Algebra of Programming

C.A.R. Hoare, Series Editor

APT., K.R., *From Logic to Prolog*
ARNOLD, A., *Finite Transition Systems*
ARNOLD, A. and GUESSARIAN, I., *Mathematics for Computer Science*
BARR, M. and WELLS, C., *Category Theory for Computing Science (2nd edn)*
BEN-ARI, M., *Principles of Concurrent and Distributed Programming*
BEN-ARI, M., *Mathematical Logic for Computer Science*
BEST, E., *Semantics of Sequential and Parallel Programs*
BIRD, R. and DE MOOR, O., *The Algebra of Programming*
BIRD, R. and WADLER, P., *Introduction to Functional Programming*
BOVET, D.P. and CRESCENZI, P., *Introduction to the Theory of Complexity*
DE BROCK, B., *Foundations of Semantic Databases*
BRODA, EISENBACH, KHOSHNEVISAN and VICKERS, *Reasoned Programming*
BRUNS, G., *Distributed Systems Analysis with CCS*
BURKE, E. and FOXLEY, E., *Logic Programming*
BURKE, E. and FOXLEY, E., *Logic and Its Applications*
CLEMENT, T., *Programming Language Paradigms*
DAHL, O.-J., *Verifiable Programming*
DUNCAN, E., *Microprocessor Programming and Software Development*
ELDER, J., *Compiler Construction*
ELLIOTT, R.J. and HOARE, C.A.R. (eds), *Scientific Applications of Multiprocessors*
FREEMAN, T.L. and PHILLIPS, R.C., *Parallel Numerical Algorithms*
GOLDSCHLAGER, L. and LISTER, A., *Computer Science: A modern introduction (2nd edn)*
GORDON, M.J.C., *Programming Language Theory and Its Implementation*
GRAY, P.M.D., KULKARNI, K.G. and PATON, N.W., *Object-oriented Databases*
HAYES, I. (ed.), *Specification Case Studies (2nd edn)*
HEHNER, E.C.R., *The Logic of Programming*
HINCHEY, M.G. and BOWEN, J.P., *Applications of Formal Methods*
HOARE, C.A.R., *Communicating Sequential Processes*
HOARE, C.A.R. and GORDON, M.J.C. (eds), *Mechanized Reasoning and Hardware Design*
HOARE, C.A.R. and JONES, C.B. (eds), *Essays in Computing Science*
HUGHES, J.G., *Database Technology: A software engineering approach*
HUGHES, J.G., *Object-oriented Databases*
INMOS LTD, *Occam 2 Reference Manual*
JONES, C.B., *Systematic Software Development Using VDM (2nd edn)*
JONES, C.B. and SHAW, R.C.F. (eds), *Case Studies in Systematic Software Development*
JONES, G. and GOLDSMITH, M., *Programming in Occam 2*
JONES, N.D., GOMARD, C.K. and SESTOFT, P., *Partial Evaluation and Automatic Program Generatio*
JOSEPH, M. (ed.), *Real-time Systems: Specification, verification and analysis*
KALDEWAIJ, A., *Programming: The derivation of algorithms*
KING, P.J.B., *Computer and Communications Systems Performance Modelling*
LALEMENT, R., *Computation as Logic*
McCABE, F.G., *Logic and Objects*
McCABE, F.G., *High-level Programmer's Guide to the 68000*
MEYER, B., *Introduction to the Theory of Programming Languages*
MEYER, B., *Object-oriented Software Construction*
MILNER, R., *Communication and Concurrency*
MITCHELL, R., *Abstract Data Types and Modula 2*
MORGAN, C., *Programming from Specifications* (2nd edn)
OMONDI, A.R., *Computer Arithmetic Systems*
PATON, COOPER, WILLIAMS and TRINDER, *Database Programming Languages*
PEYTON JONES, S.L., *The Implementation of Functional Programming Languages*

Algebra of Programming

Richard Bird
and
Oege de Moor
University of Oxford

Prentice Hall

An imprint of Pearson Education

Harlow, England · London · New York · Reading, Massachusetts · San Francisco
Toronto · Don Mills, Ontario · Sydney · Tokyo · Singapore · Hong Kong · Seoul
Taipei · Cape Town · Madrid · Mexico City · Amsterdam · Munich · Paris · Milan

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
http://www.pearsoneduc.com

First published by Prentice Hall

Printed and bound in Great Britain by
MPG Books Ltd, Bodmin, Cornwall

Library of Congress Cataloguing-in-Publication Data

Available from the publisher

British Library Cataloguing in Publication Data

A catalogue record for this book is available from
the British Library
ISBN 0-13-507245-X

---

10 9 8 7 6 5 4 3 2

04 03 02 01 00

Contents

Foreword

It is a great pleasure and privilege to introduce this book on the Algebra of Programming as the hundredth book in the Prentice Hall International Series in Computing Science. It develops and consolidates one of the abiding and central themes of the series: it codifies the basic laws of algorithmics, and shows how they can be used to classify many ingenious and important programs into families related by the algebraic properties of their specifications. The formulae and equations that you will see here share the elegance of those which underlie physics or chemistry or any other branch of basic science; and like them, they inspire our interest, enlarge our understanding, and hold out promise of enduring benefits in application.

Tony Hoare

Preface

Our purpose in this book is to show how to calculate programs. We describe an algebraic approach to programming, suitable both for the derivation of individual programs and for the study of programming principles in general. The programming principles we have in mind are those paradigms and strategies of program construction that form the core of the subject known as Algorithm Design. Examples of such principles include: dynamic programming, greedy algorithms, exhaustive search, and divide and conquer.

The main ideas of the algebraic approach are illustrated by an extensive study of optimisation problems, conducted in Chapters 7–10. These are problems that involve finding a largest, smallest, cheapest, and so on, member of a set of possible solutions satisfying some given constraints. It is surprising how many computational problems can be specified in terms of optimising some suitable measure, even problems that do not at first sight fall into the traditional areas of combinatorial optimisation. However, the book is not primarily about optimisation problems, rather it is about one approach to solving programming problems in general.

Our mathematical framework is a categorical calculus of relations. The calculus is categorical because we want to formulate algorithmic strategies without reference to specific datatypes, and relational because we need a degree of freedom in specification and proof that a calculus of functions alone would not provide. With the help of this calculus, the standard principles of algorithm design can be formulated as *theorems* about classes of problems whose specifications possess a particular structure. The problems are abstract in the sense that they are parameterised by one or more datatypes. These theorems say that, under appropriate conditions, a certain strategy works and leads to a particular form of abstract solution.

Specific algorithms for specific problems are obtained by checking that the conditions hold and instantiating the results. The solution may take the form of a function, but more usually a relation, characterised as the solution to a certain recursive equation. The recursive equation is then refined to a recursive program that delivers a function, and the result is translated into a functional programming language. All the programs derived in Chapters 7–10 follow this pattern, and the popular language Gofer (Jones 1994) is used to implement the results.

A categorical calculus provides not only a means for formulating algorithmic strategies abstractly, but also a smooth and integrated framework for conducting proofs. The style employed throughout the book is one of equational and inequational point-free reasoning with functions and relations. A point-free calculation is one in which the expressions under manipulation denote functions or relations, built using functional and relational composition as the basic combining form. In contrast, pointwise reasoning is reasoning conducted at the level of functional or relational application and expressed in a formalism such as the predicate calculus.

The point-free style is intrinsic to a categorical approach, but is less common in proofs about programs. One of the advantages of a point-free style is that one is unencumbered by many of the complications involved in manipulating formula dealing with bound variables introduced by explicit quantifications. Point-free reasoning is therefore well suited to mechanisation, though none of the many calculations recorded in this book were in fact produced with the help of a mechanical proof assistant.

Audience

The book is addressed primarily to the mathematically inclined functional programmer, though the non-functional – but still mathematically inclined – programmer is not excluded. Although we have taken pains to make the book as self-contained as possible, and have provided lots of exercises for self-study, the reader will need some mathematical background to appreciate and master the more abstract material.

A first course in functional programming will help quite a lot, since many of the ideas we describe can be found there in more concrete clothing. Prior exposure to the basic ideas of set theory would be a significant bonus, as would some familiarity with relations and equational reasoning in a logical calculus. The bibliographical remarks at the end of each chapter describe where appropriate background material can be found.

Outline

Roughly speaking, the first half of the book (Chapters 1–6) is devoted to basic theory, while the second half (Chapters 7–10) pursues the theme of finding efficient solutions for various kinds of optimisation problem. But most of the early chapters contain some applications of the theory to programming problems.

Chapter 1 reviews some basic concepts and techniques in functional programming, and the same ideas are presented again in categorical terms in Chapter 2. This material is followed in Chapter 3 with one or two simple applications to program derivation, as well as a discussion of additional categorical ideas. Building on this material, Chapters 4 and 5 present a categorical calculus of relations, and Chapter 6 contains a treatment of recursion in a relational setting. This chapter also contains discussions of various problems, including sorting and breadth-first search.

The methods explored in Chapters 7–10 fall into two basic kinds, depending on whether a solution to an optimisation problem is viewed as being composed out of smaller ones, or decomposed into smaller ones. The two views are complementary and individual problems can fall into both classes. Chapter 7 discusses greedy algorithms that assemble a solution to a problem by a bottom-up process of constructing solutions to smaller problems, while Chapter 10 studies another class of greedy algorithm that chooses an optimal decomposition at each stage.

The remaining chapters, Chapter 8 and Chapter 9, deal with similar views of dynamic programming. Each of these four chapters contains three or four case studies of non-trivial problems, most of which have been taken from textbooks on algorithm design, general programming, and combinatorial optimisation.

The chapters are intended to be read in sequence. Bibliographical remarks are included at the end of each chapter, and the majority of individual sections contain a selection of exercises. Answers to all the exercises in the first six chapters can be obtained from the World-wide Web: see the URL

`http://www.comlab.ox.ac.uk/oucl/publications/books/algebra/`

Acknowledgements

Many people have had a significant impact on the work, and detailed acknowledgements about who did what can be found at the end of each chapter. We owe a particular debt of gratitude to the following people, who took time to comment on an earlier draft, and to make many constructive suggestions: Roland Backhouse, Sharon Curtis, Jeremy Gibbons, Martin Henson, Tony Hoare, Guy LaPalme, Bernhard Möller, Jesus Ravelo, and Philip Wadler.

We would like to thank Jim Davies for knocking our LaTeX into shape, and Jackie Harbor, our editor at Prentice Hall, for enthusiasm, moral support, and a number of lunches.

The diagrams in this book were drawn using Paul Taylor's package (Taylor 1994).

Richard Bird would like to record a special debt of gratitude to Lambert Meertens for his friendship and collaboration over many years. Oege de Moor would like to thank the Dutch STOP project and British Petroleum for the financial assistance that enabled him to come and work in Oxford. The first part of this book was written at the University of Tokyo, while visiting Professors Masato Takeichi and Hidehiko Tanaka. Their hospitality and the generosity of Fujitsu, which made the visit possible, are gratefully acknowledged.

We would be pleased to hear of any errors, oversights and comments.

Richard Bird (bird@comlab.ox.ac.uk)
Oege de Moor (oege@comlab.ox.ac.uk)

April, 1996

'Now, then,' she said, somewhat calmer. 'An explanation, if you please, and a categorical one. What's the idea? What's it all about? Who the devil's that inside the winding-sheet?'

P.G. Wodehouse, *The Code of the Woosters*

Programs

Most of the derivations recorded in this book end with a program, more specifically, a functional program. In this opening chapter we settle notation for expressing functional programs and review those features of functional languages that will emerge again in a more general setting later on. Many aspects of modern functional languages (of which there is an abundance, e.g. Gofer, Haskell, Hope, Miranda™, Orwell, SML) are not covered. For example, we will not go into questions of strict versus non-strict semantics, infinite values, evaluation strategies, cost models, or operating environments. For fuller information we refer the reader to the standard texts on the subject, some of which are mentioned in the bibliographical remarks at the end of the chapter. Our main purpose is to identify familiar landmarks that will help readers to navigate through the abstract material to come.

1.1 Datatypes

At the heart of functional programming is the ability to introduce new datatypes and to define functions that manipulate their values. Datatypes can be introduced by simple enumeration of their elements; for example:

$$\begin{array}{lcl} \mathit{Bool} & ::= & \mathit{false} \mid \mathit{true} \\ \mathit{Char} & ::= & \mathit{ascii}0 \mid \mathit{ascii}1 \mid \cdots \mid \mathit{ascii}127. \end{array}$$

The type *Bool* consists of two values and *Char* consists of 128. It would be painful to refer to characters only by their ASCII numbers, so most languages provide an alternative syntax, allowing one to write 'A' for $\mathit{ascii}65$, 'a' for $\mathit{ascii}97$, '\n' for $\mathit{ascii}10$, and so on. The various identifiers, $\mathit{ascii}0$, *true*, and so on, are called *constructors* and the vertical bar $\mid$ is interpreted as the operation of disjoint union. Thus, distinct constructors are associated with distinct values.

Datatypes can be defined in terms of other datatypes; for example:

$$\mathit{Either} \;\; ::= \;\; \mathit{bool}\ \mathit{Bool} \mid \mathit{char}\ \mathit{Char}$$

$$Both \quad ::= \quad tuple\,(Bool, Char).$$

The type *Either* consists of 130 values: *bool false*, *bool true*, *char ascii*0, and so on. The type *Both* consists of 256 values, one for each combination of a value in *Bool* with a value in *Char*. In these datatypes the constructors *bool*, *char* and *tuple* denote functions; for example, *char* produces a value of type *Either* given a value of type *Char*.

As a departure from tradition, we write $f : A \leftarrow B$ rather than $f : B \rightarrow A$ to indicate the source and target types associated with a function f. Thus

$$\begin{aligned} &char : Either \leftarrow Char \\ &tuple : Both \leftarrow (Bool \times Char). \end{aligned}$$

The reason for this choice has to do with functional composition, whose definition now takes the smooth form: if $f : A \leftarrow B$ and $g : B \leftarrow C$, then $f \cdot g : A \leftarrow C$ is defined by $(f \cdot g)x = f(g\,x)$. Writing the target type on the left and the source type on the right is also consistent with the normal notation for application, in which functions are applied to arguments on the right. In the alternative, so-called *diagrammatic* forms, one writes $x\,f$ for application and $f;g$ for composition, where $x\,(f;g) = (x\,f)g$. The conventional order is consistent with adjectival order in English, in which adjectives are functions taking noun phrases to noun phrases.

Given the assurance about different constructors producing different values, we can define functions on datatypes by pattern matching; for example,

$$\begin{aligned} not\,false &= true \\ not\,true &= false \end{aligned}$$

defines the negation operator $not : Bool \leftarrow Bool$, and

$$switch\,(tuple\,(b, c)) \quad = \quad tuple\,(not\,b, c)$$

defines a function $switch : Both \leftarrow Both$.

Functions of more than one argument can be defined in one of two basic styles: either by pairing the arguments, as in

$$\begin{aligned} and\,(false, b) &= false \\ and\,(true, b) &= b \end{aligned}$$

or by currying, as in

$$\begin{aligned} cand\,false\,b &= false \\ cand\,true\,b &= b. \end{aligned}$$

The difference between *and* and *cand* is just one of type:

$$\begin{aligned} &\mathit{and} : \mathit{Bool} \leftarrow (\mathit{Bool} \times \mathit{Bool}) \\ &\mathit{cand} : (\mathit{Bool} \leftarrow \mathit{Bool}) \leftarrow \mathit{Bool}. \end{aligned}$$

More generally, we can define a function f of two arguments by choosing any of the types

$$\begin{aligned} &f : A \leftarrow (B \times C) \\ &f : (A \leftarrow B) \leftarrow C \\ &f : (A \leftarrow C) \leftarrow B. \end{aligned}$$

With the first type we would write $f\,(b, c)$; with the second, $f\; c\; b$; and with the third, $f\; b\; c$. For obvious reasons, the first and third seem more natural companions. The function *curry*, with type

$$\mathit{curry} : ((A \leftarrow C) \leftarrow B) \leftarrow (A \leftarrow (B \times C)),$$

converts a non-curried function into a curried one:

$$\mathit{curry}\, f\; b\; c \quad = \quad f\,(b, c).$$

One can also define a function *uncurry* that goes the other way.

Functional programmers prefer to curry their functions as a matter of course, one reason being that it usually leads to fewer brackets. However, we will be more sparing with currying, reserving its use for those situations that really need it. The reason is that the product type $A \times B$ is a simpler object than the function space type $C \leftarrow D$ in an abstract setting. We will see some examples of curried functions below, but functional programmers are warned at this point that some familiar functions will make their appearance in non-curried form.

To return to datatypes, we can parameterise datatypes with other types; for example, the definition

$$\mathit{maybe}\, A \quad ::= \quad \mathit{nothing} \mid \mathit{just}\, A$$

introduces a type *maybe* A in terms of a parameter type A. For example, *just true* has type *maybe Bool*, while *just ascii*0 has type *maybe Char*. We will write non-parameterised datatypes using a capital initial letter, and parameterised datatypes using lower case letters only. The reason, as we shall explain later on, is that the name of a parameterised datatype will also be used for a certain function associated with the datatype, and we write the names of functions using lower case letters.

1.2 Natural numbers

Datatypes can also be defined recursively; for example,

$$Nat \;::=\; zero \mid succ\ Nat$$

introduces the type of natural numbers. *Nat* is the union of an infinite number of distinct values: *zero*, *succ zero*, *succ* (*succ zero*), and so on. If two distinct expressions of *Nat* denoted the same value, we could show, for some element n of *Nat*, that both *zero* and *succ* n denoted the same value, contradicting the basic assumption that different constructors produce different values.

