top\ \} \\ & \mathit{id} \leqslant L - \underline{c} \wedge \underline{c}^{\circ} \cdot (\top - \underline{c}) \ \subseteq\ \bot \end{array}$$

[4] Try $[2,3,3]\ !!\ (-1)$, for instance.

[5] Partial functions are *simple* relations, as we know.

[6] Knowing that $\mathsf{take}\ 10$ will always yield its input or a smaller list, and that *smaller than injective is injective* (5.82), we only need to focus on $(c:) \cdot \mathsf{filter}\ (c \neq)$.

[7] Recall section 3.1.

$$
\begin{array}{ll}
\Leftarrow & \{ \text{ smaller than injective is injective ; } \underline{c}^\circ \cdot (\top - \underline{c}) = \bot \text{ (5.157) } \} \\
& id \leqslant L
\end{array}
$$

□

Having given two examples of contract checking in two quite different domains, let us prepare for checking that of the Alcuin puzzle. By exercise 5.20 we already know that any of the starting states $w = \underline{Left}$ or $w = \underline{Right}$ satisfy the invariant:

$$starving\ w = w \cdot CanEat\ \subseteq\ w \cdot \underline{Farmer}.$$

The only operation defined is

$$carry\ who\ where = (\in who) \rightarrow cross \cdot where\ ,\ where$$

Clearly, calculating the weakest precondition for this operation to preserve *starving* is expected to be far more complex than in the previous examples, since *where* is everywhere in the invariant. Can this be made simpler?

The answer is positive provided we understand a technique to be adopted, called *abstract interpretation*. So we postpone the topic of this paragraph to section 7.6, where abstract interpretation will be introduced. In between, we shall study a number of rules that can be used to address contracts in a structured way.

***Exercise** 7.2. Consider the voting system described by the relations of the diagram below,*

$$\begin{tikzcd}
C \arrow[r, "p"'] \arrow[rr, bend left, "V'"] \arrow[dr, "dC"'] & P \arrow[r, "V"'] \arrow[d, "Di"'] & E \arrow[dl, "dE"] \\
& D &
\end{tikzcd}$$

where electors can vote in political parties or nominally in members of such parties. In detail: (a) $p\ c$ denotes the party of candidate c; (b) $dC\ c$ denotes the district of candidate c; (c) $dE\ e$ denotes the district of elector e; (d) $d\ Di\ p$ records that party p has a list of candidates in district d; (e) $e\ V\ p$ indicates that elector e voted in party p; (f) $e\ V'\ c$ indicates that elector e voted nominally in candidate c.

There are several invariants to take into account in this model, namely:

$$inv1\ (V, V') = V : E \leftarrow P \text{ and } V' : E \leftarrow C \text{ are injective} \tag{7.17}$$

$$inv2\ (V, V') = V^\circ \cdot V' = \bot \tag{7.18}$$

since an elector cannot vote in more than one candidate or party;

$$inv3\ (V, V')\ \ =\ \ dE \cdot [V, V']\ \subseteq\ [Di, dC] \tag{7.19}$$

since each elector is registered in one district and can only vote in candidates of that district.

When the elections take place, relations p, dC, dE and Di are static, since all lists and candidates are fixed before people can vote. Once it is over, the scrutinity of the votes is carried out by function

$$\mathit{batch}\ (V, V', X) = \ldots$$

where $X : E \rightarrow (P + C)$ *is a batch of votes to be loaded into the system.*

Complete the definition of batch and discharge the proof obligations of the contracts that this function must satisfy.

□

7.4 A CALCULUS OF FUNCTIONAL CONTRACTS

The number and complexity of invariants in real life problems invites us to develop *divide & conquer* rules alleviating the proof obligations that have to be discharged wherever contracts are needed. All such rules have definition (7.2) as starting point. Let us see, for instance, what happens wherever the input predicate in (7.4) is a disjunction:

$$\begin{array}{ll}
& \Phi_q \xleftarrow{f} \Phi_{p_1} \cup \Phi_{p_2} \\
\equiv & \quad \{ \ (7.3) \ \} \\
& f\ (\Phi_q \leftarrow (\Phi_{p_1} \cup \Phi_{p_2}))\, f \\
\equiv & \quad \{ \ (6.20) \ \} \\
& f\ (\Phi_q \leftarrow \Phi_{p_1} \cap \Phi_q \leftarrow \Phi_{p_2})\, f \\
\equiv & \quad \{ \ (5.56) \ \} \\
& f\ (\Phi_q \leftarrow \Phi_{p_1})\, f \wedge f\ (\Phi_q \leftarrow \Phi_{p_2})\, f \\
\equiv & \quad \{ \ (7.3) \text{ twice } \} \\
& \Phi_q \xleftarrow{f} \Phi_{p_1} \ \wedge \ \Phi_q \xleftarrow{f} \Phi_{p_2}
\end{array}$$

Recall that the disjunction $p \vee q$ of two predicates is such that $\Phi_{p \vee q} = \Phi_p \cup \Phi_q$ holds. Se we can write the result above in the simpler notation (7.4) as the contract decomposition rule:

$$q \xleftarrow{f} p \vee r \quad \equiv \quad q \xleftarrow{f} p \ \wedge \ q \xleftarrow{f} r \tag{7.20}$$

The dual rule,

$$\Phi_q \cdot \Phi_r \xleftarrow{f} \Phi_p \quad \equiv \quad \Phi_q \xleftarrow{f} \Phi_p \ \wedge \ \Phi_r \xleftarrow{f} \Phi_p$$

is calculated in the same way — via (6.21) — and written

$$q \wedge r \xleftarrow{f} p \quad \equiv \quad q \xleftarrow{f} p \ \wedge \ r \xleftarrow{f} p \tag{7.21}$$

in the same notation, since $\Phi_{p \wedge q} = \Phi_p \cap \Phi_q$. The fact that contracts compose sequentially (7.8) enables the corresponding decomposition, once a suitable middle predicate r is found:

$$q \xleftarrow{g \cdot h} p \quad \Leftarrow \quad q \xleftarrow{g} r \ \wedge \ r \xleftarrow{h} p \tag{7.22}$$

This follows straight from (7.4, 7.2), as does the obvious rule concerning identity

$$q \xleftarrow{id} p \quad \equiv \quad q \Leftarrow p \tag{7.23}$$

since $p \Rightarrow q \Leftrightarrow \Phi_p \subseteq \Phi_q$. The expected

$$p \xleftarrow{id} p$$

immediately follows from (7.23).

Now suppose that we have contracts $q \xleftarrow{f} p$ and $r \xleftarrow{g} p$. What kind of contract can we infer for $\langle f,g \rangle$? We calculate:

$$\Phi_q \xleftarrow{f} \Phi_p \quad \wedge \quad \Phi_r \xleftarrow{g} \Phi_p$$

$$\equiv \qquad \{ \text{ (7.4,7.2) twice } \}$$

$$f \cdot \Phi_p \subseteq \Phi_q \cdot f \quad \wedge \quad g \cdot \Phi_p \subseteq \Phi_r \cdot g$$

$$\equiv \qquad \{ \text{ cancellations (2.22) } \}$$

$$\pi_1 \cdot \langle f,g \rangle \cdot \Phi_p \subseteq \Phi_q \cdot f \quad \wedge \quad \pi_2 \cdot \langle f,g \rangle \cdot \Phi_p \subseteq \Phi_r \cdot g$$

$$\equiv \qquad \{ \text{ universal property (5.103) } \}$$

$$\langle f,g \rangle \cdot \Phi_p \subseteq \langle \Phi_q \cdot f, \Phi_r \cdot g \rangle$$

$$\equiv \qquad \{ \text{ absorption (5.106) } \}$$

$$\langle f,g \rangle \cdot \Phi_p \subseteq (\Phi_q \times \Phi_r) \cdot \langle f,g \rangle$$

$$\equiv \qquad \{ \text{ (7.4,7.2) } \}$$

$$\Phi_q \times \Phi_r \xleftarrow{\langle f,g \rangle} \Phi_p$$

Defining $p \boxtimes q$ such that $\Phi_{p \boxtimes q} = \Phi_p \times \Phi_q$ we obtain the contract decomposition rule:

$$q \boxtimes r \xleftarrow{\langle f,g \rangle} p \quad \equiv \quad q \xleftarrow{f} p \wedge r \xleftarrow{g} p \tag{7.24}$$

which justifies the existence of arrow $\langle f,g \rangle$ in the diagram

$$\begin{tikzcd}
q & q \boxtimes r \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & r \\
& p \arrow[ul, "f"] \arrow[u, "\langle f,g \rangle"] \arrow[ur, "g"'] &
\end{tikzcd} \tag{7.25}$$

where predicates (coreflexives) are promoted to objects (nodes in diagrams).

***Exercise** 7.3. Check the contracts $q \xleftarrow{\pi_1} q \boxtimes r$ and $q \boxtimes r \xrightarrow{\pi_2} r$ of diagram (7.25).*

□

Let us finally see how to handle conditional expressions of the form $if\ (c\ x)\ then\ (f\ x)\ else\ (g\ x)$ which, by (5.215), transform into

$$c \to f\ ,\ g\ =\ f \cdot \Phi_c \cup g \cdot \Phi_{\neg c} \tag{7.26}$$

In this case, (7.5) offers a better standpoint for calculation than (7.2), as the reader may check in calculating the following rule for conditionals:

$$\Phi_q \xleftarrow{c \to f\ ,\ g} \Phi_p \quad \equiv \quad \left\{ \begin{array}{l} \Phi_q \xleftarrow{f} \Phi_p \cdot \Phi_c \\ \Phi_q \xleftarrow{g} \Phi_p \cdot \Phi_{\neg\ c} \end{array} \right. \tag{7.27}$$

This is because it is hard to handle $c \to f\ ,\ g$ on the upper side, $\top$ being more convenient.

Further contract rules can calculated on the same basis, either by elaborating on the predicate structure or on the combinator structure. However, all the cases above involve functions only and the semantics of computations are, in general, relations. So our strategy is to generalize definition (7.2) from functions to arbitrary relations.

RELATIONAL CONTRACTS Note that $S = R \cdot \Phi_p$ means

$$b\ S\ a\ \Leftrightarrow\ p\ a \wedge b\ R\ a$$

—- that is, S is R pre-conditioned by p. Dually, $\Phi_q \cdot R$ is the largest part of R which yields outputs satisfying q — R post-conditioned by q. By writing

$$R \cdot \Phi_p\ \subseteq\ \Phi_q \cdot R \tag{7.28}$$

— which is equivalent to

$$R \cdot \Phi_p\ \subseteq\ \Phi_q \cdot \top \tag{7.29}$$

by (5.210) and even equivalent to

$$\Phi_p\ \subseteq\ R \setminus (\Phi_q \cdot \top) \tag{7.30}$$

by (5.161) — we express a very important fact about R regarded as a (possibly non-deterministic, undefined) program R: condition p on the inputs is *sufficient* for condition q to hold on the outputs:

$$\langle \forall\ a\ :\ p\ a :\ \langle \forall\ b\ :\ b\ R\ a :\ q\ b \rangle \rangle$$

Thus we generalize functional contracts (7.2) to arbitrary relations,

$$p \xrightarrow{R} q\ \ \equiv\ \ R \cdot \Phi_p\ \subseteq\ \Phi_q \cdot R \tag{7.31}$$

a definition equivalent to

$$p \xrightarrow{R} q\ \ \equiv\ \ R \cdot \Phi_p\ \subseteq\ \Phi_q \cdot \top \tag{7.32}$$

as seen above.

Exercise *7.4. In a relational contract* $q \xleftarrow{R} p$ *(7.31), for* $R : A \to B$*, it may happen that, for some inputs* $a \in A$ *satisfying* p *(that is,* $p\ a = \mathsf{True}$ *for such inputs)* R *does not react, which embodies a slight contradiction. To avoid this the following additional constraint is often required,*

$$\Phi_p \subseteq R^\circ \cdot \Phi_q \cdot R \tag{7.33}$$

known as satisfiability. *Render (7.33) in pointwise notation and explain in your own words how it addresses the issue about contract* $q \xleftarrow{R} p$ *raised above. Moreover show that, for* R *simple, satisfiability (7.33) alone ensures contract* $q \xleftarrow{R} p$ *.*
□

7.5 RELATIONAL HOARE LOGIC

Recall Reynold's *relation on functions* (6.10):

$$f(R \leftarrow S)g \equiv f \cdot S \subseteq R \cdot g$$

$$\begin{tikzcd} A \arrow[d, "f"'] \arrow[dr, phantom, "\subseteq"] & B \arrow[l, "S"'] \arrow[d, "g"] \\ C & D \arrow[l, "R"] \end{tikzcd}$$

In a sense, this tells us that f and g behave in the same way within the particular *context* provided by the pair (S, R): for S-related inputs, the outputs are R-related:

$$a\ S\ b \Rightarrow (f\ a)\ R\ (g\ b)$$

This perspective of (6.10) has given rise to so-called *Relational Hoare logic* (RHL), a topic which is currently under much research [10].