Functions over *Nat* can be defined by recursion; for example,

$$\begin{aligned} plus\,(m, zero) &= m \\ plus\,(m, succ\,n) &= succ\,(plus\,(m, n)) \end{aligned}$$

and

$$\begin{aligned} mult\,(m, zero) &= zero \\ mult\,(m, succ\,n) &= plus\,(m, mult\,(m, n)). \end{aligned}$$

Forcing the programmer to write *succ* (*succ* (*succ zero*)) instead of 3, and to re-create all of arithmetic from scratch, would be a curious decision, to say the least, by the designers of a programming language, so a standard syntax for numbers is usually provided, as well as the basic arithmetic operations. In particular, *zero* is written 0 and *succ* n is written $n + 1$. With these conventions, we can write definitions in a more perspicuous form; for example,

$$\begin{aligned} fact\,0 &= 1 \\ fact\,(n+1) &= (n+1) \times fact\,n \end{aligned}$$

defines the factorial function, and

$$\begin{aligned} fib\,0 &= 0 \\ fib\,1 &= 1 \\ fib\,(n+2) &= fib\,n + fib\,(n+1) \end{aligned}$$

defines the Fibonacci function. The expression $n + 2$ corresponds to the pattern *succ* (*succ* n), which is disjoint from the patterns *zero* and *succ zero*.

Some systems of recursive equations do not define functions; for example,

$$f\,n = f\,(n+1).$$

Every constant function satisfies the equation for f, but none is defined by it. On the other hand, the two equations

$$\begin{aligned} f\,0 &= c \\ f\,(n+1) &= h\,(f\,n) \end{aligned}$$

do define a unique function f for every constant c and function h of appropriate types. More precisely, if c has type A for some A, and if h has type $h : A \leftarrow A$, then f is defined uniquely for every natural number and has type $f : A \leftarrow \mathit{Nat}$. The above scheme is called definition by *structural* recursion over the natural numbers, and is an instance of a slightly more general scheme called *primitive* recursion. Much of this book is devoted to understanding and exploiting the idea of defining a function (or, more generally, a relation) by structural recursion over a datatype.

The two equations given above can be captured in terms of a single function *foldn* that takes the constant c and function h as arguments; thus $f = \mathit{foldn}\,(c, h)$. The function *foldn* is called the *fold* operator for the type *Nat*. Observe that $\mathit{foldn}\,(c, h)$ works by taking a natural number expressed in terms of *zero* and *succ*, replacing *zero* by c and *succ* by h, and then evaluating the result. In other words, $\mathit{foldn}\,(c, h)$ describes a *homomorphism* of *Nat*.

It is a fact that not every computable function over the natural numbers can be described using structural recursion, so certainly some functional programs are inaccessible if only structural recursion is allowed. However, in the presence of currying and other bits and pieces, structural recursion is both a flexible and powerful tool (see Exercise 1.6). For example,

$$\begin{aligned} \mathit{plus}\ m &= \mathit{foldn}\,(m, \mathit{succ}) \\ \mathit{mult}\ m &= \mathit{foldn}\,(0, \mathit{plus}\ m) \\ \mathit{expn}\ m &= \mathit{foldn}\,(1, \mathit{mult}\ m) \end{aligned}$$

define curried versions of addition, multiplication and exponentiation. In these definitions currying plays an essential role since *foldn* gives us no way of defining recursive functions on pairs of numbers.

As two more examples, the factorial function can be computed by

$$\begin{aligned} \mathit{fact} &= \mathit{outr} \cdot \mathit{foldn}\,((0,1), f) \\ \mathit{outr}\,(m, n) &= n \\ f\,(m, n) &= (m+1, (m+1) \times n), \end{aligned}$$

and the Fibonacci function can be computed by

$$\begin{aligned} \mathit{fib} &= \mathit{outl} \cdot \mathit{foldn}\,((0,1), f) \\ \mathit{outl}\,(m, n) &= m \\ f\,(m, n) &= (n, m+n). \end{aligned}$$

The two functions *outl* (short for 'out-left') and *outr* ('out-right') are *projection* functions that select the left and right elements of a pair of values. These programs for *fact* and *fib* can be regarded as implementations of the recursive definitions. The program for *fib* has the advantage that values of *fib* are computed in linear time, while the recursive definition, if implemented directly, would require exponential time. The program for *fib* illustrates an important idea, called *tabulation*, in which function values are stored for subsequent use rather than being calculated afresh each time. Here, the table is very simple, consisting of just a pair of values: $foldn\,((0,1), f)\,n$ returns the pair $(fib\,n, fib\,(n+1))$. The theme of tabulation will emerge again in Chapter 9 on dynamic programming.

Further examples of recursive datatypes appear in subsequent sections.

f n = plus 1 (f n)

Exercises

1.1 Give an example of a recursion equation that is not satisfied by any function.

1.2 Consider the recursion equation

x < y

$$\begin{array}{rcll} m\,(x,y) & = & y+1, & \text{if } x = y \\ & = & m(x, m(x-1, y+1)), & \text{otherwise.} \end{array}$$

Does this determine a unique function m?

1.3 Construct a datatype Nat^+ for representing the integers > 0, together with an operator $foldn^+$ for iterating over such numbers. Give functions $f : Nat^+ \leftarrow Nat$ and $g : Nat \leftarrow Nat^+$ such that $f \cdot g$ is the identity function on Nat^+ and $g \cdot f$ is the identity function on Nat.

1.4 Express the squaring function $sqr : Nat \leftarrow Nat$ in the form $sqr = f \cdot foldn\,(c, h)$ for suitable f, c and h.

1.5 Consider the function $last\,p : Nat \leftarrow Nat$ such that $last\,p\,n$ returns the largest natural number $m \leq n$ satisfying $p : Bool \leftarrow Nat$. Assuming that $p\,0$ holds, construct suitable f, c and h so that $last\,p = f \cdot foldn\,(c, h)$.

1.6 Ackermann's function $ack : Nat \leftarrow Nat \times Nat$ is defined by the equations

C = (0, 0)

h (X, Y) = (X+1, ite(P, x, y))

f = outr

$$\begin{array}{rcl} ack(0, y) & = & y+1 \\ ack(x+1, 0) & = & ack(x, 1) \\ ack(x+1, y+1) & = & ack(x, ack(x+1, y)). \end{array}$$

The function $curry\,ack$ can be expressed as $foldn(succ, f)$ for an appropriate f. What is f? (Remark: this shows that, in the presence of currying, functions which are not *primitive recursive* can be expressed in terms of $foldn$.)

f = foldn (g 1, f)

1.3 Lists

The datatype of lists dominates functional programming; much of the subject is taken up with notation for lists, and the names and properties of useful functions for manipulating them. The Appendix contains a summary of the more important list-processing functions, together with other combinators we will use throughout the book.

There are two basic views of lists, given by the type declarations

$$\begin{aligned} \mathit{listr}\,A &::= \mathit{nil} \mid \mathit{cons}\,(A, \mathit{listr}\,A) \\ \mathit{listl}\,A &::= \mathit{nil} \mid \mathit{snoc}\,(\mathit{listl}\,A, A). \end{aligned}$$

The former describes the type of cons-lists, in which elements are added to the front of a list; the latter describes the type of snoc-lists, in which elements are added to the rear. Thus *listr* builds lists from the right, while *listl* builds lists from the left. The constructor *nil* is overloaded in that it denotes both the empty cons-list and the empty snoc-list; in any program making use of both forms of list, distinct names would have to be chosen.

The two types of list are different, though isomorphic to one another. For example, the function $\mathit{convert} : \mathit{listr}\,A \leftarrow \mathit{listl}\,A$ that converts a snoc-list into a cons-list can be defined recursively by

$$\begin{aligned} \mathit{convert}\;\mathit{nil} &= \mathit{nil} \\ \mathit{convert}\,(\mathit{snoc}\,(x, a)) &= \mathit{snocr}\,(\mathit{convert}\,x, a) \\ \mathit{snocr}\,(\mathit{nil}, b) &= \mathit{cons}\,(b, \mathit{nil}) \\ \mathit{snocr}\,(\mathit{cons}\,(a, x), b) &= \mathit{cons}\,(a, \mathit{snocr}\,(x, b)). \end{aligned}$$

match l with:
| nil —> nil
| (l.t)

The function $\mathit{snocr} : \mathit{listr}\,A \leftarrow (\mathit{listr}\,A \times A)$ appends an element to the end of a cons-list. This function takes $O(n)$ steps on a list of length n, so *convert* takes $O(n^2)$ steps to convert a list of length n. The number of steps can be brought down to $O(n)$ using a technique known as an *accumulation parameter* (see the exercises).

It is inconvenient to have to manipulate two versions of what is essentially the same datatype, so functional languages have traditionally given privileged status to just one of them. (The alternative, explored in (Wadler 1987), is to regard both types as different views of one and the same type, and to create a mechanism for moving from one view to the other, quietly and efficiently, as occasion demands.) Cons-lists are taken as the basic view, and special syntax is provided for *nil* and *cons*. The empty list is written $[\,]$, and $\mathit{cons}\,(a, x)$ is written $a : x$. In addition, $[a]$ can be used for $a : [\,]$, $[a, b]$ for $a : b : [\,]$, and so on. However, since we want to treat both types of list on an equal footing, we will not use the syntax $a : x$; for now we stick with the slightly cumbersome forms *nil*, $\mathit{cons}\,(a, x)$ and $\mathit{snoc}\,(x, a)$.

The concatenation of two lists x and y is denoted by $x \mathbin{+\!\!+} y$. For example,

$$[1,2,3] \mathbin{+\!\!+} [4,5] \quad = \quad [1,2,3,4,5].$$

In particular,

$$\begin{aligned} \mathit{cons}\,(a,x) &= [a] \mathbin{+\!\!+} x \\ \mathit{snoc}\,(x,a) &= x \mathbin{+\!\!+} [a]. \end{aligned}$$

Later on, but not just yet, we will use the expressions on the right as alternatives for those on the left; this is an extension of a similar convention for *Nat*, in which we wrote $n+1$ for $\mathit{succ}\, n$, thereby harnessing the operation $+$ for a more primitive purpose.

The type and definition of concatenation depends on the version of lists under consideration. For example, taking

$$(\mathbin{+\!\!+}) : \mathit{listl}\, A \leftarrow \mathit{listl}\, A \times \mathit{listl}\, A,$$

so that $x \mathbin{+\!\!+} y$ abbreviates $\mathbin{+\!\!+}(x,y)$, we can define $\mathbin{+\!\!+}$ by

$$\begin{aligned} x \mathbin{+\!\!+} \mathit{nil} &= x \\ x \mathbin{+\!\!+} \mathit{snoc}\,(y,a) &= \mathit{snoc}\,(x \mathbin{+\!\!+} y, a). \end{aligned}$$

Using this definition, we can show that $\mathbin{+\!\!+}$ is an associative operation, and that *nil* is a left unit as well as a right one. The proof that

$$x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z) \quad = \quad (x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z$$

proceeds by induction on z. The base case is

$$\begin{aligned} & x \mathbin{+\!\!+} (y \mathbin{+\!\!+} \mathit{nil}) \\ = \quad & \quad \{\text{first equation defining } \mathbin{+\!\!+}\} \\ & x \mathbin{+\!\!+} y \\ = \quad & \quad \{\text{first equation defining } \mathbin{+\!\!+}\} \\ & (x \mathbin{+\!\!+} y) \mathbin{+\!\!+} \mathit{nil}. \end{aligned}$$

The induction step is

$$\begin{aligned} & x \mathbin{+\!\!+} (y \mathbin{+\!\!+} \mathit{snoc}\,(z,a)) \\ = \quad & \quad \{\text{second equation defining } \mathbin{+\!\!+}\} \\ & x \mathbin{+\!\!+} \mathit{snoc}\,(y \mathbin{+\!\!+} z, a) \\ = \quad & \quad \{\text{second equation defining } \mathbin{+\!\!+}\} \\ & \mathit{snoc}\,(x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z), a) \end{aligned}$$

$$
\begin{array}{ll}
= & \{\text{induction hypothesis}\} \\
& \mathit{snoc}\,((x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z, a) \\
= & \{\text{second equation defining } \mathbin{+\!\!+} \text{ (backwards)}\} \\
& (x \mathbin{+\!\!+} y) \mathbin{+\!\!+} \mathit{snoc}\,(z, a).
\end{array}
$$

We leave the proof that *nil* is the unit of $\mathbin{+\!\!+}$ as an exercise. The above style and format for calculations will be adopted throughout the book.

A most useful operation on lists is the function that applies a function to every element of a list. Traditionally, this operation is called $\mathit{map}\, f$. If $f : A \leftarrow B$, then $\mathit{map}\, f : \mathit{list}\, A \leftarrow \mathit{list}\, B$ is defined informally by

$$
\mathit{map}\, f\, [a_1, a_2, \ldots, a_n] \quad = \quad [f\, a_1, f\, a_2, \ldots, f\, a_n].
$$

We will not, however, use the traditional name, preferring to use $\mathit{listr}\, f$ for the map operation on cons-lists, and $\mathit{listl}\, f$ for the same operation on snoc-lists. Thus the name of the type plays a dual role, signifying the type in type expressions, and the map operation in value expressions. The same convention is extended to other parameterised types. The reason for this choice will emerge in the next chapter.

The function $\mathit{listr}\, f$ can be defined recursively:

$$
\begin{array}{rcl}
\mathit{listr}\, f\, \mathit{nil} & = & \mathit{nil} \\
\mathit{listr}\, f\, (\mathit{cons}\, (a, x)) & = & \mathit{cons}\, (f\, a, \mathit{listr}\, f\, x).
\end{array}
$$

There is a similar definition for $\mathit{listl}\, f$. Instead of writing down recursion equations we can appeal to a standard recipe similar to that introduced for *Nat*. Consider the scheme

$$
\begin{array}{rcl}
f\, \mathit{nil} & = & c \\
f\, (\mathit{cons}\, (a, x)) & = & h\, (a, f\, x)
\end{array}
$$

for defining a recursive function f with source type $\mathit{listr}\, A$ for some A. We encapsulate this pattern of recursion by a function *foldr*, so that $f = \mathit{foldr}\, (c, h)$. In other words,

$$
\begin{array}{rcl}
\mathit{foldr}\, (c, h)\, \mathit{nil} & = & c \\
\mathit{foldr}\, (c, h)\, (\mathit{cons}\, (a, x)) & = & h(a, \mathit{foldr}\, (c, h)\, x).
\end{array}
$$

Given $h : B \leftarrow A \times B$ and $c : B$, we have $\mathit{foldr}\, (c, h) : B \leftarrow \mathit{listr}\, A$. In particular,

$$
\mathit{listr}\, f \quad = \quad \mathit{foldr}\, (\mathit{nil}, h) \qquad \text{where } h(a, x) = \mathit{cons}\, (f\, a, x).
$$

In a similar spirit, we define

$$\begin{aligned} \mathit{foldl}\,(c,h)\,\mathit{nil} &= c \\ \mathit{foldl}\,(c,h)\,(\mathit{snoc}\,(x,a)) &= h(\mathit{foldl}\,(c,h)\,x,a), \end{aligned}$$

so that $\mathit{foldl}\,(c,h) : B \leftarrow \mathit{listl}\,A$ provided $h : B \leftarrow B \times A$ and $c : B$. Now we have

$$\mathit{listl}\,f \quad = \quad \mathit{foldl}\,(\mathit{nil},h) \qquad \text{where } h(x,a) = \mathit{snoc}\,(x, fa).$$

The functions $\mathit{foldr}\,(c,h)$ and $\mathit{foldl}\,(c,h)$ work in a similar fashion to $\mathit{foldn}\,(c,h)$ of the preceding section: $\mathit{foldr}\,(c,h)$ transforms a list by systematically replacing *nil* by c and *cons* by h; similarly, $\mathit{foldl}\,(c,h)$ replaces *nil* by c and *snoc* by h. Like *foldn* on the natural numbers, these two functions embody structural recursion on their respective datatypes and can be used to define many useful functions. For example, on snoc-lists we can define a curried version of concatenation by

$$\mathit{cat}\,x \quad = \quad \mathit{foldl}\,(x, \mathit{snoc}).$$

We have $\mathit{cat}\,x\,y = x \mathbin{+\!\!+} y$. This definition mirrors the earlier definition of addition: $\mathit{plus}\,m = \mathit{foldn}\,(m, \mathit{succ})$. We leave it as an exercise to define a version of *cat* over cons-lists.

Other examples on cons-lists include

$$\begin{aligned} \mathit{sum} &= \mathit{foldr}\,(0, \mathit{plus}) \\ \mathit{product} &= \mathit{foldr}\,(1, \mathit{mult}) \\ \mathit{concat} &= \mathit{foldr}\,(\mathit{nil}, \mathit{cat}) \\ \mathit{length} &= \mathit{sum} \cdot \mathit{listr}\,\mathit{one}, \qquad \text{where } \mathit{one}\,a = 1. \end{aligned}$$

The function $\mathit{concat} : \mathit{listr}\,A \leftarrow \mathit{listr}\,(\mathit{listr}\,A)$ concatenates a list of lists into one long list, and *length* returns the length of a list. The length function can also be defined in terms of a single *foldr*:

$$\mathit{length} \quad = \quad \mathit{foldr}\,(0,h), \qquad \text{where } h(a,n) = n+1.$$

This is an example of a general phenomenon: any function which can be expressed as a fold after a mapping operation can also be expressed as a single fold. We will state and prove a suitably general version of this result in the next chapter.

Another example is provided by the function $\mathit{filter}\,p : \mathit{listr}\,A \leftarrow \mathit{listr}\,A$, where p has type $p : \mathit{Bool} \leftarrow A$. This function filters a list, retaining only those elements that satisfy p. It can be defined as follows:

$$\begin{aligned} \mathit{filter}\,p &= \mathit{concat} \cdot \mathit{listr}\,(p \rightarrow \mathit{wrap}, \mathit{nilp}) \\ (p \rightarrow f, g)\,a &= \begin{cases} f\,a, & \text{if } p\,a \\ g\,a, & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} \mathit{wrap}\, a &= \mathit{cons}\,(a, \mathit{nil}) \\ \mathit{nilp}\, a &= \mathit{nil}. \end{aligned}$$

The McCarthy conditional form $(p \to f, g)$ is used to describe conditional expressions, *wrap* turns its argument into a singleton list, and *nilp* is a constant function that returns the empty list for each argument. The function *filter* p works by turning an element that satisfies p into a singleton list, and an element that doesn't satisfy p into the empty list, and concatenating the results.

We can express *filter* as a single *foldr*:

$$\mathit{filter}\, p \;=\; \mathit{foldr}\,(\mathit{nil}, (p \cdot \mathit{outl} \to \mathit{cons}, \mathit{outr})).$$

The projection functions *outl* and *outr* were introduced earlier. Applied to (a, x), the function $(p \cdot \mathit{outl} \to \mathit{cons}, \mathit{outr})$ returns $\mathit{cons}\,(a, x)$ if $p\ a$ is true, and x otherwise. Yet another way to express *filter* is given in the last section of this chapter.

Finally, let us consider an example where the difference between cons-lists and snoc-lists plays an essential role. Consider the problem of converting some suitable representation of a decimal value into the real number it represents. Suppose the number is

$$d_m d_{m-1} \ldots d_0 . e_1 e_2 \ldots e_n,$$

which represents the number $w + f$, where

$$\begin{aligned} w &= 10^m d_m + 10^{m-1} d_{m-1} + \cdots 10^0 d_0 \\ f &= e_1/10^1 + e_2/10^2 + \cdots e_n/10^n. \end{aligned}$$

Observing that

$$\begin{aligned} w &= 10 \times ((\ldots(10 \times (10 \times 0 + d_m) + d_{m-1}) \ldots)) + d_0 \\ f &= (e_1 + \cdots (e_{n-1} + e_n/10)/10 \cdots)/10, \end{aligned}$$

we can see that one sensible way to represent decimal numbers is by a pair of lists *listl Digit* $\times$ *listr Digit*. We can then define the evaluating function *eval* by

$$\begin{aligned} \mathit{eval} &: \mathit{Real} \leftarrow (\mathit{listl\ Digit} \times \mathit{listr\ Digit}) \\ \mathit{eval}\,(x, y) &= \mathit{foldl}\,(0, f)\, x + \mathit{foldr}\,(0, g)\, y \\ f\,(n, d) &= 10 \times n + d \\ g\,(e, r) &= (e + r)/10. \end{aligned}$$

It is appropriate to represent the whole number part by a snoc-list because it is evaluated more conveniently by processing the digits from left to right; on the other hand, the fractional part is more appropriately represented by a cons-list since the processing is from right to left.

Lists in functional programming

In traditional functional programming, the functions $\mathit{foldr}\,(c, h)$ and $\mathit{foldl}\,(c, h)$ are defined a little differently. There are two minor differences and one major one. First, *foldr* and *foldl* are usually defined as curried functions, writing $\mathit{foldr}\; h\; c$ instead of $\mathit{foldr}\,(c, h)$, and similarly for *foldl*. One small advantage of the switch of arguments is that some functions can be defined more succinctly; for example, the curried function *cat* on snoc-lists can now be defined by

$$\mathit{cat} \;=\; \mathit{foldl}\; \mathit{snoc}.$$

The second minor difference is that the argument h in $\mathit{foldr}\; h\; c$ is also curried because *cons* is usually introduced as a curried function. Since we have introduced *cons* to have type $\mathit{listr}\, A \leftarrow (A \times \mathit{listr}\, A)$, it is appropriate to take the type of h to be $B \leftarrow (A \times B)$.

The more important difference is that in traditional functional programming the basic view of lists is cons-lists and, because *foldl* is a useful operation to have, the type assigned to $\mathit{foldl}\,(c, h)$ is $B \leftarrow \mathit{listr}\, A$, for some A and B. This means that *foldl* is given a different definition, namely,

$$\begin{aligned} \mathit{foldl}\,(c, h)\; \mathit{nil} &= c \\ \mathit{foldl}\,(c, h)\,(\mathit{cons}\,(a, x)) &= \mathit{foldl}\,(h\,(c, a), h)\; x. \end{aligned}$$

This is essentially an *iterative* definition and corresponds to a loop in imperative programming. The first component of the first argument of *foldl* is treated as an accumulation parameter, and models the state of an imperative program. We leave it as an exercise to show that the two definitions are equivalent, and to discover a way of expressing this version of *foldl* in terms of *foldr*.

This definition of *foldl* as an operation on cons-lists can be used to good effect. Consider, for example, the function *reverse* that reverses the elements of a list. As a function on cons-lists, we can define

$$\begin{aligned} \mathit{reverse} &= \mathit{foldr}\,(\mathit{nil}, \mathit{append}) \\ \mathit{append}\,(a, x) &= \mathit{snocr}\,(x, a), \end{aligned}$$

where *snocr* was defined earlier. As a function on snoc-lists, we can define

$$\begin{aligned} \mathit{reverse} &= \mathit{foldl}\,(\mathit{nil}, \mathit{prepend}) \\ \mathit{prepend}\,(x, a) &= \mathit{cons}\,(a, x). \end{aligned}$$

As an implementation of *reverse* on cons-lists, the first definition takes $O(n^2)$ steps to reverse a list of length n, the reason being that *snocr* requires linear time. However, interpreting *foldl* as an operation on cons-lists, the second definition of *reverse* takes linear time because *cons* takes constant time.

Non-empty lists

Having the empty list around sometimes causes more trouble than it is worth. Fortunately, we can always introduce the types

$$\begin{aligned} \mathit{listr}^+ A &::= \mathit{wrap}\, A \mid \mathit{cons}\,(A, \mathit{listr}^+ A) \\ \mathit{listl}^+ A &::= \mathit{wrap}\, A \mid \mathit{snoc}\,(\mathit{listl}^+ A, A) \end{aligned}$$

of non-empty cons-lists and snoc-lists. Here, *wrap* returns a singleton list and the generic fold operation replaces the function *wrap* by a function f and *cons* by a function g:

$$\begin{aligned} \mathit{foldr}^+ (f, g)\,(\mathit{wrap}\, a) &= f\, a \\ \mathit{foldr}^+ (f, g)\,(\mathit{cons}\,(a, x)) &= g\,(a, \mathit{foldr}^+ (f, g)\, x). \end{aligned}$$

In particular, the function $\mathit{head} : A \leftarrow \mathit{listr}^+ A$ that returns the first element of a non-empty list can be defined by

$$\mathit{head} = \mathit{foldr}^+ (\mathit{id}, \mathit{outl}).$$

In some functional languages the fold operator on non-empty cons-lists is denoted by *foldr*1, with the definition

$$\mathit{foldr}1\, f = \mathit{foldr}^+ (\mathit{id}, f).$$

So *foldr*1 cannot express the general fold operator on non-empty cons-lists, but only the special case (admittedly the most frequent in practice) in which the first argument is the identity function.

List comprehensions

Finally, we introduce a useful piece of syntax that can be used as an alternative to many expressions involving *listr* and *filter*. An expression of the form

$$[\mathit{expr}0 \mid \mathit{var} \leftarrow \mathit{expr}1; \mathit{expr}2]$$

is called a *list comprehension* and produces a list of values of the form *expr*0 for values *var* drawn from the list expression *expr*1 and satisfying the boolean expression *expr*2. For example,

$$[n \times n \mid n \leftarrow [1\,..\,10]; \mathit{even}\, n]$$

produces, in order, the list of squares of even numbers n in the range $1 \leq n \leq 10$. In particular, we have

$$\begin{aligned} \mathit{listr}\, f\, x &= [f\, a \mid a \leftarrow x] \\ \mathit{filter}\, p\, x &= [a \mid a \leftarrow x; p\, a]. \end{aligned}$$

There is a more general form of list comprehension, but we will not need it; indeed, list comprehensions are used only occasionally in what follows.

Exercises

1.7 Construct the function $convert : listr\, A \leftarrow listl\, A$ in the form $foldl\,(c, h)$ for suitable c and h.

1.8 Consider the curried function $catconv : (listr\, A \leftarrow listl\, A) \leftarrow listr\, A$ defined by $catconv\; x\; y = convert\; x \mathbin{+\!\!+} y$. Express $catconv$ in the form $foldl\,(c, h)$ and hence show how $convert$ can be carried out in linear time.

1.9 Prove that nil is a left unit of $\mathbin{+\!\!+}$.

1.10 Construct $cat : (listr\, A \leftarrow listr\, A) \leftarrow listr\, A$.

1.11 Construct the iterative function $foldl$ over cons-lists in terms of $foldr$.

1.12 The function $take\; n : listr\, A \leftarrow listr\, A$ takes the first n items of a list, or the whole list if its length is no larger than n. Construct suitable h and c for which $take\; n\; x = foldr\,(c, h)\; x\; n$. Similarly, define the function $drop\; n$ (which drops the first n items from a list) in terms of $foldr$.

1.4 Trees

We will briefly consider two more examples of recursive datatypes to drive home the points made in preceding sections. First, consider the type

$$tree\; A \quad ::= \quad tip\; A \mid bin\,(tree\; A, tree\; A)$$

of binary trees with elements from A in the tips. In particular, the expression

$$bin\,(tip\; 0, bin\,(tip\; 1, tip\; 2))$$

denotes an element of $tree\; Nat$, while

$$bin\,(tip\; \text{`A'}, bin\,(tip\; \text{`B'}, tip\; \text{`C'}))$$

denotes an element of $tree\; Char$.

The generic form of the fold operator for binary trees is $foldt\,(f, g)$, defined by

$$\begin{aligned} foldt\,(f, g)\,(tip\; a) &= f\; a \\ foldt\,(f, g)\,(bin\,(x, y)) &= g\,(foldt\,(f, g)\; x, foldt\,(f, g)\; y). \end{aligned}$$

Here, $\mathit{foldt}\,(f, g) : B \leftarrow \mathit{tree}\, A$ if $f : B \leftarrow A$ and $g : B \leftarrow B \times B$. In particular, the map function for trees is given by

$$\mathit{tree}\, f \quad = \quad \mathit{foldt}\,(\mathit{tip} \cdot f, \mathit{bin}).$$

The functions *size* and *depth* for determining the size and the depth of a tree are given by

$$\begin{aligned} \mathit{size} &= \mathit{foldt}\,(\mathit{one}, \mathit{plus}), \quad \text{where } \mathit{one}\; a = 1 \\ \mathit{depth} &= \mathit{foldt}\,(\mathit{zero}, \mathit{succ} \cdot \mathit{bmax}), \quad \text{where } \mathit{zero}\; a = 0. \end{aligned}$$

Here, $\mathit{bmax}\,(x, y)$ (short for 'binary maximum') returns the greater of x and y; the depth of the tree $\mathit{bin}\,(x, y)$ is one more than the greater of the depths of trees x and y.

The final example is of two mutually recursive datatypes. Consider the types

$$\begin{aligned} \mathit{tree}\, A &::= \mathit{fork}\,(A, \mathit{forest}\, A) \\ \mathit{forest}\, A &::= \mathit{null} \mid \mathit{grow}\,(\mathit{tree}\, A, \mathit{forest}\, A), \end{aligned}$$

defining trees and forests in terms of each other. The type $\mathit{forest}\, A$ is in fact isomorphic to $\mathit{listr}\,(\mathit{tree}\, A)$, so we could also have introduced trees using lists rather than forests.

The generic fold operation for this kind of tree is not defined by a single recursion, but as the first of a pair of functions, $\mathit{foldt}\,(g, c, h)$ and $\mathit{foldf}\,(g, c, h)$, defined simultaneously by mutual recursion:

$$\begin{aligned} \mathit{foldt}\,(g, c, h)\,(\mathit{fork}\,(a, \mathit{xs})) &= g\,(a, \mathit{foldf}\,(g, c, h)\, \mathit{xs}) \\ \mathit{foldf}\,(g, c, h)\, \mathit{null} &= c \\ \mathit{foldf}\,(g, c, h)\,(\mathit{grow}\,(x, \mathit{xs})) &= h\,(\mathit{foldt}\,(g, c, h)\, x, \mathit{foldf}\,(g, c, h)\, \mathit{xs}). \end{aligned}$$

For example, the size of a tree is defined by

$$\mathit{size} \quad = \quad \mathit{foldt}\,(\mathit{succ} \cdot \mathit{outr}, 0, \mathit{plus}).$$

We have now seen enough examples to get the general idea: when introducing a new datatype, also define the generic fold operation for that datatype. When the datatype is parameterised, also introduce the appropriate mapping operation. Given these functions, a number of other useful functions can be quickly defined.

It would be nice if we could give, once and for all, a single completely generic definition of the fold operator, parameterised by the structure of the datatype being defined. Indeed, we shall do just this in the next chapter. But in most functional languages currently available, this is not possible: we can parameterise functions with abstract operators, but we cannot parameterise functions with abstract datatypes.

Recently, several authors have proposed new languages that overcome this restriction, and some references can be found in the bibliographical remarks at the end of this chapter.

Exercises

1.13 Consider the type

$$gtree\ A \quad ::= \quad node\,(A, listl\,(gtree\ A))$$

of general trees with nodes labelled with elements from A. Define the generic *foldg* function for this kind of tree, and hence construct functions *size* and *depth* for computing the size and depth of a tree.

1.14 Continuing on from the preceding exercise, represent the expression

$$f\,(g\,(a, b), h\,(c), d)$$

as an element of *gtree Char*. Convert this expression to curried form, and represent the result as an element of *tree Char*. Using this translation as a guide, construct functions

$$\begin{aligned} curry &: tree\ A \leftarrow gtree\ A \\ uncurry &: gtree\ A \leftarrow tree\ A \end{aligned}$$

for converting from general trees to binary trees and vice-versa.

1.5 Inverses

Another theme that will emerge in subsequent chapters is the use of inverses in program specification and synthesis. Some functions are best specified as inverses to other functions. Consider, for example, the function *zip* with type

$$zip : listr\,(A \times B) \leftarrow (listr\ A \times listr\ B),$$

which is defined informally by

$$zip\,([a_1, a_2, \ldots, a_n], [b_1, b_2, \ldots, b_n]) \quad = \quad [(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)].$$

One way of specifying *zip* is as the inverse of a function

$$unzip : listr\ A \times listr\ B \leftarrow listr\,(A \times B),$$

defined by

$$unzip \quad = \quad pair\,(listr\ outl, listr\ outr),$$

where $pair\,(f,g)\,x = (f\,x, g\,x)$. Thus, *unzip* takes a list of pairs and returns a pair of lists consisting of the first components (*listr outl*) and the second components (*listr outr*). The function *unzip* can also be expressed in terms of a single fold:

$$\begin{aligned} unzip &= foldr\,(nils, conss) \\ nils &= (nil, nil) \\ conss\,((a,b),(x,y)) &= (cons\,(a,x), cons\,(b,y)). \end{aligned}$$

This is another example of a general result that we will give later in the book: a pair of folds can always be expressed as a single fold.

Now, *unzip* is an injective function, which means that we can specify *zip* by the condition

$$zip \cdot unzip = id.$$

Using this specification, we can synthesise a direct recursive definition of *zip*:

$$\begin{aligned} zip\,(nil, nil) &= nil \\ zip\,(cons\,(a,x), cons\,(b,y)) &= cons\,((a,b), zip\,(x,y)). \end{aligned}$$

Note that *zip* is a partial function, defined only on lists of equal length. This is because *unzip* is not surjective. In functional programming, *zip* is made total by extending its recursive definition to read

$$\begin{aligned} zip\,(nil, y) &= nil \\ zip\,(cons\,(a,x), nil) &= nil \\ zip\,(cons\,(a,x), cons\,(b,y)) &= cons\,((a,b), zip\,(x,y)). \end{aligned}$$

This version of *zip* works for two lists of different lengths, stopping when either list is exhausted.

As another example, consider the function $decimal : listl\ Digit \leftarrow Nat$ that converts a natural number to the decimal numeral that represents it. The inverse function in this case is $eval : Nat \leftarrow listl\ Digit$ defined by

$$\begin{aligned} eval &= foldl\,(0,f) \\ f\,(n,d) &= 10 \times n + d. \end{aligned}$$

However, *eval* is not an injective function, so we cannot specify *decimal* simply by the equation $decimal \cdot eval = id$. There are two ways out of this problem: either we can define a type *Decimal*, replacing *listl Digit*, so that *eval* is an injective function on *Decimal*; or else specify $decimal\,n$ to be, say, a *shortest* member of the set $\{x \mid eval\,x = n\}$. Both methods will be examined in due course, so we will not go into details at this stage. The main point we want to make here is that definition by inverse is a useful method of specification, but one that involves difficulties when working exclusively with functions. The solution, as we shall see, is to move to relations: all relations possess a unique converse, so there is no problem in specifying one relation as the converse of another. If we want to specify a function in this way, then we have to find some functional *refinement* of the converse. We shall also study methods for doing just this.

Exercises

1.15 Construct the curried version $\mathit{zip} : (\mathit{listr}\,(A \times B) \leftarrow \mathit{listr}\,B) \leftarrow \mathit{listr}\,A$ in the form $\mathit{foldr}\,(c, h)$ for suitable h and c.

1.16 Define a datatype *Digits* that represents non-empty lists of digits, not beginning with zero. Define the generic *fold* function for *Digits*, and use it to construct the evaluating function $\mathit{eval} : \mathit{Nat}^+ \leftarrow \mathit{Digits}$, where Nat^+ is the type of positive integers. Can you specify $\mathit{decimal} : \mathit{Digits} \leftarrow \mathit{Nat}^+$ as the inverse of *eval*?