The usual RHL notation for (6.10) is

$$f \sim g : S \Rightarrow R$$

As happens with functional contracts, this logic in general considers arbitrary programs instead of the functional f and g in (6.10). Then, for suitably typed relations P, Q (regarded as programs), we have:

$$P \sim Q : S \Rightarrow R \quad \equiv \quad P \cdot S \subseteq R \cdot Q \tag{7.34}$$

Let us see an example of RHL rule derivation, that arising when $P = p \to U, V$:

$$\begin{array}{cl} & (p \to U, V) \sim Q : S \Rightarrow R \\ \equiv & \quad \{\ (7.34)\ \} \\ & (p \to U, V) \cdot S \subseteq R \cdot Q \end{array}$$

$$
\begin{array}{ll}
\equiv & \quad \{ \text{ conditionals } \} \\
& [U,V] \cdot p\,? \cdot S \ \subseteq \ R \cdot Q \\
\equiv & \quad \{ \text{ (5.214) etc } \} \\
& (U \cdot \Phi_p \cup V \cdot \Phi_{\neg\, p}) \cdot S \ \subseteq \ R \cdot Q \\
\equiv & \quad \{ \text{ linearity (5.61) and } \cup \text{-universal (5.59) } \} \\
& \left\{ \begin{array}{l} U \cdot \Phi_p \cdot S \ \subseteq \ R \cdot Q \\ V \cdot \Phi_{\neg\, p} \cdot S \ \subseteq \ R \cdot Q \end{array} \right. \\
\equiv & \quad \{ \text{ (7.34) twice } \} \\
& \left\{ \begin{array}{l} U \sim Q : \Phi_p \cdot S \Rightarrow R \\ V \sim Q : \Phi_{\neg\, p} \cdot S \Rightarrow R \end{array} \right.
\end{array}
$$

In words: provided U (resp. V) behave similarly to Q in the strengthened input context $\Phi_p \cdot S$ (resp $\Phi_{\neg\, p} \cdot S$) then the conditional program $p \rightarrow U, V$ behaves similarly to Q in the wider input context S. Recall that $b\ (\Phi_p \cdot S)\ a$ means $b\ S\ a \wedge p\ b$.

Note that the rule is an equivalence

$$
(p \rightarrow U, V) \sim Q : S \Rightarrow R \qquad \equiv \qquad \left\{ \begin{array}{l} U \sim Q : \Phi_p \cdot S \Rightarrow R \\ V \sim Q : \Phi_{\neg\, p} \cdot S \Rightarrow R \end{array} \right.
$$

that is, the two clauses on the right are *weakest* pre-conditions. In case the constraints Φ_p and $\Phi_{\neg\, p}$ are dropped (as is usual) they become stronger and just sufficient conditions.

7.6 ABSTRACT INTERPRETATION

In practice, the proofs involved in verifying contracts may be hard to perform due to the intricacies of real-life sized software specifications, which may involve hundreds of invariants of arbitrary complexity. Such situations can only be tackled with the support of a theorem prover, and in many situations even this is not enough to accomplish the task. This problem has made software theorists to think of strategies helping designers to simplify their proofs. One such strategy is *abstract interpretation*.

It is often the case that the proof of a given contract does not require the whole model because the contract is only concerned with a particular *view* of the whole thing. As a very simple example, think of a model that is made of two independent parts $A \times B$ and of an invariant that constrains part A only. Then one may safely ignore B in the proofs. This is equivalent to applying projection $\pi_1 : A \times B \rightarrow A$ (2.21) to the original model. Note that π_1 is an *abstraction*, since it is a surjective function — recall figure 5.1.

In general, software models are not as "separable" as $A \times B$ is, but abstraction functions exist that yield much simpler models where proofs can be made easier. Different abstractions help in different proofs —

a kind of "on demand" *abstraction* making a model more *abstract* only with respect to the *specific* property one wishes to check.

In general, techniques of this kind are known as *abstract interpretation* techniques and play a major role in *program analysis*, for instance. To explain abstract interpretation we need to introduce the notion of a *relational type*.

RELATIONS AS TYPES A function h is said to have *relation type* $R \to S$, written $R \xrightarrow{h} S$ if

$$h \cdot R \subseteq S \cdot h \qquad \begin{tikzcd} B \arrow[d, "h"'] & B \arrow[l, "R"'] \arrow[d, "h"] \\ A & A \arrow[l, "S"] \end{tikzcd} \tag{7.35}$$

holds. Note that (7.35) could be written $h\ (S \leftarrow R)\ h$ in the notation of (6.10). In case $h : B \to A$ is surjective, i.e. h is an *abstraction function*, we also say that $A \xleftarrow{S} A$ is an *abstract simulation* of $B \xleftarrow{R} B$ through h.

A special case of relational type defines so-called *invariant functions*. A function of relation type $R \xrightarrow{h} id$ is said to be R-*invariant*, in the sense that

$$\langle \forall\ b, a\ :\ b\ R\ a :\ h\ b = h\ a \rangle \tag{7.36}$$

holds. When h is R-invariant, observations by h are not affected by R-transitions. In pointfree notation, an R-invariant function h is always such that:

$$R \subseteq \frac{h}{h} \tag{7.37}$$

For instance, a binary operation θ is *commutative* iff θ is swap-invariant, that is

$$\mathsf{swap} \subseteq \frac{\theta}{\theta} \tag{7.38}$$

holds.

Exercise *7.5. What does (7.35) mean in case R and S are partial orders?*
□

Exercise *7.6. Let t_0 be a real number. Show that*

$$(+t_0) \to id$$

is the relational type of all periodic functions (on $\mathbb{R}$) with period t_0.
□

***Exercise** 7.7. Show that relational types compose, that is* $Q \xleftarrow{k} S$ *and* $S \xleftarrow{h} R$ *entail* $Q \xleftarrow{k \cdot h} R$ *.*

□

***Exercise** 7.8. Sow that an alternative way of stating (7.31) is*

$$p \xrightarrow{R} q \quad \equiv \quad R \cdot \Phi_p \subseteq \Phi_q \cdot \top \tag{7.39}$$

□

***Exercise** 7.9. Recalling exercise 5.12, let the following relation specify that two dates are at least one week apart in time:*

$$d \; Ok \; d' \;\Leftrightarrow\; \mid d - d' \mid > 1 \; week$$

Looking at the type diagram below, say in your own words the meaning of the invariant specified by the relational type (7.35) statement below, on the left:

$$\mathsf{ker}\ (home \cup away) - id \xrightarrow{date} Ok$$

$$\begin{tikzcd}
G \arrow[r, "home \cup away"] \arrow[d, "date"'] & T \\
D & G \arrow[l, "date"] \arrow[u, "home \cup away"']
\end{tikzcd}$$

□

ABSTRACT INTERPRETATION Suppose that one wishes to show that $q : B \rightarrow \mathbb{B}$ is an invariant of some operation $B \xrightarrow{R} B$, i.e. that $q \xrightarrow{R} q$ holds and you know that $q = p \cdot h$, for some $h : B \rightarrow A$, as shown in the diagram. Then one can factor the proof in two steps:

$$\begin{tikzcd}
\mathbb{B} & A \arrow[l, "p"'] & A \arrow[l, "S"'] \\
 & B \arrow[u, "h"] \arrow[ul, bend left, "q"] & B \arrow[l, "R"'] \arrow[u, "h"]
\end{tikzcd}$$

- show that there is an abstract *simulation* S such that $R \xrightarrow{h} S$;
- prove $p \xrightarrow{S} p$, that is, that p is an (abstract) *invariant* of (abstract) S.

This strategy is captured by the following calculation:

$$R \cdot \Phi_q \subseteq \Phi_q \cdot \top$$

$$\begin{array}{ll}
\equiv & \quad \{\ q = p \cdot h\ \} \\
 & R \cdot \Phi_{(p \cdot h)} \ \subseteq\ \Phi_{(p \cdot h)} \cdot \top \\
\equiv & \quad \{\ (5.208) \text{ etc }\ \} \\
 & R \cdot \Phi_{(p \cdot h)} \ \subseteq\ h^\circ \cdot \Phi_p \cdot \top \\
\equiv & \quad \{\ \text{shunting}\ \} \\
 & h \cdot R \cdot \Phi_{(p \cdot h)} \ \subseteq\ \Phi_p \cdot \top \\
\Leftarrow & \quad \{\ R \xrightarrow{h} S\ \} \\
 & S \cdot h \cdot \Phi_{(p \cdot h)} \ \subseteq\ \Phi_p \cdot \top \\
\Leftarrow & \quad \{\ \Phi_{(p \cdot h)} \ \subseteq\ h^\circ \cdot \Phi_p \cdot h\ (5.212)\ \} \\
 & S \cdot h \cdot h^\circ \cdot \Phi_p \cdot h \ \subseteq\ \Phi_p \cdot \top \\
\Leftarrow & \quad \{\ \top = \top \cdot h \text{ (cancel } h\text{); img } h \ \subseteq\ id\ \} \\
 & S \cdot \Phi_p \ \subseteq\ \Phi_p \cdot \top
\end{array}$$

□

The following exercise gives a very simple example of application of abstract interpretation.

***Exercise** 7.10. A list of pairs $x \in (A \times A)^*$ can be represented simply by some $y \in A^*$ provided the length of y is even. Let*

$$\begin{array}{l}
\theta :: (A \times A) \to A^* \to A^* \\
\theta\ (a, b)\ y = a : b : y
\end{array}$$

be the operation that adds pairs to such a representation used in a safety-critical device. Show by abstract interpretation that $\theta\ (a, b)$ preserves the invariant

$$inv\ y = even\ (\mathsf{length}\ y) \tag{7.40}$$

by finding an abstract simulation ϕ of θ in the diagram:

$$\begin{tikzcd}
\mathbb{B} & \mathbb{N}_0 \arrow[l, "even"'] & \mathbb{N}_0 \arrow[l, "\phi"'] \\
 & A^* \arrow[u, "\mathsf{length}"] \arrow[ul, bend left, "inv"] & A^* \arrow[u, "\mathsf{length}"'] \arrow[l, "\theta\ (a{,}b)"]
\end{tikzcd}$$

□

Abstract interpretation techniques usually assume that h is an adjoint of a Galois connection. The examples below do not assume this, for an easy start.

7.7 SAFETY AND LIVENESS PROPERTIES

Before showing examples of abstract interpretation, let us be more specific about what was meant by "some operation $B \xrightarrow{R} B$ " above. In section 4.9 a monad was studied called the *state monad*. This monad is inhabited by state-transitions encoding state-based automata known as *Mealy machines*.

With relations one may be more relaxed on how to characterize state automata. In general, functional models generalize to so called *state-based* relational models in which there is

- a set Σ of *states*
- a subset $I \subseteq \Sigma$ of *initial* states
- a *step* relation $\Sigma \xrightarrow{R} \Sigma$ which expresses transition of states.

We define:

- $R^0 = id$ — no action or transition takes place
- $R^{i+1} = R \cdot R^i$ — all "paths" made of $i+1$ R-transitions
- $R^* = \bigcup_{i \geqslant 0} R^i$ — the set of all possible R-paths.

We represent the set I of initial states by the coreflexive $\Sigma \xrightarrow{\Phi_{(\in I)}} \Sigma$, simplified to $\Sigma \xrightarrow{I} \Sigma$ to avoid symbol cluttering.

Given $\Sigma \xrightarrow{R,I} \Sigma$ (i.e. a nondeterministic automaton, model) there are two kinds of property that one may wish to prove — *safety* and *liveness* properties. *Safety* properties are of the form $R^* \cdot I \subseteq S$, that is,

$$\langle \forall\, n \;:\; n \geqslant 0 :\; R^n \cdot I \subseteq S \rangle \tag{7.41}$$

for some safety relation $S : \Sigma \to \Sigma$, meaning: *All paths in the model originating from its initial states are* bounded *by* S. In the particular case $S = \frac{true}{p}$ [8]

$$\langle \forall\, n \;:\; n \geqslant 0 :\; R^n \cdot I \subseteq \frac{true}{p} \rangle \tag{7.42}$$

meaning that formula p holds for every state reachable by R from an initial state. Invariant preservation is an example of a safety property: if starting from a "good" state, the automaton only visits "good" (valid) states.

In contrast to safety properties, the so-called *liveness* properties are of the form

$$\langle \exists\, n \;:\; n \geqslant 0 :\; Q \subseteq R^n \cdot I \rangle \tag{7.43}$$

for some *target* relation $Q : \Sigma \to \Sigma$, meaning: *the target relation Q is eventually* realizable, *after n steps starting from an initial state.* In the particular case $Q = \frac{true}{p}$ we have

$$\langle \exists\, n \;:\; n \geqslant 0 :\; \frac{true}{p} \subseteq R^n \cdot I \rangle \tag{7.44}$$

meaning that, for a sufficiently large n, formula p will eventually hold.

[8] Recall that $\frac{true}{p} = \Phi_p \cdot \top$ (5.208).

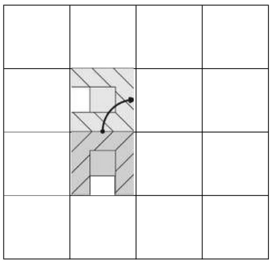
7.8 EXAMPLES

The Alcuin puzzle is an example of a problem that is characterized by a liveness and safety property:

- From initial state $where = \underline{Left}$, state $where = \underline{Right}$ is eventually reachable — a *liveness* property.
- Initial state $where = \underline{Left}$ is valid and no step of the automaton leads to invalid *where* states — a *safety* property.

The first difficulty in ensuring properties such as (7.42) e (7.44) is the quantification on the number of path steps. In the case of (7.44) one can try and find a particular path using a *model checker*. In both cases, the complexity /size of the *state space* may offer some impedance to proving / model checking. Below we show how to circumvent such difficulties by use of *abstract interpretation*.

THE HEAVY ARMCHAIR PROBLEM Let us show a simple, but effective example of abstract interpretation applied to a well-known problem — the *heavy armchair* problem.[9] Consider the following picture:



We wish to move the armchair to an adjacent square, horizontally or vertically. However, because the armchair is too heavy, it can only be rotated over one of its four legs, as shown in the picture.

The standard model for this problem is a pair (p, o) where $p = (y, x)$ captures the square where the armchair is positioned and o is one of the complex numbers $\{i, -i, 1, -1\}$ indicating the orientation of the armchair (that is, it can face N,S,E,W). Let the following step-relation be proposed,

$$R = P \times Q$$

where P captures the *adjacency* of two squares and Q captures 90° rotations. A *rotation* multiplies an orientation o by $\pm\ i$, depending on choosing a clockwise ($-i$) or anti-clockwise (i) rotation. Altogether:

$$((y',x'),d')\ R\ ((y,x),d)\ \Leftrightarrow \\ \left\{ \begin{array}{l} y' = y\ \pm\ 1 \wedge x' = x \vee y' = y \wedge x' = x\ \pm\ 1 \\ d' = (\pm\ i)\ d \end{array} \right.$$

[9] Credits: this version of the problem and the pictures shown are taken from [7].