1.6 Polymorphic functions

Some of the list-processing functions defined above are polymorphic in that they do not depend in any essential way on the particular type of lists being considered. For example, $\mathit{concat} : \mathit{listr}\,A \leftarrow \mathit{listr}\,(\mathit{listr}\,A)$ does not depend in any essential way on the type A. Such functions satisfy certain identities appropriate to their type. For example, we have

$$\mathit{listr}\,f \cdot \mathit{concat} \quad = \quad \mathit{concat} \cdot \mathit{listr}\,(\mathit{listr}\,f).$$

This equation can be interpreted as the assertion that the recipe of concatenating a list of lists, and then renaming the elements, has the same outcome as renaming each element in the list of lists, and then concatenating. Thus, *concat* does not depend on the structure of the elements of the lists being concatenated. A formal proof of the equation above is left as an exercise.

As another example, consider the function $\mathit{inits} : \mathit{listl}\,(\mathit{listl}\,A) \leftarrow \mathit{listl}\,A$ that returns a list of all prefixes of a list:

$$\begin{aligned} \mathit{inits} &= \mathit{foldl}\,([\mathit{nil}], f) \\ f\,(\mathit{snoc}\,(xs, x), a) &= \mathit{snoc}\,(\mathit{snoc}\,(xs, x), \mathit{snoc}\,(x, a)). \end{aligned}$$

For example,

$$\mathit{inits}\,[a_1, a_2, a_3] \quad = \quad [[\,], [a_1], [a_1, a_2], [a_1, a_2, a_3]].$$

Like *concat*, the function *inits* does not depend in any essential way on the nature of elements in the list; the result is the same whether we take the prefixes and then process each element in each list, or first process each element and then take prefixes. We therefore have the identity

$$\mathit{listl}\,(\mathit{listl}\,f) \cdot \mathit{inits} \quad = \quad \mathit{inits} \cdot \mathit{listl}\,f.$$

In a similar fashion, the function $\mathit{reverse} : \mathit{listr}\ A \leftarrow \mathit{listr}\ A$ satisfies the identity

$$\mathit{listr}\,f \cdot \mathit{reverse} \quad = \quad \mathit{reverse} \cdot \mathit{listr}\,f.$$

Finally, the function $\mathit{zip} : \mathit{listr}\,(A \times B) \leftarrow (\mathit{listr}\ A \times \mathit{listr}\ B)$ satisfies the identity

$$\mathit{listr}\,(\mathit{cross}\,(f, g)) \cdot \mathit{zip} \quad = \quad \mathit{zip} \cdot \mathit{cross}\,(\mathit{listr}\,f, \mathit{listr}\,g),$$

where $\mathit{cross}\,(f, g)\,(a, b) = (f\ a, g\ b)$. Functions, like *concat*, *inits*, *reverse*, *zip*, and so on, which do not depend in any essential way on the structure of the elements in their arguments, will be studied in a general setting later on, where they are called *natural transformations.*

Exercises

1.17 Give proofs by induction of the various identities cited in this section.

1.18 Suppose you are given a polymorphic function *foo* with type

$$\mathit{foo} : \mathit{tree}\,(A \times B) \leftarrow (\mathit{listr}\ A \times B).$$

What identity would you expect *foo* to satisfy?

1.19 Similarly to the preceding exercise, guess the identity for

$$\mathit{foo} : \mathit{listl}\ A \leftarrow \mathit{gtree}\ A.$$

1.7 Pointwise and point-free

There are two basic styles for expressing functions, the pointwise style and the point-free style. In the pointwise style we describe a function by describing its application to arguments. Many of the examples above are expressed in the pointwise style. In the point-free style we describe a function exclusively in terms of functional composition; we have also seen some examples in this style too. In this section we want to illustrate with the aid of a small example how a point-free style leads to a very simple method for reasoning about functions.

Recall that the function $\mathit{filter}\,p$ can be defined on lists by the equation

$$\mathit{filter}\,p \quad = \quad \mathit{concat} \cdot \mathit{listr}\,(p \rightarrow \mathit{wrap}, \mathit{nilp}).$$

The function *wrap* takes a value and returns a singleton list; thus $\mathit{wrap}\ a = [a]$. The function *nilp* takes a value and returns the empty list; thus, $nilp\ a = nil$. Our aim in this section is to prove the identity

$$filter\ p \quad = \quad listr\ outl \cdot filter\ outr \cdot zip \cdot pair\,(id, listr\ p).$$

The operational reading of the right-hand side is that each element of a list is paired with its boolean value under p, and then those elements paired with *true* are selected. Although we won't go into details, the identity is useful in optimising computations of $filter\ p$ when the structure of the predicate p enables $listr\ p$ to be computed efficiently.

For the proof we will need a number of identities concerning the functions that appear in the two expressions for *filter*. The first group concerns the following combinators for expressing pairing:

$$\begin{aligned} pair\,(f,g)\ a &= (f\ a, g\ a) \\ outl\,(a,b) &= a \\ outr\,(a,b) &= b. \end{aligned}$$

These functions are related by the properties

$$outl \cdot pair\,(f,g) \quad = \quad f \qquad (1.1)$$

$$outr \cdot pair\,(f,g) \quad = \quad g. \qquad (1.2)$$

As we shall see in the next chapter, these properties characterise the notion of a categorical *product*.

For the function *nilp* we have two rules:

$$nilp \cdot f \quad = \quad nilp \qquad (1.3)$$

$$listr\ f \cdot nilp \quad = \quad nilp. \qquad (1.4)$$

The first rule states that *nilp* is a constant function, and the second rule states that this constant is the empty list.

For *wrap* and *concat* we have the rules

$$listr\ f \cdot wrap \quad = \quad wrap \cdot f \qquad (1.5)$$

$$listr\ f \cdot concat \quad = \quad concat \cdot listr\,(listr\ f). \qquad (1.6)$$

These state that *wrap* and *concat* are natural transformations.

For the function *zip* we use a similar rule:

$$zip \cdot pair\,(listr\ f, listr\ g) \quad = \quad listr\,(pair\,(f,g)). \qquad (1.7)$$

This states that *zip* is a natural transformation taking pairs of lists to lists of pairs.

For *listr* we have two rules:

$$\begin{aligned} listr\,(f \cdot g) &= listr\,f \cdot listr\,g && (1.8)\\ listr\,id &= id. && (1.9)\end{aligned}$$

As we will see in the next chapter, these rules say that *listr* is what is known as a *functor*.

For conditionals we will use the following rules:

$$\begin{aligned} (p \to f, g) \cdot h &= (p \cdot h \to f \cdot h, g \cdot h) && (1.10)\\ h \cdot (p \to f, g) &= (p \to h \cdot f, h \cdot g). && (1.11)\end{aligned}$$

These rules say how composition distributes over conditionals.

Finally, the identity function *id* satisfies two properties, namely

$$\begin{aligned} f \cdot id &= f && (1.12)\\ id \cdot f &= f. && (1.13)\end{aligned}$$

The two occurrences of *id* denote different instances of the identity function, one on the source type of f, and one on its target type.

It might appear that these dozen or so rules have been plucked out of thin air but, as we have hinted, they form coherent groups based on a small number of concepts (products, functors, natural transformations, and so on) to be studied in the next chapter. For now we just accept them.

Having armed ourselves with sufficient tools, we calculate:

$$\begin{aligned}
&\quad listr\ outl \cdot filter\ outr \cdot zip \cdot pair\,(id, listr\,p)\\
=&\quad \{\text{definition of } filter\}\\
&\quad listr\ outl \cdot concat \cdot listr\,(outr \to wrap, nil) \cdot zip \cdot pair\,(id, listr\,p)\\
=&\quad \{\text{equation } (1.6)\}\\
&\quad concat \cdot listr\,(listr\ outl) \cdot listr\,(outr \to wrap, nil) \cdot zip \cdot pair\,(id, listr\,p)\\
=&\quad \{\text{equation } (1.8)\ (\text{backwards})\}\\
&\quad concat \cdot listr\,(listr\ outl \cdot (outr \to wrap, nil) \cdot zip \cdot pair\,(id, listr\,p)\\
=&\quad \{\text{equations } (1.11), (1.5), \text{and } (1.4)\}\\
&\quad concat \cdot listr\,(outr \to wrap \cdot outl, nil) \cdot zip \cdot pair\,(id, listr\,p)\\
=&\quad \{\text{equation } (1.9)\ (\text{backwards})\}\\
&\quad concat \cdot listr\,(outr \to wrap \cdot outl, nil) \cdot zip \cdot pair\,(listr\,id, listr\,p)\\
=&\quad \{\text{equation } (1.7)\}\\
&\quad concat \cdot listr\,(outr \to wrap \cdot outl, nil) \cdot listr\,(pair\,(id, p))
\end{aligned}$$

$$
\begin{array}{ll}
= & \quad \{\text{equation (1.8) (backwards)}\} \\
& \mathit{concat} \cdot \mathit{listr}\,(\mathit{outr} \rightarrow \mathit{wrap} \cdot \mathit{outl}, \mathit{nil}) \cdot \mathit{pair}\,(\mathit{id}, p) \\
= & \quad \{\text{equations (1.10), (1.1), and (1.3)}\} \\
& \mathit{concat} \cdot \mathit{listr}\,(p \rightarrow \mathit{wrap} \cdot \mathit{id}, \mathit{nil}) \\
= & \quad \{\text{equation (1.12)}\} \\
& \mathit{concat} \cdot \mathit{listr}\,(p \rightarrow \mathit{wrap}, \mathit{nil}) \\
= & \quad \{\text{definition of } \mathit{filter}\} \\
& \mathit{filter}\, p.
\end{array}
$$

Although this calculation is fairly long – and would have been twice the length if we had not combined steps – it is very simple. Some slight variations in the order of the steps is possible; for example, we could have simplified $\mathit{zip} \cdot \mathit{pair}\,(\mathit{id}, \mathit{listr}\, p)$ to $\mathit{listr}\,(\mathit{pair}\,(\mathit{id}, p))$ earlier in the calculation. Apart from this, almost every step is forced. Indeed, when some students were set the problem in an examination, almost nobody had difficulties solving it. The problem was also given as a test example to a graduate student who had designed a simple proof editor, including a 'go' button that automatically applied identities from a given set from left to right until no more rules in the set were applicable. Apart from expressing rules (1.8) and (1.9) in reverse form, the calculation proceeded quickly and automatically to the desired conclusion, somewhat to the student's surprise.

With this single exercise we hope to have convinced the reader that point-free reasoning can be effective reasoning. Indeed, most of the many calculations to come are done in a point-free style. However, while calculations – whether point-free or pointwise – are satisfying to do, they are far less satisfying to read. It has been said that calculating is not a spectator sport. Therefore, our advice to the reader in studying a calculation is first to try and do it for oneself. Only when difficulties arise should the text be consulted. Although we have strived to present calculations in the best possible way, there will no doubt be occasions when the diligent reader can find a shorter or clearer route to the desired conclusion.

Bibliographical remarks

There are numerous introductory textbooks on functional programming; probably the best background for the material presented here is (Bird and Wadler 1988). A more modern text that is based on *Haskell* is (Davie 1992). Both of these books take *non-strict* semantics as the point of departure; a good introduction to *strict* functional programming can be found in (Paulson 1991). Other recommended books on functional programming are (Field and Harrison 1988; Henson 1987; Reade 1988; Wikström 1987). There is an archive for functional programming on the *world-wide web* which contains a wealth of articles describing the latest developments:

`http://www.lpac.ac.uk/SEL-HPC/Articles/FuncArchive.html`

Readers who wish to experiment with the programs presented in this book might consider the *Gofer* system (Jones 1994), which is freely available from

`ftp://ftp.cs.nott.ac.uk/nott-fp/languages/gofer/`

In fact, in later chapters, when we come to study some non-trivial programming examples, we shall present the result of our derivations as Gofer programs.

The realisation that functional programs are good for equational reasoning is as old as the subject itself. Two landmark papers are (Backus 1978; Burstall and Darlington 1977). More recent work on an algebraic approach to the derivation of functional programs, in which we were involved ourselves, is described in e.g. (Bird 1986, 1987; Bird and Meertens 1987; Bird, Gibbons, and Jones 1989; Bird 1989a, 1989b, 1990; Bird and De Moor 1993b; Jeuring 1989, 1990, 1994; Meertens 1987, 1989). The material of this book evolved from all these works. Quite similar in spirit, but slightly different in notation and style are (Backus 1981, 1985; Harrison and Khoshnevisan 1988; Harrison 1988; Williams 1982), and (Pettorossi and Burstall 1983; Pettorossi 1985).

Recently there has been a surge of interest in functional languages that, given the definition of a datatype, automatically provide the user with the associated fold. One approach, which is quite transparent to the naive user, can be found in (Fegaras, Sheard, and Stemple 1992; Sheard and Fegaras 1993; Kieburtz and Lewis 1995). Another approach, which is more elegant but also requires more understanding on the user's part, is the use of *constructor classes* in (Jeuring 1995; Jones 1995; Meijer and Hutton 1995).

Functions and Categories

This chapter provides a brief introduction to the elements of category theory that are necessary for understanding the rest of the book. In particular, it emphasises ways in which category theory offers economy in definitions and proofs. Subsequently, it is shown how category theory can be used in defining the basic building blocks of datatypes, and how these definitions give rise to a set of combinators that unify the operators found in functional programming and program derivation. In Chapter 3 these combinators, and the associated theory, are illustrated in a number of small but representative programming examples.

One does not so much learn category theory as absorb it over a period of time. It is difficult, at a first or second reading, to appreciate the point of many definitions and the reasons for the subject's abstract nature. We have tried to take this into account in two ways: first, by adopting a strictly minimalist style, leaving out anything that is not germane to our purpose; and second, by confining attention to a small range of examples, all drawn from the area of program specification and derivation, which is, after all, our main topic.

2.1 Categories

A *category* $\mathbf{C}$ is an algebraic structure consisting of a class of *objects*, denoted by $A, B, C, \ldots$, and so on, and a class of *arrows*, denoted by $f, g, h, \ldots$, and so on, together with three total operations and one partial operation.

The first two total operations are called *target* and *source*; both assign an object to an arrow. We write $f : A \leftarrow B$ (pronounced 'f is of type A from B') to indicate that the target of the arrow f is A and the source of f is B.

The third total operation takes an object A to an arrow $id_A : A \leftarrow A$, called the *identity arrow on* A.

The partial operation is called *composition* and takes two arrows to another one.

The composition $f \cdot g$ (pronounced 'f after g') is defined if and only if $f : A \leftarrow B$ and $g : B \leftarrow C$ for some objects A, B, and C, in which case $f \cdot g : A \leftarrow C$. In other words, if the source of f is the target of g, then $f \cdot g$ is an arrow whose target is the target of f and whose source is the source of g.

Composition is required to be associative and to have identity arrows as units:

$$f \cdot (g \cdot h) \quad = \quad (f \cdot g) \cdot h$$

for all $f : A \leftarrow B$, $g : B \leftarrow C$ and $h : C \leftarrow D$, and

$$id_A \cdot f = f = f \cdot id_B$$

for all $f : A \leftarrow B$.

Examples of categories

The motivating example of a category is **Fun**, the category of sets and total functions. In this category the objects are sets and the arrows are typed functions. More precisely, an arrow is a triple (f, A, B) in which the set A contains the range of f and the set B is the domain of f. By definition, A is the target and B the source of (f, A, B). The identity arrow $id_A : A \leftarrow A$ is the identity function on A, and the composition of two arrows (f, A, B) and (g, C, D) is defined if and only if $B = C$, in which case

$$(f, A, B) \cdot (g, B, D) \quad = \quad (f \cdot g, A, D),$$

where, on the right, $f \cdot g$ denotes the usual composition of functions f and g.

Another example of a category is **Par**, the category of sets and partial functions. The definition is similar to **Fun** except that, now, the triple (f, A, B) is an arrow if A contains the range of f and B contains the domain of f. Since a total function is a special case of a partial function, **Fun** is a *subcategory* of **Par**.

Generalising still further, a third example of a category is **Rel**, the category of sets and relations. This time the arrows are triples (R, A, B), where R is a subset of the cartesian product $A \times B$. Again, the target of (R, A, B) is A and the source B. The identity arrow $id_A : A \leftarrow A$ is the relation

$$id_A \quad = \quad \{ (a, a) \mid a \in A \}$$

and the composition of arrows (R, A, B) and (S, B, C) is the arrow (T, A, C), where, writing aRb for $(a, b) \in R$, we have

$$aTc \quad = \quad (\exists b : aRb \wedge bSc).$$

We can also combine two categories **A** and **B** to form another category $\mathbf{A} \times \mathbf{B}$, called the *product* category of **A** and **B**. The product category has, as objects, pairs (A, B), where A is an object of **A** and B is an object of **B**. The arrows are pairs (f, g), where f is an arrow of **A** and g is an arrow of **B**. Composition is defined component-wise:

$$(f, g) \cdot (h, k) \quad = \quad (f \cdot h, g \cdot k).$$

The identity arrow $id_{A \times B}$ is, of course, (id_A, id_B).

Although we shall see a number of other examples of categories in due course, **Fun**, **Par** and **Rel** – and especially **Fun** and **Rel** – will be our main focus of interest.

Diagrams

As illustrated by the above examples, the requirement that each arrow has a unique target and source can be something of a burden when it comes to spelling out the details of an expression or equation. For this reason it is quite common to refer to an arrow $f : A \leftarrow B$ simply by the identifier f, leaving A and B implicit. Furthermore, whenever one writes a composition it is implicitly assumed to be well defined. For these abbreviations to be legitimate, the type information should always be clear from the context.