We want to check the *liveness* property:

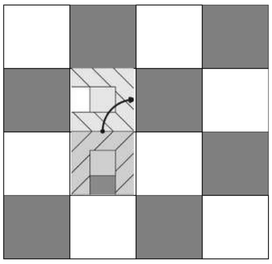
$$\textit{For some } n,\ ((y,x+1),d)\ R^n\ ((y,x),d)\ \textit{holds.} \tag{7.45}$$

That is, we wish to move the armchair to the adjacent square on its right, keeping the armchair's orientation. This is exactly what the pointfree version of (7.45) tells:

$$\langle \exists\, n\ ::\ (id \times (1+)) \times id\ \subseteq\ R^n \rangle$$

In other words: *there is a path with n steps that realizes the* function $move = (id \times (1+)) \times id$.

Note that the state of this problem is arbitrarily large. (The squared area is unbounded.) Moreover, the specification of the problem is nondeterministic. (For each state, there are four possible successor states.) We resort to *abstract interpretation* to obtain a bounded, deterministic (*functional*) model: the floor is coloured as a chess board and the armchair behaviour is abstracted by function $h = col \times dir$ which tells the *colour* of the square where the armchair is and the *direction* of its current orientation:



Since there are two colours (black, white) and two directions (horizontal, vertical), both can be modelled by Booleans. Then the action of moving to any adjacent square abstracts to *color* negation and any 90° rotation abstracts to *direction* negation:

$$P \xrightarrow{col} (\neg) \tag{7.46}$$

$$Q \xrightarrow{dir} (\neg) \tag{7.47}$$

In detail:

$$col\ (y,x) = even\ (y+x)$$
$$dir\ x = x \in \{1,-1\}$$

For instance, $col\ (0,0) = \mathsf{True}$ (black in the picture), $col\ (1,1) = \mathsf{True}$, $col\ (1,2) = \mathsf{False}$ and so on; $dir\ 1 = \mathsf{True}$ (horizontal orientation), $dir\ (-i) = \mathsf{False}$, and so on. Checking (7.47):

$$\begin{array}{cl} & dir\ ((\pm\ i)\ x) \\ = & \quad \{\ dir\ x = x \in \{1,-1\}\ \} \\ & (\pm\ i)\ x \in \{1,-1\} \end{array}$$

$$
\begin{array}{ll}
= & \quad \{ \ \mathit{multiply\ by}\ (\pm\ i) \text{ within } \{1, i, -1, -i\} \ \} \\
 & x \in \{-i, i\} \\
= & \quad \{ \ \text{the remainder of } \{-i, i\} \text{ is } \{1, -1\} \ \} \\
 & \neg\ (x \in \{1, -1\}) \\
= & \quad \{ \ \mathit{dir}\ x = x \in \{1, -1\} \ \} \\
 & \neg\ (\mathit{dir}\ x) \\
\Box &
\end{array}
$$

Checking (7.46):

$$
\begin{array}{ll}
 & (\neg) \xleftarrow{\mathit{col}} P \\
\equiv & \quad \{ \ \text{(7.35) for functions} \ \} \\
 & \mathit{col} \cdot P \ \subseteq \ \neg \cdot \mathit{col} \\
\equiv & \quad \{ \ \text{shunting ; go pointwise} \ \} \\
 & (y', x')\ P\ (y, x) \Rightarrow \mathit{even}\ (y' + x') = \neg\ \mathit{even}\ (y + x) \\
\equiv & \quad \{ \ \text{unfold} \ \} \\
 & \left\{ \begin{array}{l} y' = y\ \pm\ 1 \wedge x' = x \Rightarrow \mathit{even}\ (y' + x') = \neg\ \mathit{even}\ (y + x) \\ y' = y \wedge x' = x\ \pm\ 1 \Rightarrow \mathit{even}\ (y' + x') = \neg\ \mathit{even}\ (y + x) \end{array} \right. \\
\equiv & \quad \{ \ \text{substitutions ; trivia} \ \} \\
 & \left\{ \begin{array}{l} \mathit{even}\ (y\ \pm\ 1) = \neg\ \mathit{even}\ y \\ \mathit{even}\ (x\ \pm\ 1) = \neg\ \mathit{even}\ x \end{array} \right. \\
\equiv & \quad \{ \ \text{trivia} \ \} \\
 & \mathit{true} \\
\Box &
\end{array}
$$

Altogether:

$$R \xrightarrow{\mathit{col} \times \mathit{dir}} (\neg \times \neg)$$

That is, step *relation* R is simulated by $s = \neg \times \neg$, i.e. the *function*

$$s\ (c, d) = (\neg\ c, \neg\ d)$$

over a state space with 4 possibilities only: wherever the armchair turns over one of its legs, whatever this is, it changes *both* the colour of the square where it is, and its direction.

At this level, we note that *observation* function

$$f\ (c, d) = c\ \oplus\ d \tag{7.48}$$

is *s-invariant* (7.36), that is

$$f \cdot s = f \tag{7.49}$$

since $\neg\ c\ \oplus\ \neg\ d = c\ \oplus\ d$ holds. By induction on $n, f \cdot s^n = f$ holds too.

Expressed under this abstraction, (7.45) is rephrased into: *there is a number of steps n such that* $s^n\ (c,d) = (\neg\ c,d)$ *holds*. Let us check this abstract version of the original property, assuming variable n existentially quantified:

$$\begin{array}{ll}
 & s^n\ (c,d) = (\neg\ c,d) \\
\Rightarrow & \qquad \{\ \text{Leibniz}\ \} \\
 & f\ (s^n\ (c,d)) = f\ (\neg\ c,d) \\
\equiv & \qquad \{\ f \text{ is } \textit{s-invariant}\ \} \\
 & f\ (c,d) = f\ (\neg\ c,d) \\
\equiv & \qquad \{\ (7.48)\ \} \\
 & c\ \oplus\ d = \neg\ c\ \oplus\ d \\
\equiv & \qquad \{\ 1\ \oplus\ d = \neg\ d \text{ and } 0\ \oplus\ d = d\ \} \\
 & d = \neg\ d \\
\equiv & \qquad \{\ \text{trivia}\ \} \\
 & \mathit{false}
\end{array}$$

Thus, for all paths of arbitrary length n, $s^n\ (c,d) \neq (\neg\ c,d)$. We conclude that the proposed liveness property does not at all hold!

ALCUIN PUZZLE EXAMPLE Abstract interpretation applies nicely to this problem, thanks to its symmetries. On the one hand, one does not need to work over the 16 functions in $\mathit{Bank}^{\mathit{Being}}$, since starting from the left margin or from the right margin is irrelevant. Another symmetry can be found in type *Being*, suggesting the following abstraction of beings into three classes:

$$f : \mathit{Being} \to \{\alpha, \beta, \gamma\}$$

$$f = \left(\begin{tikzcd}
\mathit{Goose} \arrow[r] & \alpha \\
\mathit{Fox} \arrow[r] & \beta \\
\mathit{Beans} \arrow[ru] & \\
\mathit{Farmer} \arrow[r] & \gamma
\end{tikzcd}\right)$$

The abstraction consists in unifying the maximum and minimum elements of the "food chain". In fact, the simultanous presence of one α and one β is enough for defining the invariant — which *specific* being eats the other is irrelevant detail. This double abstraction is captured by

$$\begin{tikzcd}
\mathit{Bank} & \mathit{Being} \arrow[l, "w"'] \arrow[d, "f"] \\
1 \arrow[u, "\underline{\mathit{Left}}"] & \{\alpha, \beta, \gamma\} \arrow[l, "V"]
\end{tikzcd} \qquad V = \underline{\mathit{Left}}^{\circ} \cdot w \cdot f^{\circ}$$

where the choice of $Left$ as reference bank is arbitrary. Thus function w is abstracted by the row *vector* relation V [10] such that:

$$\_\; V\ x = \langle \exists\, b \;:\; x = f\ b :\; w\ b = Left \rangle$$

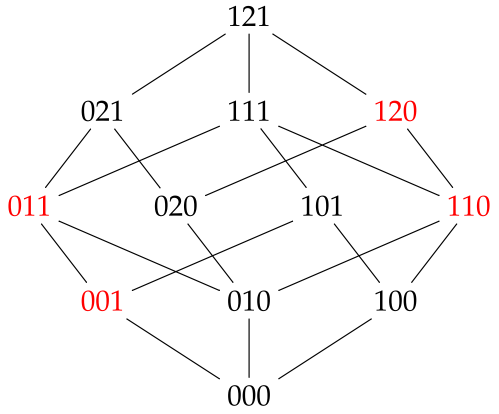
Vector V tells whether at least one being of class x can be found in the reference bank. Noting that there could be more than one β there, we refine the abstraction a bit so that the number of beings of each class is counted.[11] This leads to the following *state-abstraction* (higher order) function h based on f:

$$\begin{array}{l} h : (Being \to Bank) \to \{\alpha, \beta, \gamma\} \to \{0,1,2\} \\ h\ w\ x = \langle \sum b \;:\; x = f\ b \wedge w\ b = Left :\; 1 \rangle \end{array}$$

For instance,

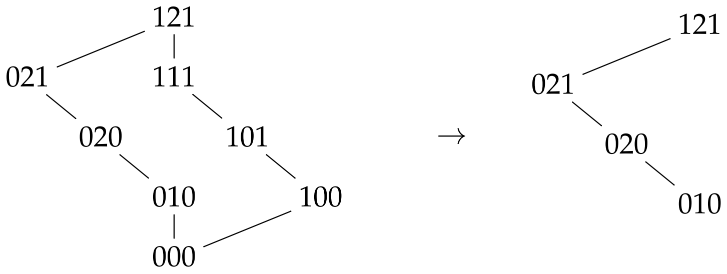
$$\begin{array}{rcl} h\ \underline{Left} & = & 121 \\ h\ \underline{Right} & = & 000 \end{array}$$

abbreviating by vector xyz the mapping $\{\alpha \mapsto x, \beta \mapsto y, \gamma \mapsto z\}$.[12] To obtain the other bank just compute: $\overline{x} = 121 - x$. Note that there are $2 \times 3 \times 2 = 12$ possible state vectors, 4 of which are invalid (these are marked in red):



The ordering implicit in the lattice above is pointwise $(\leqslant)$. This is complemented by $\overline{x} = 121 - x$, which gives the information of the other bank.

The 8 valid states can be further abstracted to only 4 of them,

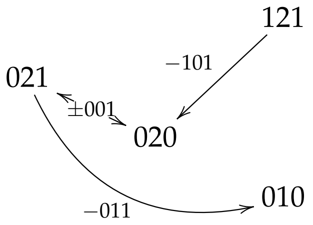


[10] A fragment of $! : \{\alpha, \beta, \gamma\} \to 1$, recall section 5.5.

[11] This suggests that linear algebra would be a good alternative to relation algebra here!

[12] This version of the model is inspired in [7].

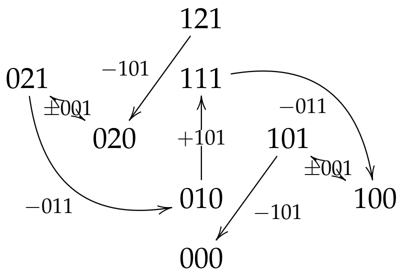
since, due to complementation (cf. the Left-Right margin symmetry), we only need to reach state 010. Then we reverse the path through the complements. In this setting, the automaton is deterministic, captured by the abstract automaton:



Termination is ensured by disabling toggling between states 021 and 020:

$$\begin{array}{r} 121\\ -101\\ \hline 020\\ +001\\ \hline 021\\ -011\\ \hline 010 \end{array}$$

We then take the complemented path 111 → 100 → 101 → 000. So the abstract solution for the Alcuin puzzle is, finally:



$$\begin{array}{r} 121\\ -101\\ \hline 020\\ +001\\ \hline 021\\ -011\\ \hline 010\\ +101\\ \hline 111\\ -011\\ \hline 100\\ +001\\ \hline 101\\ -101\\ \hline 000 \end{array}$$

At this point note that, according to the principles of abstract interpretation stated above, quite a few steps are pending in this exercise: abstract the *starving* invariant to the vector level, find an abstract simulation of *carry*, and so on and so forth. But — why bother doing all that? There is no other operation in the problem, so the abstraction found is, in a sense, universal: we should have started from the vector model and not from the *Being* → *Bank* model, which is not *sufficiently* abstract.

The current scientific basis of programming enables the calculation of programs, following the scientific method. So, programming is lesser and lesser an *art*. Where is creativity gone to? To the *art* of abstract modelling and elegant proving — this is where it can be found nowadays.

***Exercise** 7.11. Verification of code involves calculations of real numbers and is often done on the basis of an abstract interpretation called sign analysis:*

$$\begin{array}{l} sign : \mathbb{R} \rightarrow \{-,0,+\} \\ sign\ 0 = 0 \\ sign\ x = \mathbf{if}\ x > 0\ \mathbf{then}\ +\ \mathbf{else}\ - \end{array}$$

Suppose there is evidence that the operation $\theta : \{-,0,+\}^2 \rightarrow \{-,0,+\}$ *defined by*

$$\begin{array}{c|ccc} \theta & - & 0 & + \\ \hline - & + & 0 & - \\ 0 & 0 & 0 & 0 \\ + & - & 0 & + \end{array} \tag{7.50}$$

is the abstract simulation induced by sign of a given concrete operation $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$*, that is, that*

$$\theta \cdot (sign \times sign) = sign \cdot f \tag{7.51}$$

holds. It is easy to see, by inspection of (7.50), that θ is a commutative operation, recalling (7.38).

- *Show that* $sign \cdot f$ *is necessarily commutative as well. (Hint: the free theorem of* swap *can be useful here.)*
- *Does the previous question guarantee that the specific operation f is also commutative? Answer informally.*

□

7.9 "FREE CONTRACTS"

In design by contract, many functional *contracts* arise naturally as corollaries of *free theorems*. This has the advantage of saving us from proving such contracts explicitly.

The following exercises provide ample evidence of this.

***Exercise** 7.12. Confirm that (7.9) is the free theorem of functional composition* $(\cdot)$*, which has type*

$$(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$

leading to contract composition *(7.22) as shown earlier on.*

□

***Exercise** 7.13. Show that contract* $q^\star \xleftarrow{\mathsf{map}\, f} p^\star$ *holds provided contract* $q \xleftarrow{f} p$ *holds.*

□

***Exercise** 7.14. Suppose a functional programmer wishes to prove the following property of lists:*

$$\langle \forall\, a,s \;:\; (p\, a) \wedge \langle \forall\, a' \;:\; a' \in \mathsf{elems}\, s \;:\; p\, a' \rangle \;:\; \langle \forall\, a'' \;:\; a'' \in \mathsf{elems}\, (a:s) \;:\; p\, a'' \rangle \rangle$$

Show that this property is a contract arising (for free) from the polymorphic type of the cons *operation* $(:)$ *on lists.*

□

7.10 REASONING BY APPROXIMATION

Abstraction interpretation situations $S \xleftarrow{h} R$

$$\begin{tikzcd}
C \arrow[d, "h"'] & C \arrow[l, "R"'] \arrow[d, "h"] \arrow[dl, phantom, "\subseteq"] \\
A & A \arrow[l, "S"]
\end{tikzcd}$$

include the particular case

$$\begin{tikzcd}
C \arrow[d, "\alpha"'] & C \arrow[l, "f"'] \arrow[d, "\alpha"] \arrow[dl, phantom, "\subseteq"] \\
A & A \arrow[l, "(\leqslant)\cdot g"]
\end{tikzcd}$$

where α is left adjoint of a Galois connection

$$\begin{tikzcd}
(\leqslant) \arrow[r, bend left, "\gamma"] \arrow[r, phantom, "\top"] & (\sqsubseteq) \arrow[l, bend left, "\alpha"]
\end{tikzcd}$$

that is,

$$\alpha^\circ \cdot (\leqslant) = (\sqsubseteq) \cdot \gamma$$

holds (5.134). Given concrete function f, we seek for functional abstract *simulations* g as solutions to the equation

$$\alpha \cdot f \ \dot{\leqslant}\ g \cdot \alpha \tag{7.52}$$

that is, $\langle \forall\ x\ :\ x \in C :\ \alpha\ (f\ x) \leqslant g\ (\alpha\ x) \rangle$. In words: abstract g is a "good" simulation of concrete f with respect to the $(\sqsubseteq)$ ordering. Note that (7.52) is equivalent to

$$f \ \dot{\sqsubseteq}\ \gamma \cdot g \cdot \alpha \tag{7.53}$$

So, the performance of concrete f is at most as good as that of simulation g at abstract level.

Let us try and solve the equation for unknown g:

$$\begin{array}{ll}
& \alpha \cdot f \ \dot{\leqslant}\ g \cdot \alpha \\
\equiv & \quad \{\ (5.93);\ \text{shunting } \alpha \text{ to the left}\ \} \\
& \alpha \cdot f \cdot \alpha^{\circ} \ \subseteq\ (\leqslant) \cdot g \qquad (7.54) \\
\Rightarrow & \quad \{\ \text{monotonicity of composition}\ \} \\
& \alpha \cdot f \cdot \alpha^{\circ} \cdot (\leqslant) \ \subseteq\ (\leqslant) \cdot g \cdot (\leqslant) \\
\Rightarrow & \quad \{\ g \text{ monotone: } g \cdot (\leqslant)\ \subseteq\ (\leqslant) \cdot g;\ (\leqslant) \text{ transitive}\ \} \\
& \alpha \cdot f \cdot \alpha^{\circ} \cdot (\leqslant) \ \subseteq\ (\leqslant) \cdot g \qquad (7.55) \\
\equiv & \quad \{\ \text{Galois connection (5.134)}\ \} \\
& \alpha \cdot f \cdot \gamma \cdot (\sqsubseteq) \ \subseteq\ (\leqslant) \cdot g \qquad (7.56) \\
\equiv & \quad \{\ \text{back to (7.55)}\ \} \\
& \alpha \cdot f \cdot \alpha^{\circ} \cdot (\leqslant) \ \subseteq\ (\leqslant) \cdot g \\
\Rightarrow & \quad \{\ id\ \subseteq\ (\leqslant)\ \} \\
& \alpha \cdot f \cdot \alpha^{\circ} \ \subseteq\ (\leqslant) \cdot g
\end{array}$$

Note the circular implication. So, every step in the reasoning is equivalent to the equation we started from (7.52). In particular, step (7.55) is equivalent to

$$g\ a = \langle \bigvee\ c\ :\ \alpha\ c \leqslant a :\ (\alpha \cdot f)\ c \rangle$$

by (6.22) in case suprema exist (complete lattices). So, in this situation — existence of universal suprema — (7.52) has one solution only, which we denote by $f^{\sharp}$:

$$\alpha \cdot f \ \dot{\leqslant}\ g \cdot \alpha \quad \equiv \quad g\ a = \underbrace{\langle \bigvee\ c\ :\ \alpha\ c \sqsubseteq a :\ (\alpha \cdot f)\ c \rangle}_{f^{\sharp}\ a}$$

Step (7.56) implies $\alpha \cdot f \cdot \gamma \ \dot{\leqslant}\ g$ — by $id\ \subseteq\ (\sqsubseteq)$ —, which provides a hint to the pointfree definition of such a solution:

$$\begin{array}{l}
f^{\sharp} : A \rightarrow A \\
f^{\sharp} = \alpha \cdot f \cdot \gamma
\end{array}$$

Indeed, $f^\sharp = \alpha \cdot f \cdot \gamma$ solves the equation:

$$\begin{array}{ll}
 & f \mathrel{\dot{\sqsubseteq}} \gamma \cdot f^\sharp \cdot \alpha \\
\equiv & \qquad \{ \; f^\sharp = \alpha \cdot f \cdot \gamma \; \} \\
 & f \mathrel{\dot{\sqsubseteq}} \gamma \cdot (\alpha \cdot f \cdot \gamma) \cdot \alpha \\
\Leftarrow & \qquad \{ \; id \mathrel{\dot{\sqsubseteq}} \gamma \cdot \alpha \; \} \\
 & f \mathrel{\dot{\sqsubseteq}} f \cdot \gamma \cdot \alpha \\
\Leftarrow & \qquad \{ \; f \text{ monotone: } f \cdot (\sqsubseteq) \;\subseteq\; (\sqsubseteq) \cdot f \text{; monotonicity of composition } \} \\
 & id \mathrel{\dot{\sqsubseteq}} \gamma \cdot \alpha \\
\equiv & \qquad \{ \; id \mathrel{\dot{\sqsubseteq}} \gamma \cdot \alpha \; \} \\
 & true \\
\Box &
\end{array}$$

Summing up: for α the lower adoint of a Galois connection involving complete lattices,

$$\alpha \cdot f \mathrel{\dot{\leqslant}} g \cdot \alpha \quad \equiv \quad g = f^\sharp$$

holds where

$$\begin{array}{lcl}
f^\sharp & = & \alpha \cdot f \cdot \gamma \\
 & = & \langle \bigvee \; c \; : \; \alpha \; c \sqsubseteq a : \; (\alpha \cdot f) \; c \rangle
\end{array}$$

Example: suppose that, for some reason, we decide to abstact finite non-empty lists (of suitably ordered data) by their suprema and infima,

$$\alpha = \langle minimum, maximum \rangle$$

and represent intervals in the expected way:

$$\gamma \; (a, b) = [a \mathinner{..} b]$$

Consider the concrete operation $(a{:})$. Then, for instance,

$$\begin{array}{ll}
 & (1{:})^\sharp \; (2, 4) \\
= & \qquad \{ \; f^\sharp = \alpha \cdot f \cdot \gamma \; \} \\
 & \alpha \; (1 : [2, 3, 4]) \\
= & \qquad \{ \; \alpha = \langle minimum, maximum \rangle \; \} \\
 & (1, 4)
\end{array}$$

That is:

$$(a{:})^\sharp \; (x, y) = (\mathbf{if} \; a < x \; \mathbf{then} \; a \; \mathbf{else} \; x, \mathbf{if} \; a > y \; \mathbf{then} \; a \; \mathbf{else} \; y)$$

The underlying GC is

$$\alpha\ x \leqslant (a,b) \ \Leftrightarrow\ x \sqsubseteq \gamma\ (a,b)$$

where $(\leqslant)$ is interval containment and $(\sqsubseteq) = \mathsf{elems}^{\circ} \cdot (\subseteq) \cdot \mathsf{elems}$. Abstraction interpretation here means that repetition and position of specific values in the list are irrelevant for the analysis under way: only the range of values matters.

7.11 BIBLIOGRAPHY NOTES

To be completed

8

PROGRAMMING FROM RELATIONAL HYLOMORPHISMS

This chapter relates to chapter 3 in the same way as chapter 5 relates to chapter 2. In essence, we wish to know how to address inductive (recursive) relations. It turns out that the concept of (functional) catamorphism extends relation-wise in a very smooth way, starting from (8.2) below. But the richer algebra of relations enables us with a far wider treatment of recursion. In particular, we shall show how recursive programs arise as special cases of inductive relations.

Take the relation S in exercise 5.17 as the *specification* of *sorting*. As the function *bag* that yields the multiset of elements of a finite list can be expressed as a catamorphism,

$$\begin{array}{l} bag\ [\,] = \underline{0} \\ bag\ (a : x) = bag\ x \oplus a \end{array}$$

for $(f \oplus a)\ x = f\ x + (\mathbf{if}\ x = a\ \mathbf{then}\ 1\ \mathbf{else}\ 0)$, then it comes to mind that, somehow,

$$Perm = \frac{bag}{bag} \tag{8.1}$$

— the *permutes* equivalence relation among finite lists — should be expressible inductively over finite lists too. We show this to be true and a special case of a much wider setting in programming from (formal) specifications.

8.1 RELATIONAL CATAMORPHISMS

Recall from section 6.3 the notion of a *relator* F, that is, a mathematical construction such that, for any type A, type $\mathsf{F}\ A$ is defined and for any relation $R : B \leftarrow A$, relation $\mathsf{F}\ R : \mathsf{F}\ B \leftarrow \mathsf{F}\ A$ is defined such that $\mathsf{F}\ id = id$, $\mathsf{F}\ R^\circ = (\mathsf{F}\ R)^\circ$ and $\mathsf{F}\ (R \cdot S) = (\mathsf{F}\ R) \cdot (\mathsf{F}\ S)$.

Any relation $R : A \leftarrow \mathsf{F}\ A$ is said to be a (relational) F-*algebra*. Special cases include functional F-algebras and, among these, those that are isomorphisms. Within these, the so-called *initial* F-algebras, say $\mathsf{in} : \mathsf{T} \leftarrow \mathsf{F}\ \mathsf{T}$, are such that, given any other F-algebra $R : A \leftarrow \mathsf{F}\ A$, there is a unique relation of type $A \leftarrow \mathsf{T}$, usually written $(\!|R|\!)$, such that $(\!|R|\!) \cdot \mathsf{in} = R \cdot \mathsf{F}\ (\!|R|\!)$ holds.

Type T (often denoted by μ_{F} to express its relationship with the base relator F) is also referred to as *initial*. The meaning of such relations $(\!| R |\!)$, usually referred to as *catamorphisms*, or *folds*, is captured by the *universal property*:[1]

$$X = (\!| R |\!) \quad \equiv \quad X \cdot \mathsf{in} = R \cdot (\mathsf{F}\ X) \tag{8.2}$$

Functor (relator) F captures the recursive pattern of type T. For instance, for $\mathsf{T} = \mathbb{N}_0$ one has

$$\left\{ \begin{array}{l} \mathsf{F}\ X = 1 + X \\ \mathsf{F}\ f = id + f \end{array} \right. \tag{8.3}$$

as we have seen in previous chapters.

Let us see an example of relational catamorphism,

$$(\geqslant) = (\!| [\top, \mathsf{succ}] |\!) \tag{8.4}$$

the $\mathbb{N}_0 \xleftarrow{\geqslant} \mathbb{N}_0$ relation on the natural numbers:

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[r, bend left, "\mathsf{in}^\circ"] \arrow[r, phantom, "\cong"] \arrow[d, "(\geqslant)"'] & 1 + \mathbb{N}_0 \arrow[l, bend left, "\mathsf{in}"] \arrow[d, "id+(\geqslant)"] \\
\mathbb{N}_0 & 1 + \mathbb{N}_0 \arrow[l, "{[\top, \mathsf{succ}]}"]
\end{tikzcd}$$

$$(\geqslant) \cdot \mathsf{in} = [\top, \mathsf{succ}] \cdot (id + (\geqslant))$$

Note how $(\geqslant)$ compares with $\mathsf{in} = [\mathsf{zero}, \mathsf{succ}]$ — the base case zero expands as much as possible to the largest relation $\top$ of its type ($\mathbb{N}_0 \leftarrow 1$).

Let us see what comes out of this catamorphism once rendered into pointwise notation:

$$\begin{array}{cl}
& (\geqslant) \cdot \mathsf{in} = [\top, \mathsf{succ}] \cdot (id + (\geqslant)) \\
\equiv & \qquad \{\ \mathsf{in} = [\mathsf{zero}, \mathsf{succ}];\ \text{coproducts}\ \} \\
& [(\geqslant) \cdot \mathsf{zero}, (\geqslant) \cdot \mathsf{succ}] = [\top, \mathsf{succ} \cdot (\geqslant)] \\
\equiv & \qquad \{\ \text{coproducts again}\ \} \\
& \left\{ \begin{array}{l} (\geqslant) \cdot \mathsf{zero} = \top \\ (\geqslant) \cdot \mathsf{succ} = \mathsf{succ} \cdot (\geqslant) \end{array} \right. \\
\equiv & \qquad \{\ \text{go pointwise}\ \} \\
& \left\{ \begin{array}{l} y \geqslant 0 \ \Leftrightarrow\ \mathsf{True} \\ y \geqslant x + 1 \ \Leftrightarrow\ y\ (\mathsf{succ} \cdot (\geqslant))\ x \end{array} \right. \\
\equiv & \qquad \{\ \text{go pointwise}\ \} \\
& \left\{ \begin{array}{l} y \geqslant 0 \ \Leftrightarrow\ \mathsf{True} \\ y \geqslant x + 1 \ \Leftrightarrow\ \langle \exists\, z\ :\ y = z + 1 :\ z \geqslant x \rangle \end{array} \right.
\end{array}$$

[1] Relational catamorphisms can be shown to arise from functional ones via the often called *Eilenberg-Wright Lemma* [12], itself a consequence of the adjoint catamorphism construction (4.74) via the powerset monad which stems from adjunction (5.251).