A useful device for recording type information is a *diagram*. In a diagram an arrow $f : A \leftarrow B$ is represented as $A \xleftarrow{f} B$, and its composition with an arrow $g : B \leftarrow C$ is represented as $A \xleftarrow{f} B \xleftarrow{g} C$. For example, one can depict the type information in the equation $id_A \cdot f = f$ as

$$\begin{tikzcd}
A & & B \arrow[ll, "f"'] \arrow[dl, "f"] \\
& A \arrow[ul, "id_A"] &
\end{tikzcd}$$

This diagram has the property that any two paths between the same pair of objects depicts the same arrow: in such cases, a diagram is said to *commute*. As another example, here is the diagram that illustrates one of the laws of the last chapter, namely, $listr\, f \cdot wrap = wrap \cdot f$:

$$\begin{tikzcd}
listr\ A \arrow[d, "listr\ f"'] & A \arrow[l, "wrap"'] \arrow[d, "f"] \\
listr\ B & B \arrow[l, "wrap"]
\end{tikzcd}$$

It is possible to phrase precise rules about reasoning with diagrams, giving them the same formal status as, say, formulae in predicate calculus (Freyd and Ščedrov 1990). However, in what follows we shall use diagrams mainly for the simple purpose of supplying necessary type information. Just occasionally we will use a diagram in place of a calculational proof.

Reasoning with arrows

As a model for the algebra of functions a category is rather a simple structure, and one has to interpret familiar ideas about functions in terms of composition alone. As a typical example, consider how the notion of an injective function can be rendered in an arbitrary category. An arrow $m : A \leftarrow B$ is said to be *monic* if

$$f = g \quad \equiv \quad m \cdot f = m \cdot g$$

for all $f, g : B \leftarrow C$. In the particular case of **Fun**, an arrow is monic if and only if it is injective. To appreciate the calculational advantage of the above definition over the usual set-theoretic one, let us prove that the composition of two monic arrows is again monic. Suppose $m : A \leftarrow B$ and $n : B \leftarrow C$ are monic. Then we have

$$\begin{array}{cl} & m \cdot n \cdot f = m \cdot n \cdot g \\ \equiv & \quad \{\text{since } m \text{ is monic}\} \\ & n \cdot f = n \cdot g \\ \equiv & \quad \{\text{since } n \text{ is monic}\} \\ & f = g, \end{array}$$

and so $m \cdot n : A \leftarrow C$ is monic.

We can model the notion of a surjective function in an arbitrary category in a similar fashion. An arrow $e : B \leftarrow C$ is said to be *epic* if

$$f = g \quad \equiv \quad f \cdot e = g \cdot e$$

for all $f, g : A \leftarrow B$. In the particular case of **Fun**, an arrow is epic if and only if it is surjective. A symmetrical proof to the one given above for monics shows that the composition of two epics is again epic.

Duality

The exploitation of symmetry is very common in category theory and leads to substantial economy in proof. It is worth while, therefore, to consider it in a bit more detail. For any category **C** the *opposite* category $\mathbf{C}^{op}$ is defined to have the same objects and arrows as **C**, but the source and target operators are interchanged and composition is defined by swapping arguments:

$$f \cdot g \text{ in } \mathbf{C}^{op} \quad = \quad g \cdot f \text{ in } \mathbf{C}.$$

The category $\mathbf{C}^{op}$ may be thought of as being obtained from $\mathbf{C}$ by reversing all arrows. Reversing the arrows twice does not change anything, so $(\mathbf{C}^{op})^{op} = \mathbf{C}$.

Now, let $S(\mathbf{C})$ be a statement about the objects and arrows of a category $\mathbf{C}$. By reversing the direction of all arrows in $S(\mathbf{C})$, we obtain another statement $S^{op}(\mathbf{C}) = S(\mathbf{C}^{op})$ about $\mathbf{C}$. If $S(\mathbf{C})$ holds for each category $\mathbf{C}$, it follows that $S^{op}(\mathbf{C})$ also holds for each category $\mathbf{C}$. The converse implication is also true, because $(\mathbf{C}^{op})^{op} = \mathbf{C}$. We have thus proved the equivalence

$$(\forall \mathbf{C} : S(\mathbf{C})) \quad \equiv \quad (\forall \mathbf{C} : S^{op}(\mathbf{C})).$$

This special case of symmetry is called *duality*.

To illustrate, recall that above we proved that for any category $\mathbf{C}$, the statement $S(\mathbf{C})$ ='the composition of two monics in $\mathbf{C}$ is monic' is true. Reversing the arrows in the definition of *monic* gives precisely the definition of *epic*, and therefore the statement $S^{op}(\mathbf{C})$ = 'the composition of two epics in $\mathbf{C}$ is epic' is also true for any category $\mathbf{C}$. This argument is summarised by saying that epics are *dual* to monics.

Some definitions do not change when the arrows are reversed, and a typical example is the notion of an isomorphism. An *isomorphism* is an arrow $i : A \leftarrow B$ such that there exists an arrow in the opposite direction, say $j : B \leftarrow A$, such that

$$j \cdot i = id_B \quad \text{and} \quad i \cdot j = id_A.$$

It is easy to show that there exists at most one j satisfying this condition, and this unique arrow is called the *inverse* i^{-1} of i. If there exists an isomorphism $i : A \leftarrow B$, then the objects A and B are said to be *isomorphic*, and we write $A \cong B$. In **Fun** an arrow is an isomorphism if and only if it is a bijective function, and two objects are isomorphic whenever they have the same cardinality. When an arrow in **Fun** is both monic and epic it is also an isomorphism, but this is a particular property of **Fun** that does not hold in every category (see Exercise 2.6 below).

Exercises

2.1 Given is an arrow $u : A \leftarrow A$ such that $f \cdot u = f$ for all B and $f : B \leftarrow A$. Prove that $u = id_A$. It follows that identity arrows are unique.

2.2 Suppose we have four arrows $f : A \leftarrow B$, $g : C \leftarrow A$, $h : B \leftarrow A$, and $k : C \leftarrow B$. Which of the following compositions are well defined:

$$k \cdot h \cdot f \cdot h \qquad g \cdot k \cdot h \ ?$$

(Drawing a diagram will make this book-keeping exercise very easy.)

2.3 An arrow $r : A \leftarrow B$ is a *retraction* if there exists an arrow $r' : B \leftarrow A$ such that $r \cdot r' = id_A$. Show that if $r : A \leftarrow B$ is a retraction, then for any arrow $f : A \leftarrow C$ there exists an arrow $g : B \leftarrow C$ such that $r \cdot g = f$. What is the dual of a retraction? Give the dual statement of the above property of retractions.

2.4 Show that $f \cdot g = id$ implies that g is monic and f is epic. It follows that retractions are epic.

2.5 Show that if $f \cdot g$ is epic, then f is epic. What is the dual statement?

2.6 Any preorder $(A, \leq)$ can be regarded as a category: the objects are the elements of A, and there exists a unique arrow $a \leftarrow b$ precisely when $a \leq b$. What are the monic arrows? What are the epic arrows? Is every arrow that is both monic and epic an isomorphism?

2.7 A relation $R : A \leftarrow B$ is *onto* if for all $a \in A$, there exists $b \in B$ such that aRb. Is every onto relation an epic arrow in **Rel**? If not, are epic arrows in **Rel** necessarily partial functions?

2.8 For any category **A**, it is possible to construct a category $Arr(\mathbf{A})$ whose objects are the arrows of **A**. What is a suitable choice for the arrows of $Arr(\mathbf{A})$? What are the monic arrows in this category?

2.2 Functors

Abstractly defined, a *functor* is a homomorphism between categories. Given two categories **A** and **B**, a functor $\mathsf{F} : \mathbf{A} \leftarrow \mathbf{B}$ consists of two mappings: one maps objects to objects and the other maps arrows to arrows. Both mappings are usually, though not always, denoted by the same letter F. (A remark on notation: because we will need a variety of capital letters to denote relations, single-letter identifiers for functors will be written using sans serif font. On the other hand, multiple-letter identifiers for functors will be written in the normal italic font. For example, *id* denotes both the identity functor and the identity arrow.)

The two component mappings of a functor F are required to satisfy the property

$$\mathsf{F}f : \mathsf{F}A \leftarrow \mathsf{F}B \quad \text{whenever} \quad f : A \leftarrow B.$$

They are also required to preserve identities and composition:

$$\mathsf{F}(id_A) = id_{\mathsf{F}A} \quad \text{and} \quad \mathsf{F}(f \cdot g) = \mathsf{F}f \cdot \mathsf{F}g.$$

Together, these properties mean that functors take diagrams to diagrams.

Some examples of functors are given below. In the literature, the definition of a functor is often indicated by its action on objects alone. Although we will sometimes take advantage of this convention, it is not without ambiguity, since there may be many functors that have the same action on objects. In such cases we will, of course, specify both parts of a functor.

Functors can be composed in the obvious way: $(\mathsf{F} \cdot \mathsf{G})f = \mathsf{F}(\mathsf{G}f)$, and for every category $\mathbf{C}$ there exists an identity functor $id : \mathbf{C} \leftarrow \mathbf{C}$. It follows that functors are the arrows of a category in which the objects are themselves categories. Admittedly, the construction of such large categories can lead to paradoxes similar to those found in set theory; the interested reader is referred to (Lawvere 1966; Feferman 1969) for a detailed discussion. In the sequel, we will suppose that application of functors associates to the right, so $\mathsf{FG}A = \mathsf{F}(\mathsf{G}A)$. Accordingly, we will often denote composition of functors by juxtaposition, writing FG in preference to $\mathsf{F} \cdot \mathsf{G}$.

Examples of functors

Let us now look at some examples of functors. As we have already mentioned, there is an identity functor $id : \mathbf{C} \leftarrow \mathbf{C}$ for every category $\mathbf{C}$. This functor leaves objects and arrows unchanged.

An equally trivial functor is the constant functor $\mathsf{K}_A : \mathbf{A} \leftarrow \mathbf{B}$ that maps each object B of $\mathbf{B}$ to one and the same object A of $\mathbf{A}$, and each arrow f of $\mathbf{B}$ to the arrow id_A of $\mathbf{A}$. This functor preserves composition since $id_A \cdot id_A = id_A$.

Next, consider the *squaring* functor $(\_)^2 : \mathbf{Fun} \leftarrow \mathbf{Fun}$ defined by

$$\begin{aligned} A^2 &= \{(a,b) \mid a \in A, b \in A\} \\ f^2(a,b) &= (f\,a, f\,b). \end{aligned}$$

It is easy to check that the squaring functor preserves identities and composition and we leave details to the reader.

Compare the squaring functor to the *product* functor $(\times) : \mathbf{Fun} \leftarrow \mathbf{Fun} \times \mathbf{Fun}$. We will write $A \times B$ and $f \times g$ in preference to $\times(A, B)$ and $\times(f, g)$. This functor is defined by taking $A \times B$ to be the cartesian product of A and B, and

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

Again, we leave the proof that $\times$ preserves identities and composition to the reader. We met $f \times g$ in a programming context in the last chapter, where it was written as $cross\,(f, g)$.

Note that $(\times)$ takes two arguments (more precisely, a single argument consisting of a pair of values); such functors are usually referred to as *bifunctors*. A bifunctor is therefore a functor whose source is a product category. When F is a bifunctor, the functor laws take the form

$$\begin{aligned} \mathsf{F}(id, id) &= id \\ \mathsf{F}(f \cdot h, g \cdot k) &= \mathsf{F}(f, g) \cdot \mathsf{F}(h, k). \end{aligned}$$

Next, consider the functor *listr* : **Fun** ← **Fun** that takes a set A to the set *listr* A of cons-lists over A, and a function f to the function *listr* f that applies f to each element of a list. We met *listr* in the last chapter, where we made use of the following pair of laws:

$$\begin{aligned} listr\,(f \cdot g) &= listr\, f \cdot listr\, g \\ listr\, id &= id. \end{aligned}$$

Now we can see that these laws are simply the defining properties of the action of a functor on arrows. We can also see why this action is denoted by *listr* f rather than the more traditional *map* f.

Next, the *powerset* functor P : **Fun** ← **Fun** maps a set A to the powerset $\mathsf{P}A$, which is defined by

$$\mathsf{P}A = \{x \mid x \subseteq A\},$$

and a function f to the function $\mathsf{P}f$ that applies f to all elements of a given set. The powerset functor is, of course, closely related to the list functor, the only difference being that it acts on sets rather than lists.

Next, the *existential image* functor E : **Fun**←**Rel** maps a set A to $\mathsf{P}A$, the powerset of A, and a relation to its existential image function:

$$(\mathsf{E}R)\,x = \{\, a \mid (\exists b : aRb \wedge b \in x)\,\}.$$

For example, the existential image of a finite set $x : \mathsf{P}\,Nat$ under the relation $(\leq) : Nat \leftarrow Nat$ is the smallest initial segment of Nat containing x. Again, if $\in : A \leftarrow \mathsf{P}A$ denotes the membership relation on sets, then $\mathsf{E}(\in)$ is the function that takes a set of sets to the union of its members; in symbols, $\mathsf{E}(\in) = union$.

Note that E and P are very similar (they both send a set to its powerset), but they are functors between different categories: E : **Fun** ← **Rel** while P : **Fun** ← **Fun**. In fact, as we shall make more precise in a moment, P is the restriction of E to functions.

Finally, the *graph* functor J : **Rel** ← **Fun** goes the other way round to E. This functor maps every function to the corresponding set of pairs, but leaves the objects unchanged. The graph functor is an example of an *inclusion* functor that embeds a category as a subcategory of a larger one. In particular, we have $\mathsf{P} = \mathsf{EJ}$, which formalises the statement that P is the restriction of E to functions.

Exercises

2.9 Prove that functors preserve isomorphisms. That is, for any functor F and isomorphism i, the arrow $\mathsf{F}i$ is again an isomorphism.

2.10 What is a functor between preorders? (See Exercise 2.6 for the treatment of preorders as categories.)

2.11 For any category $\mathbf{C}$, define

$$\begin{aligned} \mathsf{H}(A,B) &= \{f \mid f : A \leftarrow B \text{ in } \mathbf{C}\} \\ \mathsf{H}(f,h)\,g &= f \cdot g \cdot h. \end{aligned}$$

Between what categories is H a functor?

2.12 Consider the datatype of binary trees:

$$\mathit{tree}\ A \quad ::= \quad \mathit{tip}\ A \mid \mathit{bin}\,(\mathit{tree}\ A,\ \mathit{tree}\ A)$$

This gives a mapping taking sets to sets. Extend this mapping to a functor, *i.e.* define *tree* on functions. (Later in this chapter we shall see how this can be done in general.)

2.13 The functor $\mathsf{P}' : \mathbf{Fun} \leftarrow \mathbf{Fun}$ is defined by

$$\begin{aligned} \mathsf{P}'A &= \mathsf{P}A \\ \mathsf{P}'(f : A \leftarrow B)\,x &= \{\, a \in A \mid (\forall b \in B : f\,b = a : b \in x)\,\}. \end{aligned}$$

Prove that this does indeed define a functor. Show that P' is different from P. It follows that P cannot be defined by merely giving its action on objects.

2.3 Natural transformations

Let $\mathsf{F}, \mathsf{G} : \mathbf{A} \leftarrow \mathbf{B}$ be functors between two categories $\mathbf{A}$ and $\mathbf{B}$. By definition, a *transformation* to F from G is a collection of arrows $\phi_B : \mathsf{F}B \leftarrow \mathsf{G}B$, one for each object B of $\mathbf{B}$. These arrows are called the *components* of ϕ. A transformation is called *natural* if

$$\mathsf{F}h \cdot \phi_B \quad = \quad \phi_A \cdot \mathsf{G}h$$

for all arrows $h : A \leftarrow B$ in $\mathbf{B}$. In a diagram, this equation can be pictured as

$$\begin{tikzcd} \mathsf{F}B \arrow[d, "\mathsf{F}h"'] & \mathsf{G}B \arrow[l, "\phi_B"'] \arrow[d, "\mathsf{G}h"] \\ \mathsf{F}A & \mathsf{G}A \arrow[l, "\phi_A"] \end{tikzcd}$$

We write $\phi : \mathsf{F} \leftarrow \mathsf{G}$ to indicate that a transformation ϕ to F from G is natural. One can remember the shape of the naturality condition by picturing ϕ above the arrow $\leftarrow$ between F and G and associating it both to the left $(\mathsf{F}h \cdot \phi)$ and to the right $(\phi \cdot \mathsf{G}h)$.

Examples of natural transformations

In the first chapter we met some natural transformations in the category **Fun**. For example, consider again the function *inits* that returns all prefixes of its argument:

$$\mathit{inits}[a_1, a_2, \dots a_n] = [[\,], [a_1], [a_1, a_2], \dots, [a_1, a_2, \dots, a_n]].$$

For each set A there is an arrow $\mathit{inits}_A : \mathit{listr}\,(\mathit{listr}\,A) \leftarrow \mathit{listr}\,A$. Since

$$\mathit{listr}\,(\mathit{listr}\,f) \cdot \mathit{inits} \quad = \quad \mathit{inits} \cdot \mathit{listr}\,f,$$

we have that *inits* is a natural transformation $\mathit{inits} : \mathit{listr} \cdot \mathit{listr} \leftarrow \mathit{listr}$.

Another example, again in **Fun**: the function $\mathit{fork}_A : A^2 \leftarrow A$ defined by $\mathit{fork}\ a = (a, a)$ is a natural transformation $\mathit{fork} : (\_)^2 \leftarrow \mathit{id}$. The naturality condition is

$$f^2 \cdot \mathit{fork} \quad = \quad \mathit{fork} \cdot f.$$

A natural transformation is called a natural *isomorphism* if its components are bijective. For example, in **Fun** the arrows $\mathit{swap}_{A,B} : A \times B \leftarrow B \times A$ defined by $\mathit{swap}\,(b, a) = (a, b)$ form a natural isomorphism, with naturality condition

$$(g \times f) \cdot \mathit{swap} \quad = \quad \mathit{swap} \cdot (f \times g).$$

The above examples are typical: all polymorphic functions in functional programming languages are natural transformations. This informal statement can be made precise, see, for instance, (Wadler 1989), but to do so here would go beyond the scope of the book.