We get $(\geqslant)$ defined inductively over the natural numbers.

The cancellation law

$$(\!|R|\!) = R \cdot \mathsf{F}\ (\!|R|\!) \cdot \mathsf{in}^\circ \tag{8.5}$$

arises immediately from (8.2) and the fact that the initial algebra $\mu_\mathsf{F} \xleftarrow{\mathsf{in}} \mathsf{F}\ \mu_\mathsf{F}$ is an isomorphism. By indirect equality over (8.5) we get:

$$\begin{array}{ll}
& X \subseteq (\!|R|\!) \\
\equiv & \quad \{\ \text{cancellation}\ \} \\
& X \subseteq R \cdot \mathsf{F}\ (\!|R|\!) \cdot \mathsf{in}^\circ \\
\Leftarrow & \quad \{\ \text{since } X \subseteq (\!|R|\!)\text{; monotonicity of } \mathsf{F}\ \} \\
& X \subseteq R \cdot (\mathsf{F}\ X) \cdot \mathsf{in}^\circ
\end{array}$$

That is:

$$X \subseteq (\!|R|\!) \quad \Leftarrow \quad X \subseteq R \cdot (\mathsf{F}\ X) \cdot \mathsf{in}^\circ \tag{8.6}$$

Similarly:

$$(\!|R|\!) \subseteq X \quad \Leftarrow \quad R \cdot \mathsf{F}\ X \cdot \mathsf{in}^\circ \subseteq X \tag{8.7}$$

Note that the cancellation law (8.5) expresses $(\!|R|\!)$ as solution (fixpoint) to the equation

$$X = R \cdot \mathsf{F}\ X \cdot \mathsf{in}^\circ.$$

Law (8.6) — resp. (8.7) — tells that it is the greatest post-fixed — resp. least pre-fixpoint — of such equation. Thus it is its unique solution.[2]

As expected, the relational catamorphism combinator is monotonic:

$$(\!|R|\!) \subseteq (\!|S|\!) \quad \Leftarrow \quad R \subseteq S \tag{8.8}$$

This follows almost immediately from the above:

$$\begin{array}{ll}
& (\!|R|\!) \subseteq (\!|S|\!) \\
\Leftarrow & \quad \{\ (8.7)\ ;\ \text{isomorphism } \mathsf{in}\ \} \\
& R \cdot \mathsf{F}\ (\!|S|\!) \subseteq (\!|S|\!) \cdot \mathsf{in} \\
\equiv & \quad \{\ \text{cancellation}\ \} \\
& R \cdot \mathsf{F}\ (\!|S|\!) \subseteq S \cdot \mathsf{F}\ (\!|S|\!) \\
\Leftarrow & \quad \{\ \text{monotonicity of composition}\ \} \\
& R \subseteq S
\end{array}$$

□

By reflection $(\!|\mathsf{in}|\!) = id$ and successively making $S := \mathsf{in}$, then $R := \mathsf{in}$ in (8.8) we get sufficient conditions for catamorphisms being coreflexive and reflexive:

$$(\!|R|\!) \subseteq id \quad \Leftarrow \quad R \subseteq \mathsf{in} \tag{8.9}$$

$$id \subseteq (\!|S|\!) \quad \Leftarrow \quad \mathsf{in} \subseteq S \tag{8.10}$$

[2] Cf. the Knaster-Tarski fixpoint theorem.

Thus $(\geqslant)$ is reflexive.

Can we calculate the converse $(\leqslant) = (\geqslant)^\circ$ from its defining catamorphism? We reason (”ping”):

$$
\begin{array}{cl}
 & (\!|R|\!) \subseteq (\geqslant)^\circ \\
\Leftarrow & \quad \{ \text{ (8.7) } \} \\
 & R \cdot \mathsf{F}\,(\geqslant)^\circ \cdot \mathsf{in}^\circ \subseteq (\geqslant)^\circ \\
\equiv & \quad \{ \text{ take converses } \} \\
 & \mathsf{in} \cdot \mathsf{F}\,(\geqslant) \cdot R^\circ \subseteq (\geqslant) \\
\equiv & \quad \{ \text{ let } R = [Q, S] \ \} \\
 & [\mathsf{zero}, \mathsf{succ} \cdot (\geqslant)] \cdot [Q, S]^\circ \subseteq (\geqslant) \\
\equiv & \quad \{ \text{ (5.121) ; (5.59) } \} \\
 & \left\{ \begin{array}{l} \mathsf{zero} \cdot Q^\circ \subseteq (\geqslant) \\ \mathsf{succ} \cdot (\geqslant) \cdot S^\circ \subseteq (\geqslant) \end{array} \right. \\
\Box &
\end{array}
$$

Since we are constrained to using the components zero or succ of the Peano algebra, we are led to $Q, S :=$ zero, zero $\cup$ succ. Note that $S :=$ succ wouldn’t work because $(\leqslant)$ should be larger than the identity. Summing up:

$$
(\!|\,[\mathsf{zero}, \mathsf{zero} \cup \mathsf{succ}]\,|\!) \subseteq (\leqslant) \tag{8.11}
$$

Now the “pong”:

$$
\begin{array}{cl}
 & (\leqslant) \subseteq (\!|\,[\mathsf{zero}, \mathsf{zero} \cup \mathsf{succ}]\,|\!) \\
\Leftarrow & \quad \{ \text{ (8.6) ; converses } \} \\
 & (\geqslant) \subseteq \mathsf{in} \cdot \mathsf{F}\,(\geqslant) \cdot [\mathsf{zero}, \mathsf{zero} \cup \mathsf{succ}]^\circ \\
\equiv & \quad \{ \text{ catamorphism } (\geqslant) \ \} \\
 & [\top, \mathsf{succ}] \cdot \mathsf{F}\,(\geqslant) \cdot \mathsf{in}^\circ \subseteq \mathsf{in} \cdot \mathsf{F}\,(\geqslant) \cdot [\mathsf{zero}, \mathsf{zero} \cup \mathsf{succ}]^\circ \\
\equiv & \quad \{ \ \mathsf{F}\,(\geqslant) = id + (\geqslant) \ \} \\
 & \top \cdot \mathsf{zero}^\circ \cup \mathsf{succ} \cdot (\geqslant) \cdot \mathsf{succ}^\circ \subseteq \mathsf{zero} \cdot \mathsf{zero}^\circ \cup \mathsf{succ} \cdot (\geqslant) \cdot \mathsf{zero}^\circ \cup \mathsf{succ} \cdot (\geqslant) \cdot \mathsf{succ}^\circ \\
\equiv & \quad \{ \text{ handle } \mathsf{succ} \cdot (\geqslant) \cdot \mathsf{succ}^\circ) \ \} \\
 & \mathsf{zero}^\circ \subseteq \mathsf{zero} \cdot \mathsf{zero}^\circ \cup \mathsf{succ} \cdot (\geqslant) \cdot \mathsf{zero}^\circ \cup \mathsf{succ} \cdot (\geqslant) \cdot \mathsf{succ}^\circ \\
\Leftarrow & \quad \{ \ (\geqslant) \text{ necessarily reflexive } \} \\
 & \mathsf{zero}^\circ \subseteq \mathsf{zero} \cdot \mathsf{zero}^\circ \cup \mathsf{succ} \cdot \mathsf{zero}^\circ \cup \mathsf{succ} \cdot \mathsf{succ}^\circ \\
\equiv & \quad \{ \text{ shunting } \mathsf{zero}^\circ\text{; zero is difunctional; kernel of constant function } \} \\
 & id \subseteq \mathsf{zero} \cup \mathsf{succ} \cdot \top \cup \mathsf{succ} \cdot \mathsf{succ}^\circ \cdot \mathsf{zero} \\
\equiv & \quad \{ \text{ zero is difunctional; in is injective, so } \mathsf{succ}^\circ \cdot \mathsf{zero} = \bot \ \} \\
 & id \subseteq \mathsf{zero} \cup \mathsf{succ} \cdot \top \\
\equiv & \quad \{ \ \mathsf{zero} = \mathsf{zero} \cdot \top;\ \top^\circ = \top;\ \text{(5.121)} \ \}
\end{array}
$$

$$
\begin{array}{ll}
& \mathsf{in}^\circ \ \subseteq \ [\top, \top]^\circ \\
\equiv & \qquad \{ \ [\top,\top] = \top \ \} \\
& \mathsf{in} \ \subseteq \ \top \\
\Box &
\end{array}
$$

So, $(\leqslant)$ is the catamorphism $(\!|\, [\mathsf{zero}, \mathsf{zero} \cup \mathsf{succ}] \,|\!)$.

CATA-FUSION Further to cancellation, which can also be written

$$(\!| R |\!) \cdot \mathsf{in} \ = \ R \cdot \mathsf{F}\, (\!| R |\!) \tag{8.12}$$

another property stems from (8.2) that proves particularly useful in calculations about $(\!| R |\!)$:

$$S \cdot (\!| R |\!) = (\!| Q |\!) \quad \Leftarrow \quad S \cdot R = Q \cdot \mathsf{F}\, S \tag{8.13}$$

This is known as the *fusion* property. Fusion is particularly helpful in the sense of finding a sufficient condition on S, R and Q for merging $S \cdot (\!| R |\!)$ into $(\!| Q |\!)$. In the words of [12], law (8.13) *is probably the most useful tool in the arsenal of techniques for program derivation*.

Let us see cata-fusion at work in proving that $(\geqslant)$ is transitive:

$$
\begin{array}{ll}
& (\geqslant) \cdot (\geqslant) \ \subseteq \ (\geqslant) \\
\Leftarrow & \qquad \{ \ \text{ping-pong} \ \} \\
& (\geqslant) \cdot (\geqslant) = (\geqslant) \\
\equiv & \qquad \{ \ \text{definition (8.4) twice} \ \} \\
& (\geqslant) \cdot (\!|\, [\top, \mathsf{succ}] \,|\!) = (\!|\, [\top, \mathsf{succ}] \,|\!) \\
\Leftarrow & \qquad \{ \ \text{cata-fusion (8.13)} \ \} \\
& (\geqslant) \cdot [\top, \mathsf{succ}] \subseteq [\top, \mathsf{succ}] \cdot (id + (\geqslant)) \\
\equiv & \qquad \{ \ \text{coproducts (5.114), etc)} \ \} \\
& (\geqslant) \cdot \mathsf{succ} = \mathsf{succ} \cdot (\geqslant) \\
\equiv & \qquad \{ \text{cancellation (8.12)} \ \} \\
& \text{TRUE}
\end{array}
$$

From (8.6, 8.7) two “weaker” versions of cata-fusion can be easily derived:

$$Q \cdot (\!| S |\!) \ \subseteq \ (\!| R |\!) \quad \Leftarrow \quad Q \cdot S \ \subseteq \ R \cdot \mathsf{F}\, Q \tag{8.14}$$

$$(\!| R |\!) \ \subseteq \ Q \cdot (\!| S |\!) \quad \Leftarrow \quad R \cdot \mathsf{F}\, Q \ \subseteq \ Q \cdot S \tag{8.15}$$

These are also quite useful in calculations. Consider, for instance, that R in $(\!| R |\!)$ is injective, that is, $R^\circ \cdot R \ \subseteq \ id$. Will $(\!| R |\!)$ be injective too? We calculate:

$$
\begin{array}{ll}
& (\!| R |\!) \cdot (\!| R |\!)^\circ \ \subseteq \ id \\
\equiv & \qquad \{ \ \text{cata reflection} \ \}
\end{array}
$$

$$
\begin{array}{ll}
& (\!|R|\!)^\circ \cdot (\!|R|\!) \subseteq (\!|\mathsf{in}|\!) \\
\Leftarrow & \quad \{ \text{ relational cata-fusion (8.14) } \} \\
& (\!|R|\!)^\circ \cdot R \subseteq \mathsf{in} \cdot \mathsf{F}\ (\!|R|\!)^\circ \\
\equiv & \quad \{ \text{ cata-cancellation ; converses } \} \\
& \mathsf{in} \cdot \mathsf{F}\ (\!|R|\!)^\circ \cdot R^\circ \cdot R \subseteq \mathsf{in} \cdot \mathsf{F}\ (\!|R|\!)^\circ \\
\Leftarrow & \quad \{ \text{ monotonicity of composition } \} \\
& R^\circ \cdot R \subseteq id
\end{array}
$$

Thus injectivity is preserved by the relational catamorphism combinator.