Relations, that is, arrows of **Rel**, can also be natural transformations. For example, the membership relations $\in_A : A \leftarrow \mathsf{P}A$ are the components of a natural transformation: $\in : \mathit{id} \leftarrow \mathsf{JE}$. To see what this means, recall that the existential image functor

E has type **Fun** $\leftarrow$ **Rel** and the inclusion functor J has type **Rel** $\leftarrow$ **Fun**. Thus, JE : **Rel** $\leftarrow$ **Rel**. The naturality condition, namely,

$$R \cdot \in \quad = \quad \in \cdot \mathsf{JE}R,$$

says, in effect, that for any set x and relation R, the process of choosing an element a of x and then a value b such that bRa, is equivalent to the process of choosing an element of the set $\{b \mid (\exists a : bRa \wedge a \in x)\}$. This equivalence holds even when x is the empty set or R is the empty relation, for in either case both processes fail to produce a result. This particular natural transformation will be discussed at length in Chapter 4.

Composition of natural transformations

For any functor F, the identity transformation $id_{\mathsf{F}} : \mathsf{F} \leftarrow \mathsf{F}$ is given by $(id_{\mathsf{F}})_A = id_{\mathsf{F}A}$. Composition of transformations is also defined componentwise. That is, if $\phi : \mathsf{F} \leftarrow \mathsf{G}$ and $\psi : \mathsf{G} \leftarrow \mathsf{H}$, then the composite transformation $\phi \cdot \psi : \mathsf{F} \leftarrow \mathsf{H}$ is defined by

$$(\phi \cdot \psi)_A \quad = \quad \phi_A \cdot \psi_A.$$

It can easily be checked that $\phi \cdot \psi$ is natural, for one can *paste* two diagrams together:

$$\begin{tikzcd}
\mathsf{F}A & \mathsf{G}A \arrow[l, "\phi_A"'] \arrow[d, "\mathsf{G}k"] & \mathsf{H}A \arrow[l, "\psi_A"'] \arrow[d, "\mathsf{H}k"] \\
\mathsf{G}A \arrow[from=u, "\mathsf{F}k"'] & \mathsf{G}B \arrow[l, "\phi_B"] & \mathsf{H}B \arrow[l, "\psi_B"]
\end{tikzcd}$$

The outer rectangle commutes because the inner two squares do. Thus, natural transformations form the arrows of a category whose objects are functors.

One can compose a functor H with each component of a transformation $\phi : \mathsf{F} \leftarrow \mathsf{G}$ to obtain a new transformation $\mathsf{H}\phi : \mathsf{HF} \leftarrow \mathsf{HG}$. The naturality of $\mathsf{H}\phi$ follows from

$$\mathsf{HF}h \cdot \mathsf{H}\phi_A = \mathsf{H}(\mathsf{F}h \cdot \phi_A) = \mathsf{H}(\phi_B \cdot \mathsf{G}h) = \mathsf{H}\phi_B \cdot \mathsf{HG}h.$$

An example is the natural transformation $\mathsf{E}(\in) : \mathsf{E} \leftarrow \mathsf{EJE}$. As we have seen, $\mathsf{E}(\in_A) = union_A$, the function that returns the union of a collection of sets over A.

In what follows we will omit subscripts when reasoning about the components of natural transformations whenever they can be inferred from context. This is common practice when reasoning about polymorphic functions in programming.

Exercises

2.14 The text did not explicitly state the functors in the naturality condition of *swap*. What are they?

2.15 The function τ_A takes an element of A and turns it into a singleton set. Verify that $\tau : \mathsf{P} \leftarrow id$. Do we also have $\mathsf{J}\tau : \mathsf{JE} \leftarrow id$?

2.16 The function *cp* returns the cartesian product of a sequence of sets. It is defined by

$$cp\,[x_1, x_2, \ldots, x_n] \quad = \quad \{\,[a_1, a_2, \ldots, a_n] \mid \forall i : 1 \leq i \leq n : a_i \in x_i\,\}.$$

Is *cp* a natural transformation? What are the functors involved?

2.17 Let $\mathsf{F}, \mathsf{G} : \mathbf{A} \leftarrow \mathbf{B}$, and $\mathsf{H} : \mathbf{B} \leftarrow \mathbf{C}$ be functors. Furthermore, let $\phi : \mathsf{F} \leftarrow \mathsf{G}$ be a natural transformation. Define a new transformation by $(\phi\mathsf{H})_A = \phi_{\mathsf{H}A}$. What is the type of this transformation? Show that $\phi\mathsf{H}$ is a natural transformation.

In this book, we follow functional programming practice by writing ϕ instead of $\phi\mathsf{H}$.

2.18 The list functor *listr* : **Fun** ← **Fun** can be generalised to a functor **Par** ← **Par** by stipulating that *listr* $f\ x$ is undefined if there exists an element in x that is not in the domain of f. For each set A, we have an arrow *head* : $A \leftarrow$ *listr* A in **Par** that returns the first element of a list. Is *head* a natural transformation $id \leftarrow$ *listr*?

2.19 The category $\mathbf{A}^{\mathbf{B}}$ has as its objects functors $\mathbf{A} \leftarrow \mathbf{B}$ and as its arrows natural transformations. Take for $\mathbf{B}$ the category consisting of two objects, with one arrow between them. Find a category that is isomorphic to $\mathbf{A}^{\mathbf{B}}$, whose description does not make use of natural transformations or functors.

2.4 Constructing datatypes

Our objective in the next few sections is to show how the basic building blocks of datatypes can be characterised in a categorical style. We will give properties that characterise various kinds of datatype, such as products, sums, lists and trees, purely in terms of composition. These definitions therefore make sense in any category – although it can happen that, in a particular category, some datatypes do not exist.

When these definitions are interpreted in **Fun** they describe the datatypes we know from programming practice. However, as we shall see, interpreting the same definitions in **Par** or **Rel** may yield unexpected results. The discussion of these unexpected interpretations serves both to deepen our understanding of the categorical definitions, and as a motivation for Chapter 5, where datatypes are discussed in a relational setting.

The simplest datatype is a datatype with only one element, so we begin with the categorical abstraction of the notion of a singleton set.

Terminal objects

A *terminal object* of a category **C** is an object T such that for each object A of **C** there is exactly one arrow $T \leftarrow A$. Any two terminal objects are isomorphic. If T' is another terminal object, then there exist unique arrows $f : T \leftarrow T'$ and $g : T' \leftarrow T$. But since the identity $id_T : T \leftarrow T$ is the only arrow of its type, it follows that $f \cdot g = id_T$ and, by symmetry, $g \cdot f = id_{T'}$, so T and T' are isomorphic. This is sometimes summarised by saying that 'terminal objects are unique up to (unique) isomorphism'.

From now on, 1 will denote some fixed terminal object, and we shall speak of *the* terminal object. The unique arrow from A to 1 is written $!_A$. The uniqueness of $!_A$ can be expressed as an equivalence:

$$h = !_A \quad \equiv \quad h : 1 \leftarrow A. \tag{2.1}$$

Such equivalences are called *universal properties* and we shall see them in abundance in the pages to follow.

Taking 1 for A in the universal property of 1, we obtain

$$!_1 \quad = \quad id_1. \tag{2.2}$$

This identity is known as the *reflection* law. We have also the *fusion* law

$$!_A \cdot f = !_B \quad \Leftarrow \quad f : A \leftarrow B, \tag{2.3}$$

because $!_A \cdot f : 1 \leftarrow B$. Note that the fusion law may be restated as saying that $!$ is a natural transformation $\mathsf{K}_1 \leftarrow id$, where K_A is the constant functor defined in Section 2.2. Like universal properties, there will be many examples of other reflection and fusion laws in due course.

In **Fun** the terminal object is a singleton set, say $\{p\}$. The arrow $!_A$ is the constant function that maps every element of A to p. The statement that the terminal object is unique up to unique isomorphism states that all singleton sets are isomorphic in a unique way. In **Par** and **Rel** the terminal object is the empty set; in both cases the unique arrow $\{\,\} \leftarrow A$ is the empty relation $\emptyset$.

Initial objects

An *initial object* of **C** is a terminal object of $\mathbf{C}^{op}$. Thus, I is initial if for each object A of **C** there is exactly one arrow of type $A \leftarrow I$. By duality, it follows that initial objects are unique up to unique isomorphism. A commonly used notation for the initial object of **C** is 0, and the unique arrow $A \leftarrow 0$ is denoted $¡_A$. In **Fun** the initial object is the empty set and $¡_A$ is the empty function. Thus, the names 0 and 1 for the initial and terminal objects connote the cardinality of the corresponding sets in **Fun**. In **Par** and **Rel** the initial object is also the empty set, so in these categories initial and terminal objects coincide.

Exercises

2.20 An *element of* A is an arrow $e : A \leftarrow 1$. An arrow $c : A \leftarrow B$ is said to be *constant* if for all other arrows $f, g : B \leftarrow C$ we have $c \cdot f = c \cdot g$. Prove that any element is constant. Assuming that B has at least one element, show that any constant arrow $c : A \leftarrow B$ can be factored as $e \cdot !_B$ for some element e of A.

2.21 An object A is said to be *empty* if the only arrows with target A are $¡_A$ and id_A. What are the empty objects in **Fun**? Same question for **Rel** and **Fun** $\times$ **Fun**.

2.22 What does it mean to say that a preorder has a terminal object? (See Exercise 2.6 for the interpretation of preorders as categories.)

2.23 Let **A** and **B** be categories that have initial and terminal objects. Does $\mathbf{A} \times \mathbf{B}$ have initial and terminal objects?

2.24 Assuming that **A** and **B** have terminal objects, what is the terminal object in $\mathbf{A}^{\mathbf{B}}$? (For the definition of $\mathbf{A}^{\mathbf{B}}$, see Exercise 2.19.)

2.5 Products and coproducts

New datatypes can be built by tupling existing datatypes or by taking their disjoint union; the categorical abstractions considered here are the notions of product and coproduct.

Products

A *product* of two objects A and B consists of an object and two arrows. The object is written as $A \times B$ and the arrows are written $outl : A \leftarrow A \times B$ and $outr : B \leftarrow A \times B$. These three things are required to satisfy the following property: for each pair of arrows $f : A \leftarrow C$ and $g : B \leftarrow C$ there exists an arrow $\langle f, g \rangle : A \times B \leftarrow C$ such that

$$h = \langle f, g \rangle \quad \equiv \quad outl \cdot h = f \text{ and } outr \cdot h = g \qquad (2.4)$$

for all $h : A \times B \leftarrow C$. This is another example of a universal property: it states that $\langle f, g\rangle$ is the *unique* arrow satisfying the property on the right. The operator $\langle f, g\rangle$ is pronounced 'pair f and g'. The following diagram summarises the type information:

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

The diagram also illustrates the *cancellation* properties

$$outl \cdot \langle f, g\rangle = f \quad \text{and} \quad outr \cdot \langle f, g\rangle = g, \qquad (2.5)$$

which are obtained by taking $h = \langle f, g\rangle$ in the right-hand side of (2.4).

Taking *outl* for f and *outr* for g in (2.4), we obtain the reflection law

$$id \;=\; \langle outl, outr\rangle.$$

Taking $\langle h, k\rangle \cdot m$ for h in (2.4) and using (2.5), we obtain the fusion law

$$\langle h, k\rangle \cdot m = \langle f, g\rangle \;\Leftarrow\; h \cdot m = f \text{ and } k \cdot m = g.$$

In other words,

$$\langle h, k\rangle \cdot m \;=\; \langle h \cdot m, k \cdot m\rangle. \qquad (2.6)$$

Use of these rules in subsequent calculations will be signalled simply by the hint *products*.

Examples of products

In **Fun** products are given by pairing. That is, $A \times B$ is the cartesian product of A and B, and *outl* and *outr* are the obvious projection functions. In the last chapter we wrote $pair\,(f, g)$ for the arrow $\langle f, g\rangle$, with the definition

$$pair\,(f, g)\, a \;=\; (f\ a, g\ a).$$

This construction does not define a product in **Par** or **Rel** since, for example, taking g to be the everywhere undefined partial function $\emptyset$ we obtain $\langle f, \emptyset\rangle = \emptyset$ and so $outl \cdot \langle f, \emptyset\rangle = \emptyset$, not f. The discussion of products in **Par** and **Rel** is deferred until we have also discussed coproducts.

Any two categories **A** and **B** also have a product. As we have seen, the category **A** $\times$ **B** has as its objects pairs (A, B), where A is an object of **A** and B is an object of $\mathbf{B}$. Similarly, the arrows are pairs (f, g) where f is an arrow of $\mathbf{A}$ and g is an arrow of $\mathbf{B}$. Composition is defined component-wise, and *outl* and *outr* are the obvious projection functions. In fact we can turn *outl* and *outr* into functors $outl : \mathbf{A} \leftarrow \mathbf{A} \times \mathbf{B}$ and $outr : \mathbf{B} \leftarrow \mathbf{A} \times \mathbf{B}$ by defining two mappings:

$$\begin{aligned} outl(A, B) &= A \quad \text{and} \quad outr(A, B) = B \\ outl(f, g) &= f \quad \text{and} \quad outr(f, g) = g. \end{aligned}$$

Spans

There is an alternative definition of a product of A and B in a category $\mathbf{C}$, namely, as the terminal object in the category $\mathbf{Span}(A, B)$ of *spans* over A and B. A span over A and B is a pair of arrows $(f : A \leftarrow C, g : B \leftarrow C)$ with a common source. The objects of $\mathbf{Span}(A, B)$ are spans over A and B, and the arrows $m : (f, g) \leftarrow (h, k)$ are arrows of $\mathbf{C}$ satisfying

$$f \cdot m = h \quad \text{and} \quad g \cdot m = k.$$

The information is summarised in the following diagram:

$$\begin{tikzcd}
A & C \arrow[l, "f"'] \arrow[r, "g"] & B \\
& D \arrow[ul, "h"] \arrow[u, "m"'] \arrow[ur, "k"'] &
\end{tikzcd}$$

Composition in $\mathbf{Span}(A, B)$ is the same as that in $\mathbf{C}$. The particular span $(outl : A \leftarrow A \times B, outr : B \leftarrow A \times B)$ is the terminal object in $\mathbf{Span}(A, B)$, and $\langle f, g \rangle$ is just another notation for $!_{(f,g)}$. Indeed, our earlier definition (2.4) is a special case of the universal property (2.1) of terminal objects. This fact implies that products are unique up to unique isomorphism. Also, the reflection and fusion law for products are special cases of the same laws for terminal objects.

The product functor

If each pair of objects in $\mathbf{C}$ has a product, one says that $\mathbf{C}$ *has products*. In such a case $\times$ can be made into a bifunctor $\mathbf{C} \leftarrow \mathbf{C} \times \mathbf{C}$ by defining it on arrows as follows:

$$f \times g \quad = \quad \langle f \cdot outl, g \cdot outr \rangle. \tag{2.7}$$

We met $\times$ in the special case $\mathbf{C} = \mathbf{Fun}$ in Section 2.2. For general $\mathbf{C}$ the proof that $\times$ preserves identities is immediate from the reflection law $\langle outl, outr \rangle = id$. To show that $\times$ also preserves composition, that is,

$$(f \times g) \cdot (h \times k) \quad = \quad (f \cdot h) \times (g \cdot k),$$

it suffices to prove the *absorption* law

$$(f \times g) \cdot \langle p, q \rangle \quad = \quad \langle f \cdot p, g \cdot q \rangle. \tag{2.8}$$

Taking $p = h \cdot outl$ and $q = k \cdot outr$ in (2.8) gives the desired result. Here is a proof of (2.8):

$$\begin{array}{ll}
 & (f \times g) \cdot \langle p, q \rangle \\
= & \quad \{\text{definition of } \times\} \\
 & \langle f \cdot outl, g \cdot outr \rangle \cdot \langle p, q \rangle \\
= & \quad \{\text{fusion law (2.6)}\} \\
 & \langle f \cdot outl \cdot \langle p, q \rangle, g \cdot outr \cdot \langle p, q \rangle \rangle \\
= & \quad \{\text{cancellation law (2.5)}\} \\
 & \langle f \cdot p, g \cdot q \rangle.
\end{array}$$

Using the definition of $\times$ and the cancellation law (2.5), we now obtain that *outl* and *outr* are natural transformations:

$$outl : outl \leftarrow (\times) \quad \text{and} \quad outr : outr \leftarrow (\times).$$

Note the two uses of *outl* and *outr*, both as a collection of arrows and as functors between two categories. Again, use of any of the above properties in calculations will often be signalled from now on simply by the hint *products*.

Coproducts

The product of A and B in $\mathbf{C}^{op}$ is called the *coproduct* of A and B in $\mathbf{C}$. Thus coproducts, like products, also consist of one object and two arrows for each A and B. The object is denoted by $A + B$, and the two arrows by $inl : A + B \leftarrow A$ and $inr : A + B \leftarrow B$. Given $f : C \leftarrow A$ and $g : C \leftarrow B$, the unique arrow $C \leftarrow A + B$ is written $[f, g]$, and pronounced 'case f or g'. Thus, the coproduct in $\mathbf{C}$ is defined by the universal property

$$h = [f, g] \quad \equiv \quad h \cdot inl = f \text{ and } h \cdot inr = g \tag{2.9}$$

for all $h : C \leftarrow A + B$. The following diagram spells out the type information:

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

The properties of coproducts follow at once from those of products by duality, but we can also describe them in a direct approach. The cancellation properties are:

$$[f,g] \cdot inl = f \quad \text{and} \quad [f,g] \cdot inr = g,$$

These can be obtained by taking $h = [f,g]$ on the right of (2.9). Taking *inl* for f and *inr* for g in (2.9) we obtain the reflection law

$$id \;=\; [inl, inr].$$

Taking $m \cdot [h,k]$ for h, we obtain the fusion law

$$m \cdot [h,k] = [f,g] \;\Leftarrow\; m \cdot h = f \text{ and } m \cdot k = g,$$

which is the same as saying

$$m \cdot [h,k] \;=\; [m \cdot h, m \cdot k].$$

Use of these laws in calculations is signalled by the hint *coproducts*.