***Exercise** 8.1. Adapt the previous argument to proving that $(\!|R|\!)$ is entire provided R is entire, now using the other relational cata-fusion law (8.15).*

□

8.2 RELATIONAL HYLOMORPHISMS

Given F-algebras $R : A \leftarrow \mathsf{F}\ A$ and $S : B \leftarrow \mathsf{F}\ B$, the composition

$$H = (\!|R|\!) \cdot (\!|S|\!)^\circ \tag{8.16}$$

of type $A \leftarrow B$, is usually referred to as a *hylomorphism* [12]. As an example, define $\delta = [id, id]$. Then the $(1+)$-hylomorphism $H = (\!|Q \cdot \delta|\!) \cdot (\!|\delta|\!)^\circ$ is known as the transitive closure of Q, usually denoted by Q^+, cf:

$$
\begin{array}{ll}
& X = [Q, Q] \cdot (id + X) \cdot [id, id]^\circ \\
= & \quad \{ \text{ relational coproducts } \} \\
& X = [Q, Q \cdot X] \cdot [id, id]^\circ \\
= & \quad \{ \text{ relational coproducts } \} \\
& X = Q \cup Q \cdot X
\end{array}
$$

□

The intermediate type μ_{F} generated by $(\!|S|\!)^\circ$ and consumed by $(\!|R|\!)$ is known as the *virtual data structure* [81] of the hylomorphism. This is regarded as the basis of so-called *divide-and-conquer* programming strategies. The opposite composition $(\!|S|\!)^\circ \cdot (\!|R|\!)$, for suitably typed S and R, is sometimes termed a *metamorphism* [25].

It can be shown that $H = (\!|R|\!) \cdot (\!|S|\!)^\circ$ is the least solution (fixpoint) of the relational equation $X = R \cdot (\mathsf{F}\ X) \cdot S^\circ$.

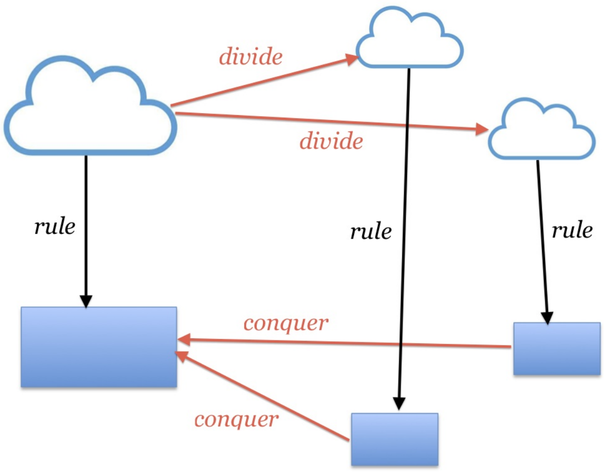


Figure 8.1.: The divide & conquer programming metaphor.

FIXPOINTS. Let ψ be a monotonic, relation-valued function. Any solution to the equation $X = \psi\ X$ is usually termed a *fixpoint* of ψ, and any solution to the equation $\psi\ X \subseteq X$ a *pre-fixpoint* of ψ. The least fixpoint of ψ is usually denoted by $\mu\ \psi$. It can be shown [3] that $\mu\ \psi$ satisfies

$$\mu\,\psi \subseteq X \quad \Leftarrow \quad \psi\,X \subseteq X \tag{8.17}$$

that is, $\mu\ \psi$ is also the least pre-fixpoint of monotonic function ψ. As an example, from (8.7) we see that $(\!|R|\!)$ is the least pre-fixpoint of $\psi\ X = R \cdot \mathsf{F}\ X \cdot \mathsf{in}^\circ$. Below we show that $\mu\ \psi = (\!|R|\!) \cdot (\!|S|\!)^\circ$ of

$$\psi\ X = R \cdot \mathsf{F}\ X \cdot S^\circ$$

via (8.17):

$$\begin{array}{cl}
& (\!|R|\!) \cdot (\!|S|\!)^\circ \subseteq X \\
\equiv & \quad \{ \text{ indirect equality over the cancellation of both catas; converses } \} \\
& R \cdot \mathsf{F}\ (\!|R|\!) \cdot \mathsf{in}^\circ \cdot \mathsf{in} \cdot \mathsf{F}\ (\!|S|\!)^\circ \cdot S^\circ \subseteq X \\
\equiv & \quad \{ \ \mathsf{in}^\circ \cdot \mathsf{in} = id; \text{relator } \mathsf{F} \ \} \\
& R \cdot \mathsf{F}\ ((\!|R|\!) \cdot (\!|S|\!)^\circ) \cdot S^\circ \subseteq X \\
\Leftarrow & \quad \{ \text{ since } (\!|R|\!) \cdot (\!|S|\!)^\circ \subseteq X\text{; relator } \mathsf{F}\text{; monotonicity } \} \\
& R \cdot \mathsf{F}\ X \cdot S^\circ \subseteq X \\
\Box &
\end{array}$$

In summary:

$$(\!|R|\!) \cdot (\!|S|\!)^\circ \subseteq X \quad \Leftarrow \quad R \cdot \mathsf{F}\ X \cdot S^\circ \subseteq X \tag{8.18}$$

[3] See e.g. [5].

Thus $(\!| R |\!) \cdot (\!| S |\!)^\circ$ is the least pre-fixpoint of ψ, and therefore its least fixpoint (solution). The following notation dispenses with naming ψ explicitly:

$$\langle \mu\, X \ :: \ R \cdot \mathsf{F}\, X \cdot S^\circ \rangle = (\!| R |\!) \cdot (\!| S |\!)^\circ \qquad (8.19)$$

Note how (8.18) generalizes (8.7), by reflection ($S = \mathsf{in}$). Similarly,

$$X \ \subseteq \ (\!| R |\!) \cdot (\!| S |\!)^\circ \quad \Leftarrow \quad X \ \subseteq \ R \cdot \mathsf{F}\, X \cdot S^\circ \qquad (8.20)$$

will generalize (8.6). Finally note that the image $(\!| R |\!) \cdot (\!| R |\!)^\circ$ of a catamorfism is a hylomorphism. By straight application of (8.18,8.20) we obtain

$$\begin{array}{l} (\!| R |\!) \cdot (\!| R |\!)^\circ \ \subseteq \ id \quad \Leftarrow \quad R \cdot R^\circ \ \subseteq \ id \\ id \ \subseteq \ (\!| R |\!) \cdot (\!| R |\!)^\circ \quad \Leftarrow \quad id \ \subseteq \ R \cdot R^\circ \end{array}$$

Thus catamorphisms preserve simplicity and surjectivity.

8.3 INDUCTIVE PREDICATES

Coreflexive catamorphisms (8.9) correspond to inductive predicates over the inductive structure μ_F. The smallest such catamorphism is $\perp$,

$$\perp = (\!| R |\!) \qquad \equiv \qquad \perp = R \cdot (\mathsf{F} \perp)$$

cf. (8.2). Such is the case of $(\!| [\perp, \mathsf{succ}] |\!)$, for instance.

A standard way of encoding inductive predicates is to pre-condition the initial algebra in,

$$\Phi = (\!| \mathsf{in} \cdot \Phi_p |\!)$$

for some predicate $\mathbb{B} \xleftarrow{p} \mathsf{F}\, \mu_\mathsf{F}$. For instance, $p = [true, even \cdot \pi_1]$ over $\mathsf{F}\ X = 1 + \mathbb{N}_0 \times X$ will let lists of even natural numbers pass, failing otherwise.

Another example of inductive predicate on finite lists is $\Psi = (\!| \mathsf{in} \cdot (id + \Phi_p) |\!)$ where $p\ (a, x) = \langle \forall\ a' \ : \ a' \in \mathsf{elems}\ x : \ a \leqslant a' \rangle$, which fails for any input list which is not in ascending order.

8.4 INDUCTIVE EQUIVALENCE RELATIONS

This section focusses on equivalence relations over inductive data types. Let $\mu_\mathsf{F} \xleftarrow{\mathsf{in}} \mathsf{F}\, \mu_\mathsf{F}$, and let $A \xleftarrow{k} \mathsf{F}\, A$ be given, so that $A \xleftarrow{(\!| k |\!)} \mu_\mathsf{F}$. It turns out that not only is $R = \frac{(\!| k |\!)}{(\!| k |\!)}$ itself a relational catamorphism

$$R = (\!| R \cdot \mathsf{in} |\!)$$

of type $\mu_\mathsf{F} \leftarrow \mu_\mathsf{F}$, but also it is a *congruence* for the algebra in. This follows from the following results:

- Let R be a congruence for an algebra $h : \mathsf{F}\ A \to A$ of functor F, that is, relational type

$$R \xleftarrow{h} \mathsf{F}\ R \tag{8.21}$$

holds and R is an equivalence relation. Then (8.21) is equivalent to:

$$R \cdot h = R \cdot h \cdot (\mathsf{F}\ R) \tag{8.22}$$

- For the particular case $h = \mathsf{in}$, (8.22) is equivalent to:

$$R = (\!| R \cdot \mathsf{in} |\!) \tag{8.23}$$

- For R presented as a kernel $R = \frac{f}{f}$, (8.21) is also equivalent to

$$f \cdot h \leqslant \mathsf{F} f \tag{8.24}$$

where $\leqslant$ is the injectivity preorder (5.234). [4]

A standard result in algebraic specification states that if a function f defined on an initial algebra is a catamorphism then $\frac{f}{f}$ is a congruence [19, 26]. We give below a proof that frames this result in the ones above by making $R = \frac{(\!| k |\!)}{(\!| k |\!)}$ in (8.23) and calculating:

$$
\begin{array}{ll}
& \frac{(\!| k |\!)}{(\!| k |\!)} = (\!| \frac{(\!| k |\!)}{(\!| k |\!)} \cdot \mathsf{in} |\!) \\
\equiv & \qquad \{ \text{ universal property (8.2) ; metaphor algebra (\textbf{??})} \ \} \\
& \frac{(\!| k |\!) \cdot \mathsf{in}}{(\!| k |\!)} = \frac{(\!| k |\!) \cdot \mathsf{in}}{(\!| k |\!)} \cdot \frac{\mathsf{F}\ (\!| k |\!)}{\mathsf{F}\ (\!| k |\!)} \\
\equiv & \qquad \{ \text{ cancellation (8.12) ;} f \cdot \frac{f}{f} = f \ \}
\end{array}
$$

[4] Proof: Equality (8.23) follows immediately from (8.22) by cancellation (8.12). Next we show the equivalence between (8.22) and (8.21):

$$
\begin{array}{ll}
& R \cdot h = R \cdot h \cdot (\mathsf{F}\ R) \\
\equiv & \qquad \{ R \cdot h \subseteq R \cdot h \cdot (\mathsf{F}\ R) \text{ holds by } id \subseteq \mathsf{F}\ R\text{, since } id \subseteq R \ \} \\
& R \cdot h \cdot (\mathsf{F}\ R) \subseteq R \cdot h \\
\equiv & \qquad \{ \ (R \cdot) \text{ is a closure operation, see (8.25) below } \} \\
& h \cdot (\mathsf{F}\ R) \subseteq R \cdot h \\
\Box &
\end{array}
$$

The last step relies on the fact that composition with equivalence relations is a *closure* operation:

$$R \cdot S \subseteq R \cdot Q \quad \equiv \quad S \subseteq R \cdot Q \tag{8.25}$$

$$
\begin{array}{ll}
& \dfrac{(\!|k|\!) \cdot \mathsf{in}}{(\!|k|\!)} = \dfrac{k \cdot \mathsf{F}\ (\!|k|\!)}{(\!|k|\!)} \\
\Leftarrow & \quad \{\ \text{Leibniz}\ \} \\
& (\!|k|\!) \cdot \mathsf{in} = k \cdot \mathsf{F}\ (\!|k|\!) \\
\equiv & \quad \{\ \text{universal property (8.2)}\ \} \\
& \mathit{true} \\
\Box &
\end{array}
$$

For example, in the case $R = \mathit{Perm}$ (8.1), (8.23) instantiates to

$$\mathit{Perm} \cdot \mathsf{in} = \mathit{Perm} \cdot \mathsf{in} \cdot (\mathsf{F}\ \mathit{Perm})$$

whose useful part is:

$$\mathit{Perm} \cdot \mathsf{cons} = \mathit{Perm} \cdot \mathsf{cons} \cdot (\mathit{id} \times \mathit{Perm}) \tag{8.26}$$

In words, this means that permuting a sequence with at least one element is the same as adding it to the front of a permutation of the tail and permuting again.[5]

The main usefulness of (8.22,8.23) is that the inductive definition of a kernel equivalence relation generated by a catamorphism is such that the recursive branch (the F term) can be added or removed where convenient.

8.5 INDUCTIVE DATATYPE ISOMORPHISM REVISITED

Recall section 3.19 where properties of catamorphisms involving inductive types on both their input and output where considered. In particular, given two parametric, inductive datatypes

$$\begin{tikzcd}
\mathsf{T}_1\ A \arrow[rr, bend left, "\mathsf{out}_1"] \arrow[rr, phantom, "\cong"] & & \mathsf{B}_1\ (A, \mathsf{T}_1\ A) \arrow[ll, bend left, "\mathsf{in}_1"]
\end{tikzcd}$$

and

$$\begin{tikzcd}
\mathsf{T}_2\ A \arrow[rr, bend left, "\mathsf{out}_2"] \arrow[rr, phantom, "\cong"] & & \mathsf{B}_2\ (A, \mathsf{T}_2\ A) \arrow[ll, bend left, "\mathsf{in}_2"]
\end{tikzcd}$$

one would like to be able to decide whether T_1 and T_2 are isomorphic, that is, whether they can be mutually converted into each other without losing information.

[5] Recall the (Kronecker) *product* (5.107): $(b,d)\ (R \times S)\ (a,c)$ holds iff both $b\ R\ a$ and $d\ S\ c$ hold. Thus (8.26) is the same as

$$y\ \mathit{Perm}\ (a : x) = \langle \exists\ z\ :\ z\ \mathit{Perm}\ x :\ y\ \mathit{Perm}\ (a : z) \rangle$$

once written pointwise.

Think of a function $f : \mathsf{T}_1\ A \to \mathsf{T}_2\ A$ converting T_1-shaped data into T_2-shaped data, for instance, $f : \mathsf{BTree}\ A \to A^*$. Clearly, f is bound to access T_1-shaped data via out_1 and to generate T_2-shaped data via in_2, inductively:

$$f = \mathsf{in}_2 \cdot \ldots f \ldots \cdot \mathsf{out}_1$$

Looking at the types of in_2 and out_1, it is clear that one needs a bridge-function α between the base functors of the two inductive types. Using a diagram:

$$\begin{tikzcd}
\mathsf{T}_1\ A \arrow[rr, "\mathsf{out}_1"] \arrow[d, "f"'] & & \mathsf{B}_1\ (A, \mathsf{T}_1\ A) \arrow[d, "\mathsf{B}_1\ (id, f)"] \\
\mathsf{T}_2\ A & \mathsf{B}_2\ (A, \mathsf{T}_2\ A) \arrow[l, "\mathsf{in}_2"] & \mathsf{B}_1\ (A, \mathsf{T}_2\ A) \arrow[l, "\alpha"]
\end{tikzcd}$$

Clearly, $f = (\!|\, \mathsf{in}_2 \cdot \alpha \,|\!)$. In the example $f : \mathsf{BTree}\ A \to A^*$,

$$1 + A \times A^* \xleftarrow{\alpha} 1 + A \times (A^* \times A^*)$$

is the expected function that concatenates the two lists resulting from visiting the two sub-trees of a nonempty BTree. And f is the preorder traversal of the input tree.

It is intuitive that f loses information: the best $\alpha = id + id \times \ldots$ can do to preserve information is to concatenate the two lists in the "...". But, from the resulting list one cannot rebuild the original lists, since concatenation is not injective, for instance: $[a, b] +\!\!+ [c] = [a] +\!\!+ [b, c]$. Choosing one the input lists would be worse and issuing the empty list would be even worser.

How can one be sure that f is a bijection (isomorphism)? Recall the rule of thumb (5.38) that tells us that a function f is bijective iff f° is a function. $f^\circ = (\!|\, \mathsf{in}_2 \cdot \alpha \,|\!)^\circ$ thus needs to be entire and simple, that is, $(\!|\, \mathsf{in}_2 \cdot \alpha \,|\!)$ should be surjective and injective. Since catamorphisms preserve these properties, it will be enough if $\mathsf{in}_2 \cdot \alpha$ is surjective and injective, which reduces to α being so, because in_2 is bijective.

In summary, a bijective α will ensure that the two inductive types are isomorphic. In case α is a natural transformation, this corresponds to the base functors B_1 and B_2 being isomorphic. $\boxed{\text{TBC}}$

9

CALCULATIONAL PROGRAM REFINEMENT

This chapter will build mainly from references [75, 69].

Part III

CALCULATING WITH MATRICES

10

TOWARDS A LINEAR ALGEBRA OF PROGRAMMING

This part of the book will build upon references [70, 62, 74]. Another chapter will address the application of typed linear algebra to analytical data processing, cf. e.g. [68]. The LAoP Haskell library [2] already supports typed linear algebra.

A

BACKGROUND — EINDHOVEN QUANTIFIER CALCULUS

This appendix is a quick reference summary of section 4.3 of reference [5].

A.1 NOTATION

The Eindhoven quantifier calculus adopts the following notation standards:

- $\langle \forall\, x \;:\; R \;:\; T \rangle$ means: *"for **all** x in the range R, term T holds"*, where R and T are logical expressions involving x.
- $\langle \exists\, x \;:\; R \;:\; T \rangle$ means: *"for **some** x in the range R, term T holds"*.

A.2 RULES

The main rules of the Eindhoven quantifier calculus are listed below:

Trading:

$$\langle \forall\, k \;:\; R \wedge S \;:\; T \rangle \;=\; \langle \forall\, k \;:\; R \;:\; S \Rightarrow T \rangle \tag{A.1}$$
$$\langle \exists\, k \;:\; R \wedge S \;:\; T \rangle \;=\; \langle \exists\, k \;:\; R \;:\; S \wedge T \rangle \tag{A.2}$$

de Morgan:

$$\neg\langle \forall\, k \;:\; R \;:\; T \rangle \;=\; \langle \exists\, k \;:\; R \;:\; \neg T \rangle \tag{A.3}$$
$$\neg\langle \exists\, k \;:\; R \;:\; T \rangle \;=\; \langle \forall\, k \;:\; R \;:\; \neg T \rangle \tag{A.4}$$

One-point:

$$\langle \forall\, k \;:\; k = e \;:\; T \rangle \;=\; T[k := e] \tag{A.5}$$
$$\langle \exists\, k \;:\; k = e \;:\; T \rangle \;=\; T[k := e] \tag{A.6}$$

Nesting:

$$\langle \forall\, a, b \;:\; R \wedge S \;:\; T \rangle \;=\; \langle \forall\, a \;:\; R \;:\; \langle \forall\, b \;:\; S \;:\; T \rangle \rangle \tag{A.7}$$
$$\langle \exists\, a, b \;:\; R \wedge S \;:\; T \rangle \;=\; \langle \exists\, a \;:\; R \;:\; \langle \exists\, b \;:\; S \;:\; T \rangle \rangle \tag{A.8}$$

Rearranging-$\forall$:

$$\langle \forall\, k \;:\; R \vee S :\; T \rangle \;=\; \langle \forall\, k \;:\; R :\; T \rangle \wedge \langle \forall\, k \;:\; S :\; T \rangle \tag{A.9}$$

$$\langle \forall\, k \;:\; R :\; T \wedge S \rangle \;=\; \langle \forall\, k \;:\; R :\; T \rangle \wedge \langle \forall\, k \;:\; R :\; S \rangle \tag{A.10}$$

Rearranging-$\exists$:

$$\langle \exists\, k \;:\; R :\; T \vee S \rangle \;=\; \langle \exists\, k \;:\; R :\; T \rangle \vee \langle \exists\, k \;:\; R :\; S \rangle \tag{A.11}$$

$$\langle \exists\, k \;:\; R \vee S :\; T \rangle \;=\; \langle \exists\, k \;:\; R :\; T \rangle \vee \langle \exists\, k \;:\; S :\; T \rangle \tag{A.12}$$

Splitting:

$$\langle \forall\, j \;:\; R :\; \langle \forall\, k \;:\; S :\; T \rangle \rangle \;=\; \langle \forall\, k \;:\; \langle \exists\, j \;:\; R :\; S \rangle :\; T \rangle \tag{A.13}$$

$$\langle \exists\, j \;:\; R :\; \langle \exists\, k \;:\; S :\; T \rangle \rangle \;=\; \langle \exists\, k \;:\; \langle \exists\, j \;:\; R :\; S \rangle :\; T \rangle \tag{A.14}$$

B

HASKELL SUPPORT LIBRARY

This library, written in the Haskell functional programming language, is still evolving.

$$\textbf{infix } 5 \times \\ \textbf{infix } 4 +$$

Products

$$\langle \cdot, \cdot \rangle :: (a \to b) \to (a \to c) \to a \to (b, c) \\ \langle f, g \rangle\ x = (f\ x, g\ x)$$

$$(\times) :: (a \to b) \to (c \to d) \to (a, c) \to (b, d) \\ f \times g = \langle f \cdot \pi_1, g \cdot \pi_2 \rangle$$

The 0-adic split is the unique function of its type

$$(!) :: a \to () \\ (!) = \underline{()}$$

Renamings:

$$\pi_1 = \mathsf{fst} \\ \pi_2 = \mathsf{snd}$$

Coproduct

Renamings:

$$i_1 = Left \\ i_2 = Right$$

Either is predefined:

$$(+) :: (a \to b) \to (c \to d) \to a + c \to b + d \\ f + g = [i_1 \cdot f, i_2 \cdot g]$$

McCarthy's conditional:

$$p \to f\ ,\ g = [f, g] \cdot p?$$

Exponentiation

Curry is predefined.

$$
\begin{aligned}
&ap :: (a \to b, a) \to b\\
&ap = \widehat{(\$)}
\end{aligned}
$$

Functor:

$$
\begin{aligned}
&\cdot^{\cdot} :: (b \to c) \to (a \to b) \to a \to c\\
&f^{\cdot} = \overline{f \cdot ap}
\end{aligned}
$$

Pair-to-predicate isomorphism (2.101):

$$
\begin{aligned}
&p2p :: (b, b) \to \mathbb{B} \to b\\
&p2p\ p\ b = \mathbf{if}\ b\ \mathbf{then}\ (\pi_2\ p)\ \mathbf{else}\ (\pi_1\ p)
\end{aligned}
$$

The exponentiation functor is $(a \to)$ predefined:

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Functor}\ ((\to)\ s)\ \mathbf{where}\\
&\mathit{fmap}\ f\ g = f \cdot g
\end{aligned}
$$

Guards

$$
\begin{aligned}
&\cdot? :: (a \to \mathbb{B}) \to a \to a + a\\
&p?\ x = \mathbf{if}\ p\ x\ \mathbf{then}\ i_1\ x\ \mathbf{else}\ i_2\ x
\end{aligned}
$$

Others

$\underline{\cdot} :: a \to b \to a$ such that $\underline{a}\ x = a$ is predefined.

Natural isomorphisms

$$
\begin{aligned}
&\mathsf{swap} :: (a, b) \to (b, a)\\
&\mathsf{swap} = \langle \pi_2, \pi_1 \rangle\\
&\mathsf{assocr} :: ((a, b), c) \to (a, (b, c))\\
&\mathsf{assocr} = \langle \pi_1 \cdot \pi_1, \mathsf{snd} \times id \rangle\\
&\mathsf{assocl} :: (a, (b, c)) \to ((a, b), c)\\
&\mathsf{assocl} = \langle id \times \pi_1, \pi_2 \cdot \pi_2 \rangle\\
&\mathsf{undistr} :: a, b + a, c \to (a, b + c)\\
&\mathsf{undistr} = [id \times i_1, id \times i_2]\\
&\mathsf{undistl} :: b, c + a, c \to (b + a, c)\\
&\mathsf{undistl} = [i_1 \times id, i_2 \times id]\\
&\mathsf{coswap} :: a + b \to b + a\\
&\mathsf{coswap} = [i_2, i_1]\\
&\mathsf{coassocr} :: a + b + c \to a + b + c\\
&\mathsf{coassocr} = [id + i_1, i_2 \cdot i_2]\\
&\mathsf{coassocl} :: b + a + c \to b + a + c
\end{aligned}
$$

$$\mathsf{coassocl} = [i_1 \cdot i_1\,, i_2 + id]$$

$$\mathsf{distl} :: (c + a, b) \rightarrow c, b + a, b$$

$$\mathsf{distl} = \widehat{[\overline{i_1}\,, \overline{i_2}]}$$

$$\begin{array}{l} \mathsf{distr} :: (b, c + a) \rightarrow b, c + b, a \\ \mathsf{distr} = (\mathsf{swap} + \mathsf{swap}) \cdot \mathsf{distl} \cdot \mathsf{swap} \end{array}$$

$$\begin{array}{l} \mathit{flatr} :: (a, (b, c)) \rightarrow (a, b, c) \\ \mathit{flatr}\ (a, (b, c)) = (a, b, c) \end{array}$$

$$\begin{array}{l} \mathit{flatl} :: ((a, b), c) \rightarrow (a, b, c) \\ \mathit{flatl}\ ((b, c), d) = (b, c, d) \end{array}$$

$$br = \langle id, ! \rangle$$

$$bl = \mathsf{swap} \cdot br$$

Class bifunctor

$$\begin{array}{l} \textbf{class}\ \mathit{BiFunctor}\, f\ \textbf{where} \\ \quad \mathit{bmap} :: (a \rightarrow b) \rightarrow (c \rightarrow d) \rightarrow (f\ a\ c \rightarrow f\ b\ d) \\ \textbf{instance}\ \mathit{BiFunctor}\ \cdot + \cdot\ \textbf{where} \\ \quad \mathit{bmap}\, f\ g = f + g \\ \textbf{instance}\ \mathit{BiFunctor}\ (,)\ \textbf{where} \\ \quad \mathit{bmap}\, f\ g = f \times g \end{array}$$

Monads

Kleisli monadic composition:

$$\begin{array}{l} \textbf{infix}\ 4 \bullet \\ (\bullet) :: \mathit{Monad}\ a \Rightarrow (b \rightarrow a\ c) \rightarrow (d \rightarrow a\ b) \rightarrow d \rightarrow a\ c \\ (f \bullet g)\ a = (g\ a) \gg\!= f \end{array}$$

Multiplication, also known as join:

$$\begin{array}{l} \mathit{mult} :: (\mathit{Monad}\ m) \Rightarrow m\ (m\ b) \rightarrow m\ b \\ \mathit{mult} = (\gg\!= id) \end{array}$$

Monadic binding:

$$\begin{array}{l} ap' :: (\mathit{Monad}\ m) \Rightarrow (a \rightarrow m\ b, m\ a) \rightarrow m\ b \\ ap' = \widehat{\mathit{flip}\ (\gg\!=)} \end{array}$$

List monad:

$$\begin{array}{l} \mathsf{singl} :: a \rightarrow [a] \\ \mathsf{singl} = \mathsf{return} \end{array}$$

Strong monads:

$$\begin{array}{l} \textbf{class}\ (\mathit{Functor}\, f, \mathit{Monad}\, f) \Rightarrow \mathit{Strong}\, f\ \textbf{where} \\ \quad \mathit{rstr} :: (f\ a, b) \rightarrow f\ (a, b) \\ \quad \mathit{rstr}\ (x, b) = \textbf{do}\ a \leftarrow x; \mathsf{return}\ (a, b) \end{array}$$

$$\begin{array}{l}
\quad \mathit{lstr} :: (b,f\ a) \rightarrow f\ (b,a)\\
\quad \mathit{lstr}\ (b,x) = \mathbf{do}\ a \leftarrow x; \mathsf{return}\ (b,a)\\
\mathbf{instance}\ \mathit{Strong}\ \mathsf{IO}\\
\mathbf{instance}\ \mathit{Strong}\ [\,]\\
\mathbf{instance}\ \mathit{Strong}\ \mathit{Maybe}
\end{array}$$

Double strength:

$$\begin{array}{l}
\mathit{dstr} :: \mathit{Strong}\ m \Rightarrow (m\ a, m\ b) \rightarrow m\ (a,b)\\
\mathit{dstr} = \mathit{rstr} \bullet \mathit{lstr}
\end{array}$$