The coproduct functor

We also obtain a bifunctor $+$ whose definition on arrows is

$$f + g \;=\; [inl \cdot f, inr \cdot g].$$

The composition and fusion laws

$$\begin{aligned} (f+g) \cdot (h+k) &= f \cdot h + g \cdot k \\ [f,g] \cdot (h+k) &= [f \cdot h, g \cdot k] \end{aligned}$$

follow at once by duality, though one can also give a direct proof.

Coproducts in **Fun** are disjoint unions:

$$A + B \;=\; \{(a,0) \mid a \in A\} \cup \{(b,1) \mid b \in B\}.$$

Thus *inl* adds a 0 tag, while *inr* adds a 1. In a functional programming style one can do this rather more directly, avoiding the artifice of using 0 and 1 as tags, by defining $A + B$ with the type declaration

$$A + B \;::=\; inl\, A \mid inr\, B$$

and the case operator by

$$[f,g]\,(inl\, a) = f\, a \quad \text{and} \quad [f,g]\,(inr\, b) = g\, b.$$

Note that in **Fun** if A is of size m and B is of size n, then $A + B$ is of size $m + n$ while $A \times B$ is of size $m \times n$. Unlike products, coproducts in **Par** and **Rel** are defined in exactly the same way as in **Fun**.

Products in Par and Rel

As we have already indicated, we cannot define products in **Par** simply by taking the cartesian product of two sets. The reason bears repeating: the cancellation laws

$$outl \cdot \langle f, g\rangle = f \quad \text{and} \quad outr \cdot \langle f, g\rangle = g$$

fail to hold under the interpretation $\langle f, g\rangle\, a = (f\ a, g\ a)$ when f and g are partial. To be sure, in *lazy* functional programming, these laws are restored to good health by extending the notion of function to include a *bottom* element $\perp$ and making constructions such as pairing *non-strict*. We will not go into details because this approach is not exploited in this book.

Instead, we can define $A \times B$ in **Par** by

$$A \times B \quad ::= \quad inl\, A \mid mid\,(A, B) \mid inr\, B.$$

The partial function $outl : A \leftarrow A \times B$ is defined by the equations

$$\begin{aligned} outl\,(inl\ a) &= a \\ outl\,(mid\,(a, b)) &= a \\ outl\,(inr\ b) &= undefined, \end{aligned}$$

and $outr$ by

$$\begin{aligned} outr\,(inl\ a) &= undefined \\ outr\,(mid\,(a, b)) &= b \\ outr\,(inr\ b) &= b. \end{aligned}$$

The pair operator is defined by

$$\begin{aligned} \langle f, g\rangle a &= inl\,(f\ a), && \text{if } defined\,(f\ a) \text{ and } undefined\,(g\ a) \\ &= inr\,(g\ a), && \text{if } undefined\,(f\ a) \text{ and } defined\,(g\ a) \\ &= mid\,(f\ a, g\ a), && \text{otherwise.} \end{aligned}$$

To check, for example, that $(outl \cdot \langle f, g\rangle)\, a = f\ a$ for all a we have to consider four cases, depending on whether $f\ a$ and $g\ a$ is defined. Taking just one case, suppose $f\ a$ is undefined and $g\ a$ is defined. Then we have $\langle f, g\rangle a = inr\,(g\ a)$. But then $outl\,(inr\,(g\ a))$ is undefined, as required. The other cases are left as exercises.

The definition of products in **Rel** is simpler because products coincide with coproducts. That is, we can define $A \times B$ to be the disjoint union of A and B. The reason is that every relation has a converse and so **Rel** is the same as $\mathbf{Rel}^{op}$. This is the same reason why initial objects and terminal objects coincide in **Rel**. We will discuss this situation in more depth in Chapter 5.

Polynomial functors

Functors built up from constants, products and coproducts are said to be *polynomial*. More precisely, the class of polynomial functors is defined inductively by the following clauses:

- The identity functor *id* and the constant functors K_A for varying A are polynomial;
- If F and G are polynomial, then so are their composition FG, their pointwise sum $\mathsf{F}+\mathsf{G}$ and their pointwise product $\mathsf{F}\times\mathsf{G}$. These pointwise functors are defined by

$$\begin{aligned}(\mathsf{F}+\mathsf{G})\,h &= \mathsf{F}h+\mathsf{G}h\\ (\mathsf{F}\times\mathsf{G})\,h &= \mathsf{F}h\times\mathsf{G}h.\end{aligned}$$

For example, the functor F defined by $\mathsf{F}X = A + X \times A$ and $\mathsf{F}h = id_A + h \times id_A$ is polynomial because

$$\mathsf{F} = \mathsf{K}_A + (id \times \mathsf{K}_A),$$

where, in this equation, $+$ and $\times$ denote the pointwise versions.

Polynomial functors are useful in the construction of datatypes, but they are not enough by themselves; we also need *type* functors, which correspond to recursively defined types. These are discussed in Section 2.7. For datatypes that make use of function spaces, and for a categorical treatment of currying in general, we need *exponential* objects; these are discussed in Chapter 3.

Exercises

2.25 The partial order $(Nat, \leq)$ of natural numbers can be regarded as a category (see Exercise 2.6). Does this category have products? Coproducts?

2.26 Show that in any category with a terminal object and products there exist natural isomorphisms

$$\begin{aligned}unit &: A \leftarrow A \times 1\\ swap &: A \times B \leftarrow B \times A\\ assocr &: A \times (B \times C) \leftarrow (A \times B) \times C.\end{aligned}$$

These natural isomorphisms arise in a number of examples later in the book. The inverse arrow for *assocr* will be denoted by *assocl*; thus,

$$assocl : (A \times B) \times C \leftarrow A \times (B \times C)$$

satisfies $assocl \cdot assocr = id$ and $assocr \cdot assocl = id$.

2.27 Prove the *exchange* law

$$\langle [f,g],[h,k] \rangle \quad = \quad [\langle f,h \rangle, \langle g,k \rangle].$$

2.28 Consider products and coproducts in **Fun**. Are the projections $(outl, outr)$ epic? Are the injections (inl, inr) monic? If the answers to these two questions are different, does this contradict duality?

2.29 Let $\mathbf{A}$ be a category with products. What are the products in $Arr(\mathbf{A})$? (See Exercise 2.8 for the definition of $Arr(\mathbf{A})$.)

2.30 Complete the verification of the construction of products in **Par**.

2.31 A lazy functional programming language can be regarded as a category, where the types are objects and the arrows are (meanings of) programs. Does pair forming give a categorical product in this category?

2.6 Initial algebras

In order to say exactly what a recursively defined datatype is, we need one final piece of machinery: the notion of an initial algebra.

Let $\mathsf{F} : \mathbf{C} \leftarrow \mathbf{C}$ be a functor. By definition, an F-*algebra* is an arrow of type $A \leftarrow \mathsf{F}A$, the object A being called the *carrier* of the algebra. For example, the algebra $(Nat, +)$ of the natural numbers and addition is an algebra of the functor $\mathsf{F}A = A \times A$ and $\mathsf{F}h = h \times h$.

A F-*homomorphism* to an algebra $f : A \leftarrow \mathsf{F}A$ from an algebra $g : B \leftarrow \mathsf{F}B$ is an arrow $h : A \leftarrow B$ such that

$$h \cdot g \quad = \quad f \cdot \mathsf{F}h.$$

The type information is provided by the diagram:

$$\begin{tikzcd} B \arrow[d, "h"'] & \mathsf{F}B \arrow[l, "g"'] \arrow[d, "\mathsf{F}h"] \\ A & \mathsf{F}A \arrow[l, "f"] \end{tikzcd}$$

To give just one simple illustration, consider the algebra $(+) : Nat \leftarrow Nat^2$ of addition, and the algebra $(\oplus) : Nat_p \leftarrow Nat_p^2$ of addition modulo p, where $Nat_p = \{0, 1, \ldots, p-1\}$ and $n \oplus m = (n + m)\ mod\ p$. The function $h\,n = n \bmod p$ is a $(\_)^2$-homomorphism to $\oplus$ from $+$.

Identity arrows are homomorphisms, and the composition of two homomorphisms is again a homomorphism, so F-algebras form the objects of a category **Alg**(F) whose arrows are homomorphisms. For many functors, including the polynomial functors of **Fun**, this category has an initial object, which we shall denote by $\alpha : T \leftarrow \mathsf{F}T$ (the letter T stands for 'Type' and also for 'Term' since such algebras are often called term algebras). The proof that these initial algebras exist is beyond the scope of the book; the interested reader should consult (Manes and Arbib 1986).

The existence of an initial F-algebra means that for any other F-algebra $f : A \leftarrow \mathsf{F}A$, there is a unique homomorphism to f from α. We will denote this homomorphism by $(\!|f|\!)$, so $(\!|f|\!) : A \leftarrow T$ is characterised by the universal property

$$h = (\!|f|\!) \quad \equiv \quad h \cdot \alpha = f \cdot \mathsf{F}h. \tag{2.10}$$

The type information is summarised in the diagram:

$$\begin{tikzcd}
T \arrow[d, "(\!|f|\!)"'] & \mathsf{F}T \arrow[l, "\alpha"'] \arrow[d, "\mathsf{F}(\!|f|\!)"] \\
A & \mathsf{F}A \arrow[l, "f"]
\end{tikzcd}$$

Arrows of the form $(\!|f|\!)$ are called *catamorphisms*, and we shall refer to uses of the above equivalence by the hint *catamorphisms*. (The word 'catamorphism' is derived from the greek preposition $\kappa\alpha\tau\alpha$ meaning 'downwards'.) Catamorphisms, like other constructions by universal properties, satisfy fusion and reflection laws. Before giving these, let us first pause to give two examples that reveal the notion of a catamorphism to be a familiar idea in abstract clothing.

Natural numbers

Initial algebras of the category **Fun** will be named by type declarations of the kind commonly found in functional programming. For example,

$$Nat \quad ::= \quad zero \mid succ\, Nat$$

declares $[zero, succ] : Nat \leftarrow \mathsf{F}\,Nat$ to be the initial algebra of the functor F defined by $\mathsf{F}A = 1 + A$ and $\mathsf{F}h = id_1 + h$. Here $zero : Nat \leftarrow 1$ is a constant function. The names *Nat*, *zero* and *succ* are inspired by the fact that we can think of *Nat* as the natural numbers, *zero* as the constant function returning 0, and *succ* as the successor function. The functor F is polynomial, so the category **Alg**(F) has an initial object; the purpose of the type declaration is to give a name to this initial algebra.

Every algebra of the functor $\mathsf{F} : \mathbf{Fun} \leftarrow \mathbf{Fun}$ takes the form $[c, f]$ for some constant function $c : A \leftarrow 1$ and function $f : A \leftarrow A$. To see this, let $h : A \leftarrow \mathsf{F}A$ be an F-algebra. We have $h = [h \cdot inl, h \cdot inr]$, so we can set $c = h \cdot inl$ and $f = h \cdot inr$. It is clumsy to write $(\![[c, f]]\!)$ so we shall drop the inner brackets and write $(\![c, f]\!)$ instead.

It is helpful to spell out exactly what function h is defined by $h = (\![c, f]\!)$. Simplifying the definition, we find

$$\begin{array}{ll}
& h \cdot \alpha = [c, f] \cdot \mathsf{F}h \\
\equiv & \quad \{\text{definition of } \mathsf{F}\} \\
& h \cdot \alpha = [c, f] \cdot (id_1 + h) \\
\equiv & \quad \{\text{coproduct}\} \\
& h \cdot \alpha = [c, f \cdot h] \\
\equiv & \quad \{\text{since } \alpha = [zero, succ]\} \\
& h \cdot [zero, succ] = [c, f \cdot h] \\
\equiv & \quad \{\text{coproduct}\} \\
& [h \cdot zero, h \cdot succ] = [c, f \cdot h] \\
\equiv & \quad \{\text{cancellation}\} \\
& h \cdot zero = c \text{ and } h \cdot succ = f \cdot h.
\end{array}$$

$$\begin{tikzcd}
T & FT \arrow[l, "\alpha"'] \\
A \arrow[u, leftarrow, "h"] & FA \arrow[l, "f"] \arrow[u, leftarrow, "Fh"']
\end{tikzcd}$$

$h \cdot \alpha = [x, y] \cdot (id + id \times f)$

$(h \cdot nil, h \cdot cons) = [x, y \cdot (id \times f)]$

Writing 0 for the particular constant returned by the constant function *zero* and $n + 1$ for $succ\, n$, we now see that $h = (\![c, f]\!)$ is the unique solution of the two equations

$$\begin{array}{rcl}
h\,(0) & = & c \\
h\,(n+1) & = & f(h\,n).
\end{array}$$

In other words, $h = foldn\,(c, f)$. Thus $(\![c, f]\!) = foldn\,(c, f)$ in the datatype *Nat*.

Strings

The second example deals with lists of characters, also called *strings*:

$$String \quad ::= \quad nil \mid cons\,(Char, String).$$

In the next section we will generalise this datatype to lists over an arbitrary type, but it is worth while considering the simpler case first. The above declaration names $[nil, cons] : String \leftarrow \mathsf{F}\,String$ to be the initial algebra of the functor $\mathsf{F}A = 1 + (Char \times A)$ and $\mathsf{F}f = id + (id \times f)$. In particular, $nil : String \leftarrow 1$ is a constant function, returning the empty string.

Like the example of *Nat* given above, every algebra of this functor takes the form $[c, f]$ for some constant $c : A \leftarrow 1$ and function $f : A \leftarrow Char \times A$. Simplifying, we find that $h = ([c, f])$ is the unique solution of the equations

$$\begin{aligned} h\, nil &= c \\ h\,(cons\,(a, x)) &= f(a, h\, x). \end{aligned}$$

In other words, $([c, f]) = foldr\,(c, f)$ in the datatype *String*. So, once again, $([c, f])$ corresponds to a fold operator.

Fusion

From the definition of catamorphisms we immediately obtain the reflection law

$$([\alpha]) \quad = \quad id \tag{2.11}$$

and the very useful fusion law

$$h \cdot ([f]) = ([g]) \quad \Leftarrow \quad h \cdot f = g \cdot \mathsf{F}h. \tag{2.12}$$

The fusion law can be proved by looking at the diagram

$$\begin{tikzcd} T \arrow[d, "([f])"'] & \mathsf{F}T \arrow[l, "\alpha"'] \arrow[d, "\mathsf{F}([f])"] \\ A \arrow[d, "h"'] & \mathsf{F}A \arrow[l, "f"'] \arrow[d, "\mathsf{F}h"] \\ B & \mathsf{F}B \arrow[l, "g"'] \end{tikzcd}$$

This diagram commutes because the lower part does (by assumption) and the upper part does (by definition of catamorphism). But since $([g])$ is the unique homomorphism from α to g, we conclude that $([g]) = h \cdot ([f])$.

The fusion law for catamorphisms is probably the most useful tool in the arsenal of techniques for program derivation, and we shall see literally dozens of uses in the programming examples given in the remainder of the book. In particular, it can be used to prove that α is an isomorphism. Suppose in the statement of fusion we take both g and h to be α. Then we obtain $\alpha \cdot ([f]) = ([\alpha]) = id$ provided $\alpha \cdot f = \alpha \cdot \mathsf{F}\alpha$. Clearly, we can choose $f = \mathsf{F}\alpha$ and as a result we obtain $\alpha \cdot ([\mathsf{F}\alpha]) = id$. We can also show that $([\mathsf{F}\alpha]) \cdot \alpha = id$:

$$([\mathsf{F}\alpha]) \cdot \alpha$$

$$
\begin{aligned}
&= \quad \{\text{cata}\} \\
&\mathsf{F}\alpha \cdot \mathsf{F}(\!|\mathsf{F}\alpha|\!) \\
&= \quad \{\mathsf{F}\ \text{functor}\} \\
&\mathsf{F}(\alpha \cdot (\!|\mathsf{F}\alpha|\!)) \\
&= \quad \{\text{above}\} \\
&\mathsf{F}\mathit{id} \\
&= \quad \{\mathsf{F}\ \text{functor}\} \\
&\mathit{id}
\end{aligned}
$$

The fact that α is an isomorphism was first recorded in (Lambek 1968), and it is sometimes referred to as *Lambek's Lemma.* Motivated by his lemma, Lambek called (α, T) a *fixpoint* of F, but we shall not use this terminology.

Exercises

2.32 Let $f_0 : A \leftarrow B \times A$, $f_1 : A \leftarrow A \times C$ and $f_2 : A \leftarrow B$. Define a functor F, an F-algebra $g : A \leftarrow \mathsf{F}A$, and mappings ϕ_i $(i = 0, 1, 2)$ such that $\phi_i\, g = f_i$.

2.33 What is the initial algebra of the identity functor?

2.34 Let $\alpha : T \leftarrow \mathsf{F}T$ be the initial algebra of F. Prove that $m' \cdot m = id_T$ implies that m is a catamorphism.

2.35 Show that $(\!|f \cdot g|\!) = f \cdot (\!|g \cdot \mathsf{F}f|\!)$.

2.36 Let $\alpha : T \leftarrow \mathsf{F}T$ be the initial algebra of F. Show that if $f : A \leftarrow T$, then $f = \mathit{outl} \cdot (\!|g|\!)$ for some g.

2.37 Give an example of a functor of **Fun** that does not have an initial algebra. (*Hint*: think of an operator F taking sets to sets such that $\mathsf{F}\,A$ is not isomorphic to A for any A.)