Exercise 4.8.13 in Jacobs' "Introduction to Coalgebra" [38]:

$$\begin{array}{l}
\mathit{splitm} :: \mathit{Strong}\ \mathsf{F} \Rightarrow \mathsf{F}\ (a \rightarrow b) \rightarrow a \rightarrow \mathsf{F}\ b\\
\mathit{splitm} = \overline{\mathit{fmap}\ \mathit{ap} \cdot \mathit{rstr}}
\end{array}$$

Monad transformers:

$$\begin{array}{ll}
\mathbf{class}\ (\mathit{Monad}\ m, \mathit{Monad}\ (t\ m)) \Rightarrow \mathit{MT}\ t\ m\ \mathbf{where} & \text{-- monad transformer class}\\
\quad \mathit{lift} :: m\ a \rightarrow t\ m\ a &
\end{array}$$

Nested lifting:

$$\begin{array}{l}
\mathit{dlift} :: (\mathit{MT}\ t\ (t_1\ m), \mathit{MT}\ t_1\ m) \Rightarrow m\ a \rightarrow t\ (t_1\ m)\ a\\
\mathit{dlift} = \mathit{lift} \cdot \mathit{lift}
\end{array}$$

Basic functions, abbreviations

$$\begin{array}{l}
\mathsf{zero} = \underline{0}\\
\mathsf{one} = \underline{1}\\
\mathsf{nil} = \underline{[\,]}\\
\mathsf{cons} = \widehat{:}\\
\mathsf{add} = \widehat{+}\\
\mathsf{mul} = \widehat{*}\\
\mathsf{conc} = \widehat{+\!\!+}\\
\mathit{inMaybe} :: +a \rightarrow \mathit{Maybe}\ a\\
\mathit{inMaybe} = [\underline{\mathsf{Nothing}}, \mathsf{Just}]
\end{array}$$

More advanced

$$\begin{array}{l}
\mathbf{class}\ (\mathit{Functor}\ f) \Rightarrow \mathit{Unzipable}\ f\ \mathbf{where}\\
\quad \mathit{unzp} :: f\ (a,b) \rightarrow (f\ a, f\ b)\\
\quad \mathit{unzp} = \langle \mathit{fmap}\ \pi_1, \mathit{fmap}\ \pi_2 \rangle\\
\mathbf{class}\ \mathit{Functor}\ g \Rightarrow \mathit{DistL}\ g\ \mathbf{where}\\
\quad \lambda :: \mathit{Monad}\ m \Rightarrow g\ (m\ a) \rightarrow m\ (g\ a)\\
\mathbf{instance}\ \mathit{DistL}\ [\,]\ \mathbf{where}\ \lambda = \mathsf{sequence}\\
\mathbf{instance}\ \mathit{DistL}\ \mathit{Maybe}\ \mathbf{where}
\end{array}$$

$$\lambda\ \mathsf{Nothing} = \mathsf{return}\ \mathsf{Nothing}$$
$$\lambda\ (\mathsf{Just}\ a) = mp\ \mathsf{Just}\ a\ \mathbf{where}\ mp\,f = (\mathsf{return} \cdot f) \bullet id$$

Convert Monad into Applicative:

$$aap :: Monad\ m \Rightarrow m\ (a \to b) \to m\ a \to m\ b$$
$$aap\ mf\ mx = \mathbf{do}\ \{f \leftarrow mf; x \leftarrow mx; \mathsf{return}\ (f\ x)\}$$

BIBLIOGRAPHY

[1] C. Aarts, R.C. Backhouse, P. Hoogendijk, E.Voermans, and J. van der Woude. A relational theory of datatypes, December 1992. Available from `www.cs.nott.ac.uk/~rcb`.

[2] J.N. Oliveira Armando Santos. Type your matrices for great good: a haskell library of typed matrices and applications (functional pearl). In Tom Schrijvers, editor, *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020*, pages 54–66. ACM, 2020.

[3] K. Backhouse and R.C. Backhouse. Safety of abstract interpretations for free, via logical relations and Galois connections. *SCP*, 15(1–2):153–196, 2004.

[4] R.C. Backhouse. On a relation on functions. In *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, pages 7–18, New York, NY, USA, 1990. Springer-Verlag.

[5] R.C. Backhouse. *Mathematics of Program Construction*. Univ. of Nottingham, 2004. Draft of book in preparation. 608 pages.

[6] R.C. Backhouse and M.M. Fokkinga. The associativity of equivalence and the Towers of Hanoi problem. *Information Processing letters*, 77(2–4):71–76, 2001.

[7] Roland Backhouse. *Algorithmic Problem Solving*. Wiley Publishing, 1st edition, 2011.

[8] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *CACM*, 21(8):613–639, August 1978.

[9] L.S. Barbosa. *Components as Coalgebras*. University of Minho, December 2001. Ph. D. thesis.

[10] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. of 31th ACM Symposium on Principles of Programming Languages*, pages 14–25, Venice, 2004.

[11] R. Bird. Introduction to Functional Programming. Series in Computer Science. Prentice-Hall International, 2nd edition, 1998. C.A.R. Hoare, series editor.

[12] R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall, 1997.

[13] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, January 1977.

[14] V. Cerf. Where is the science in computer science? *CACM*, 55(10):5, October 2012.

[15] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345, 1936.

[16] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.

[17] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'82, pages 207–212, New York, NY, USA, 1982. ACM.

[18] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[19] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer-Verlag, New-York, 1985.

[20] M. Erwig and S. Kollmannsberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16:21–34, January 2006.

[21] S. Feferman. Tarski's influence on computer science. *Logical Methods in Computer Science*, 2:1–1–13, 2006.

[22] R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19, pages 19–32. American Mathematical Society, 1967. Proc. Symposia in Applied Mathematics.

[23] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[24] P.J. Freyd and A. Scedrov. *Categories, Allegories*, volume 39 of *Mathematical Library*. North-Holland, 1990.

[25] J. Gibbons. Metamorphisms: Streaming representation-changers. *SCP*, 65:108–139, 2007.

[26] J. Gibbons. Kernels, in a nut shell. *JLAMP*, 85(5, Part 2):921–930, 2016.

[27] J. Gibbons and R. Hinze. Just do it: simple monadic equational reasoning. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP'11, pages 2–14, New York, NY, USA, 2011. ACM.

[28] Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch. When is a function a fold or an unfold?, 2001. WGP, July 2001 (slides).

[29] S. Givant. The calculus of relations as a foundation for mathematics. *J. Autom. Reasoning*, 37(4):277–322, 2006.

[30] A.S. Green, P.L. Lumsdaine, N.J. Ross, P. Selinger, and B. Valiron. An introduction to quantum programming in Quipper. *CoRR*, cs.PL(arXiv:1304.5485v1), 2013.

[31] Ralf Hinze. Adjoint folds and unfolds — an extended study. *Science of Computer Programming*, 78(11):2108–2159, 2013.

[32] Ralf Hinze. Adjoint folds and unfolds — an extended study. *Science of Computer Programming*, 78(11):2108–2159, 2013.

[33] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Conjugate hylomorphisms – or: The mother of all structured recursion schemes. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 527–538, New York, NY, USA, 2015. ACM.

[34] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12,10:576–580, 583, October 1969.

[35] P. Hudak. The Haskell School of Expression - Learning Functional Programming Through Multimedia. Cambridge University Press, 1st edition, 2000. ISBN 0-521-64408-9.

[36] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), 1993.

[37] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge Mass., 2012. Revised edition, ISBN 0-262-01715-2.

[38] B. Jacobs. *Introduction to Coalgebra. Towards Mathematics of States and Observations*. Cambridge University Press, 2016.

[39] P. Jansson and J. Jeuring. Polylib — a library of polytypic functions. In *Workshop on Generic Programming (*WGP'98*), Marstrand, Sweden*, 1998.

[40] J. Jeuring and P. Jansson. Polytypic programming. In *Advanced Functional Programming*, number 1129 in LNCS, pages 68–114. Springer-Verlag, 1996.

[41] He Jifeng, Liu Zhiming, and Li Xiaoshan. A contract-oriented approach to component-based programming. In Z. Liu, editor, *Proc. of FACS'03, (*Formal Approaches to Component Software*)*, Pisa, Spetember 2003.

[42] C.B. Jones. Turing and software verification. Technical Report CS-TR-1441, Newcastle University, 2014.

[43] S.L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003. Also published as a Special Issue of the Journal of Functional Programming, 13(1) Jan. 2003.

[44] E. Kreyszig. *Advanced Engineering Mathematics*. J. Wiley & Sons, 6th edition, 1988.

[45] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P.L. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

[46] D. Laws. Who invented the transistor? — computer history museum. `http://www.computerhistory.org/atchm/who-invented-the-transistor/`, December 2013. (Accessed on 04/02/2019).

[47] QingMing Ma and John C. Reynolds. Types, abstractions, and parametric polymorphism, part 2. In Stephen D. Brookes, Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *MFPS*, volume 598 of *Lecture Notes in Computer Science*, pages 1–40. Springer-Verlag, 1991.

[48] H.D. Macedo and J.N. Oliveira. Typing linear algebra: A biproduct-oriented approach. *SCP*, 78(11):2160–2191, 2013.

[49] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[50] R.D. Maddux. The origin of relation algebras in the development and axiomatization of the calculus of relations. *Studia Logica*, 50:421–455, 1991.

[51] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[52] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[53] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986. D. Gries, series editor.

[54] J. McCarthy. Towards a mathematical science of computation. In C.M. Popplewell, editor, *Proc. of* IFIP *62*, pages 21–28, Amsterdam-London, 1963. North-Holland Pub. Company.

[55] S. Mehner, D. Seidel, L. Straßburger, and J. Voigtländer. Parametricity and proving free theorems for functional-logic languages. PPDP'14, pages 19–30, New York, NY, USA, 2014. ACM.

[56] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In S. Peyton Jones, editor, *Proceedings of Functional Programming Languages and Computer Architecture (FPCA95)*, 1995.

[57] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[58] E. Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[59] S-C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In *MPC 2004*, pages 289–313, 2004.

[60] S.-C. Mu and J.N. Oliveira. Programming from Galois connections. *JLAP*, 81(6):680–704, 2012.

[61] S.C. Mu and R. Bird. Quantum functional programming, 2001. 2nd Asian Workshop on Programming Languages and Systems, KAIST, Dajeaon, Korea, December 17-18, 2001.

[62] D. Murta and J.N. Oliveira. A study of risk-aware program transformation. *SCP*, 110:51–77, 2015.

[63] P. Naur and B. Randell, editors. *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968*. Scientific Affairs Division, NATO, 1969.

[64] F. Nebeker. *Dawn of the Electronic Age: Electrical Technologies in the Shaping of the Modern World, 1914 to 1945*. Wiley-IEEE Press, 2009.

[65] A. Neri, R.S. Barbosa, and J.N. Oliveira. Compiling quantamorphisms for the IBM Q-Experience. 2020. Submitted to IEEE Trans. Soft. Eng.

[66] P. Nunes. *Libro de Algebra en Arithmetica y Geometria*. Original edition by Arnoldo Birckman (Anvers), 1567.

[67] A. Oettinger. The hardware-software complementarity. *Commun. ACM*, 10:604–606, October 1967.

[68] J. N. Oliveira and H. D. Macedo. The data cube as a typed linear algebra operator. In *Proc. of the 16th Int. Symposium on Database Programming Languages*, DBPL '17, pages 6:1–6:11, New York, NY, USA, 2017. ACM. (DOI).

[69] J.N. Oliveira. *Transforming Data by Calculation*. In *GTTSE'07*, volume 5235 of *LNCS*, pages 134–195. Springer-Verlag, 2008.

[70] J.N. Oliveira. Towards a linear algebra of programming. *FAoC*, 24(4-6):433–458, 2012.

[71] J.N. Oliveira. Lecture notes on relational methods in software design, 2015. Available from ResearchGate:
`https://www.researchgate.net/profile/Jose_Oliveira34`.

[72] J.N. Oliveira. Programming from metaphorisms. *JLAMP*, 94:15–44, January 2018. (DOI).

[73] J.N. Oliveira and M.A. Ferreira. Alloy meets the algebra of programming: A case study. *IEEE Trans. Soft. Eng.*, 39(3):305–326, 2013.

[74] J.N. Oliveira and V.C. Miraldo. "Keep definition, change category" — a practical approach to state-based system calculi. *JLAMP*, 85(4):449–474, 2016. (DOI).

[75] J.N. Oliveira and C.J. Rodrigues. *Pointfree Factorization of Operation Refinement*. In FM'06, volume 4085 of *Lecture Notes in Computer Science*, pages 236–251. Springer-Verlag, 2006.

[76] David Lorge Parnas. Really rethinking "formal methods". *IEEE Computer*, 43(1):28–34, 2010.

[77] M.S. Paterson and C.E. Hewitt. Comparative schematology. In *Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 119–127, August 1970.

[78] J.C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing 83*, pages 513–523, 1983.

[79] S. K. Routray. History of Electronics. *The 2004 IEEE Conference on the History of Electronics (CHE2004)*, 2004.

[80] G. Schmidt. *Relational Mathematics*. Number 132 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, November 2010.

[81] D. Swierstra and O. de Moor. Virtual data structures. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, volume 755 of *LNCS*, pages 355–371. Springer-Verlag, 1993.

[82] A. Tarski and S. Givant. *A Formalization of Set Theory without Variables*. American Mathematical Society, 1987. AMS Colloquium Publications, volume 41, Providence, Rhode Island.

[83] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[84] G. Villavicencio and J.N. Oliveira. *Reverse Program Calculation Supported by Code Slicing* . In *Proceedings of the Eighth Working Conference on Reverse Engineering (*WCRE 2001*) 2-5 October 2001, Stuttgart, Germany*, pages 35–46. IEEE Computer Society, 2001.

[85] P.L. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, pages 347–359, London, Sep. 1989. ACM.

[86] M. Winter. Arrow categories. *Fuzzy Sets and Systems*, 160(20):2893–2909, 2009.

[87] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.