2.7 Type functors

Datatypes are often parameterised. For example, we can generalise the example of strings described above to a datatype of cons-lists over an arbitrary A:

$$\mathit{listr}\ A \quad ::= \quad \mathit{nil} \mid \mathit{cons}\,(A, \mathit{listr}\ A).$$

This declares $[\mathit{nil}, \mathit{cons}]_A : \mathit{listr}\ A \leftarrow \mathsf{F}_A(\mathit{listr}\ A)$ to be the initial algebra of the functor F_A defined by $\mathsf{F}_A(B) = 1 + (A \times B)$ and $\mathsf{F}_A(f) = \mathit{id} + (\mathit{id} \times f)$. We can, and will, write $\mathsf{F}(A,B)$ instead of $\mathsf{F}_A(B)$, in which case we think of F as a bifunctor. We will always arrange the arguments of a bifunctor so that the functor obtained by fixing the first argument (and varying the second) is the one that describes the initial algebra.

To illustrate this important convention, consider the declaration

$$listl\,A \quad ::= \quad nil \mid snoc\,(listl\,A, A),$$

which describes the type of snoc-lists over A. Snoc-lists are similar to cons-lists except that we build them by adding to the end rather than to the beginning of the list. The algebra $[nil, snoc]$ is the initial algebra of the bifunctor

$$\mathsf{F}(A,B) \quad = \quad 1 + (B \times A).$$

Fixing the first argument gives us a functor $\mathsf{F}_A(f) = \mathsf{F}(id_A, f)$ and it is this functor that describes the initial algebra.

Let F be a bifunctor with the collection of initial algebras $\alpha_A : \mathsf{T}A \leftarrow \mathsf{F}(A, \mathsf{T}A)$. The construction T can be made into a functor by defining

$$\mathsf{T}f \quad = \quad (\!|\alpha \cdot \mathsf{F}(f, id)|\!). \tag{2.13}$$

For example, the cons-list functor is defined by

$$listr\,f \quad = \quad (\!|[nil, cons] \cdot (id + (f \times id))|\!)$$

which simplifies to $listr\,f = (\!|nil, cons \cdot (f \times id)|\!)$. Translated to the point level, this reads

$$\begin{aligned} listr\,f\,nil \quad &= \quad nil \\ listr\,f\,(cons\,(a,x)) \quad &= \quad cons\,(f\,a, listr\,f\,x), \end{aligned}$$

so $listr\,f$ is just what functional programmers would call $map\,f$, or $maplist\,f$.

We have, of course, to prove that T preserves identities and composition, so let us do it. First:

$$\begin{aligned} & \mathsf{T}\,id \\ = \quad & \quad \{\text{definition}\} \\ & (\!|\alpha \cdot \mathsf{F}(id, id)|\!) \\ = \quad & \quad \{\text{bifunctors preserve identities}\} \\ & (\!|\alpha|\!) \\ = \quad & \quad \{\text{reflection law}\} \\ & id. \end{aligned}$$

Second:

$$
\begin{array}{cl}
 & \mathsf{T}f \cdot \mathsf{T}g \\
= & \quad \{\text{definition}\} \\
 & (\!|\alpha \cdot \mathsf{F}(f, id)|\!) \cdot \mathsf{T}g \\
= & \quad \{\text{fusion (see below)}\} \\
 & (\!|\alpha \cdot \mathsf{F}(f, id) \cdot \mathsf{F}(g, id)|\!) \\
= & \quad \{\mathsf{F}\ \text{bifunctor}\} \\
 & (\!|\alpha \cdot \mathsf{F}(f \cdot g, id)|\!) \\
= & \quad \{\text{definition}\} \\
 & \mathsf{T}(f \cdot g).
\end{array}
$$

The appeal to fusion is justified by the following more general argument:

$$
\begin{array}{cl}
 & (\!|h|\!) \cdot \mathsf{T}g = (\!|h \cdot \mathsf{F}(g, id)|\!) \\
\equiv & \quad \{\text{definition of } \mathsf{T}\} \\
 & (\!|h|\!) \cdot (\!|\alpha \cdot \mathsf{F}(g, id)|\!) = (\!|h \cdot \mathsf{F}(g, id)|\!) \\
\Leftarrow & \quad \{\text{fusion}\} \\
 & (\!|h|\!) \cdot \alpha \cdot \mathsf{F}(g, id) = h \cdot \mathsf{F}(g, id) \cdot \mathsf{F}(id, (\!|h|\!)) \\
\equiv & \quad \{\text{cata}\} \\
 & h \cdot \mathsf{F}(id, (\!|h|\!)) \cdot \mathsf{F}(g, id) = h \cdot \mathsf{F}(g, id) \cdot \mathsf{F}(id, (\!|h|\!)) \\
\equiv & \quad \{\mathsf{F}\ \text{bifunctor}\} \\
 & \mathit{true}.
\end{array}
$$

This argument in effect shows that

$$(\!|h|\!) \cdot \mathsf{T}g \quad = \quad (\!|h \cdot \mathsf{F}(g, id)|\!). \tag{2.14}$$

In words, a catamorphism composed with its type functor can always be expressed as a single catamorphism. Equation (2.14) is quite useful by itself and we shall refer to it in calculations by the hint *type functor fusion*. To give just one example now: if $sum = (\!|zero, plus|\!)$ is the function $sum : Nat \leftarrow listr\ Nat$, then

$$sum \cdot listr\, f \quad = \quad (\!|zero, plus \cdot (f \times id)|\!).$$

Now that we have established that T is a functor, we can show that $\alpha : \mathsf{T} \leftarrow \mathsf{G}$ is a natural transformation, where $\mathsf{G}f = \mathsf{F}(f, \mathsf{T}f)$. We argue in a line:

$$\mathsf{T}f \cdot \alpha = \alpha \cdot \mathsf{F}(f, id) \cdot \mathsf{F}(id, \mathsf{T}f) = \alpha \cdot \mathsf{F}(f, \mathsf{T}f) = \alpha \cdot \mathsf{G}f.$$

In what follows we will say that (α, T) is the *initial type* defined by the bifunctor F.

Before passing on to examples we make three remarks. The first is that it is important not to confuse the type functor T associated with a datatype with the functor F that defines the structure of the datatype. We will call the latter the *base* functor. For example, the datatype of cons-lists over an arbitrary A has as base functor the functor F defined on arrows by $\mathsf{F}f = id_1 + id_A \times f$, whereas the type functor *listr* is defined on arrows by $listr\, f = (\!|nil, cons \cdot (f \times id)|\!)$.

The second remark is that, subject to certain healthiness conditions on the functor involved, the initial algebras in **Par** and **Rel** coincide with those in **Fun**. This will be proved in Chapter 5.

The third remark concerns duality. As with the definitions of terminal objects and products, one may dualise the above discussion to *coalgebras*. This gives a clean description, for instance, of infinite lists. We shall not have any use for such infinite data structures, however, and their discussion is therefore omitted. The interested reader is referred to (Manes and Arbib 1986; Malcolm 1990b; Hagino 1989) for details.

Exercises

2.38 The discussion of initial types does in fact allow bifunctors of type $\mathsf{F} : \mathbf{A} \leftarrow (\mathbf{B} \times \mathbf{A})$. Consider the the initial type (α, T). Between what categories is T a functor? An example where $\mathbf{B} = \mathbf{Fun} \times \mathbf{Fun}$ and $\mathbf{A} = \mathbf{Fun}$ is

$$\mathsf{F}((f, g), h) \quad = \quad f + g.$$

What is the initial type of this bifunctor?

2.39 Let F be a bifunctor, and let (α, T) be the corresponding initial type. Let G and H be unary functors, and define $\mathsf{L}A = \mathsf{F}(\mathsf{G}A, \mathsf{H}A)$. Prove that if $\phi : \mathsf{H} \leftarrow \mathsf{L}$, then $(\!|\phi|\!) : \mathsf{H} \leftarrow \mathsf{TG}$.

2.40 A *monad* is a functor $\mathsf{H} : \mathbf{A} \leftarrow \mathbf{A}$, together with two natural transformations $\eta : \mathsf{H} \leftarrow id$ and $\mu : \mathsf{H} \leftarrow \mathsf{HH}$, such that

$$\mu \cdot \mathsf{H}\eta = id = \mu \cdot \eta \quad \text{and} \quad \mu \cdot \mu = \mu \cdot \mathsf{H}\mu.$$

Many initial types give rise to a monad, and the purpose of this exercise is to prove that fact. Let F be a bifunctor given by

$$\mathsf{F}(f, g) \quad = \quad f + \mathsf{G}g,$$

for some other functor G. Let (α, T) be the initial type of F. Define $\phi = \alpha \cdot inl$ and $\psi = (\!|id, \alpha \cdot inr|\!)$. Prove that (T, ϕ, ψ) is a monad, and work out what this means for the special case where $\mathsf{G}g = g \times g$.

Bibliographical remarks

The material presented in this chapter is well documented in the literature. There is now a variety of textbooks on category theory that are aimed at the computing science community, for instance (Asperti and Longo 1991; Barr and Wells 1990; Pierce 1991; Rydeheard and Burstall 1988; Walters 1992a). The idea to use initiality for reasoning about programs goes back at least to (Burstall and Landin 1969), and was reinforced in (Goguen 1980). However, this work did not make use of F-algebras and thus lacks the conciseness that gives the approach its charm. Nevertheless, the advantages of algebra in program construction were amply demonstrated by the CIP-L project, see e.g. (Bauer, Berghammer, Broy, Dosch, Geiselbrechtinger, Gnatz, Hangel, Hesse, Krieg-Brückner, B., Laut, A., Matzner, T., Möller, B., Nickl, F., Partsch, H., Pepper, P., Samelson, K., Wirsing, M., and Wössner, H. 1985; Bauer, Ehler, Horsch, Möller, Partsch, Paukner, O., and Pepper, P. 1987; Partsch 1990).

The notion of F-algebras first appeared in the categorical literature during the 1960s, for instance in (Lambek 1968). Long before the applications to program derivation were realised, numerous authors e.g. (Lehmann and Smyth 1981; Manes and Arbib 1986) pointed out the advantages of F-algebras in the area of program semantics. Hagino used a generalisation of F-algebras in designing a categorical programming language (Hagino 1987a, 1987b, 1989, 1993), and (Cockett and Fukushima 1991) have similar goals.

It is (Malcolm 1990a, 1990b) who deserves credit for first making the program derivation community aware of this work. The particular treatment of datatypes given here is strongly influenced by the presentations of our colleagues in (Spivey 1989; Gibbons 1991, 1993; Fokkinga 1992a, 1992b, 1992c; Jeuring 1991, 1993; Meertens 1992; Meijer 1992; Paterson 1988). In particular, Fokkinga's thesis contains a much more thorough account of the foundations, and Jeuring presents some spectacular applications. The paper by (Meijer, Fokkinga, and Paterson 1991) is an introduction specially aimed at functional programmers.

One topic that we avoid in this book (except briefly in Section 5.6) is the categorical treatment of datatypes that satisfy equational laws. An example of such a datatype is, for instance, the datatype of finite bags. Our reason for not discussing such datatypes is that we feel the benefits in later chapters are not quite justified by the technical machinery required. The neatest categorical approach that we know of to datatypes with laws is (Fokkinga 1996); see also (Manes 1975). There are of course many data structures that are not easily expressed in terms of initial algebras, but recently it has been suggested that even graphs fit the framework presented here, provided laws are introduced (Gibbons 1995).

Another issue that we shall not address is that of mechanised reasoning. We are hopeful, however, that the material presented here can be successfully employed in a mechanised reasoning system: see, for instance, (Martin and Nipkow 1990).

Chapter 3

Applications

$$\begin{tikzcd} T \arrow[d, "(\!|f|\!)"'] & FT \arrow[l, "\alpha"'] \arrow[d, "F(\!|f|\!)"] \\ A & FA \arrow[l, "f"'] \end{tikzcd}$$

Let us now come down to earth and illustrate some of the abstract machinery we have set up in the preceding chapter with a number of programming techniques and examples. We also take the opportunity to discuss some features of functional programming that have not been covered so far in a categorical setting. These include the use of currying and conditionals.

3.1 Banana-split

Recall that the type of cons-lists over A is defined by

$$\mathit{listr}\ A \quad ::= \quad \mathit{nil} \mid \mathit{cons}\,(A, \mathit{listr}\ A).$$

The function *sum* returns the sum of a list of numbers and is defined by the catamorphism

$$\mathit{sum} \quad = \quad (\![\mathit{zero}, \mathit{plus}]\!),$$

where $\mathit{plus}(a, b) = a+b$. Similarly, the function *length* is defined by a catamorphism

$$\mathit{length} \quad = \quad (\![\mathit{zero}, \mathit{succ} \cdot \mathit{outr}]\!).$$

Given these two functions we can define the function *average* by

$$\mathit{average} \quad = \quad \mathit{div} \cdot \langle \mathit{sum}, \mathit{length} \rangle,$$

where $\mathit{div}(m, n) = m/n$. Of course, applied to the empty list *average* returns 0/0 and we had better fix this problem if *average* is to be a total function. So let $\mathit{div}(0, 0) = 0$.

Naive implementation of this definition of *average* yields a program that traverses its argument list twice: once for the computation of *sum*, and once for the computation of *length*. An obvious strategy to obtain a one-pass program is to express

$\langle sum, length \rangle$ as a single catamorphism. This is in fact possible for any pair of catamorphisms, irrespective of the details of this particular problem: we have

$$\langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle \quad = \quad \llparenthesis \langle h \cdot \mathsf{F}\, outl, k \cdot \mathsf{F}\, outr \rangle \rrparenthesis,$$

where F is the – so far – unmentioned base functor of the catamorphism. The above identity is known among researchers in the field as the *banana-split* law (because catamorphism brackets are like bananas, and because the pairing operator has also been called 'split' in the literature). To prove the banana-split law, it suffices by the universal property of catamorphisms to show that

$$\langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle \cdot \alpha \quad = \quad \langle h \cdot \mathsf{F}\, outl, k \cdot \mathsf{F}\, outr \rangle \cdot \mathsf{F} \langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle.$$

This equation can be verified as follows:

$$\begin{array}{cl}
 & \langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle \cdot \alpha \\
= & \quad \{\text{split fusion}\} \\
 & \langle \llparenthesis h \rrparenthesis \cdot \alpha, \llparenthesis k \rrparenthesis \cdot \alpha \rangle \\
= & \quad \{\text{catamorphisms}\} \\
 & \langle h \cdot \mathsf{F} \llparenthesis h \rrparenthesis, k \cdot \mathsf{F} \llparenthesis k \rrparenthesis \rangle \\
= & \quad \{\text{split cancellation (backwards)}\} \\
 & \langle h \cdot \mathsf{F}(outl \cdot \langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle), k \cdot \mathsf{F}(outr \cdot \langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle) \rangle \\
= & \quad \{\mathsf{F}\ \text{functor}\} \\
 & \langle h \cdot \mathsf{F} outl \cdot \mathsf{F} \langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle, k \cdot \mathsf{F} outr \cdot \mathsf{F} \langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle \rangle \\
= & \quad \{\text{split fusion (backwards)}\} \\
 & \langle h \cdot \mathsf{F} outl, k \cdot \mathsf{F} outr \rangle \cdot \mathsf{F} \langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle.
\end{array}$$

Applying the banana-split law to the particular problem of writing $\langle sum, length \rangle$ as a catamorphism, we find that

$$\langle sum, length \rangle \quad = \quad \llparenthesis zeros, pluss \rrparenthesis$$

where $zeros = \langle zero, zero \rangle$, and $pluss(a, (b, n)) = (a + b, n + 1)$. The banana-split law is a perfect example of the power of the categorical approach: a simple technique of program optimisation involving the merging of two loops is generalised to structural recursion over arbitrary datatypes and proved with a short and convincing argument.

$[zero, plus \cdot (N \times outl)]$ $\qquad$ $[zero, succ \cdot (N \times outr)]$

Exercises

3.1 Let $\mathsf{F}X = 1 + (N \times X)$. Show that

$$\langle [zero, plus] \cdot \mathsf{F} outl, [zero, succ \cdot outr] \cdot \mathsf{F} outr \rangle \quad = \quad [zeross, pluss].$$

3.2 Let $\mathsf{F} : \mathbf{C} \leftarrow \mathbf{C}$, where $\mathbf{C}$ is a category that has products. Define $\phi = \langle \mathsf{F}\,outl, \mathsf{F}\,outr \rangle$. Between what functors is ϕ a natural transformation? Prove that the naturality condition is indeed satisfied.

3.3 A list of numbers is called *steep* if each element is greater than the sum of the elements that follow it:

$$\begin{aligned} steep\ nil &= true \\ steep\,(cons\,(a, x)) &= a > sum\,x \ \wedge\ steep\,x. \end{aligned}$$

$$\begin{tikzcd} (A \times B) & \mathsf{F}(A \times B) \arrow[l] \arrow[d] \\ C \arrow[from=u] & \mathsf{F}C \arrow[l] \end{tikzcd}$$

A naive implementation takes quadratic time. Give a linear-time program.

3.4 The pattern in the preceding exercise can be generalised as follows. Suppose that $h : B \leftarrow \mathsf{F}B$, and

$$\begin{tikzcd} T \arrow[d, "f"'] & \mathsf{F}T \arrow[l, "\alpha"'] \arrow[d, "\mathsf{F}\langle f, (\![h]\!) \rangle"] \\ A & \mathsf{F}(A \times B) \arrow[l, "g"] \end{tikzcd}$$

commutes. Construct k such that $f = outl \cdot (\![k]\!)$ and prove that your construction works.

3.5 Consider the datatype of trees:

$$tree\ A \ ::= \ null \mid node\,(tree\ A, A, tree\ A).$$

A tree is *balanced* if at each node we have

$$1/3 \ \leq \ n/(n + m + 1) \ \leq \ 2/3,$$

where n and m are the sizes of the left and right subtree respectively.

Apply the preceding exercise to obtain an efficient program for testing whether a tree is balanced.

3.6 The function $preds : list\ Nat \leftarrow Nat$ takes a natural number n and returns the list $[n, n-1, \ldots, 1]$. Apply Exercise 3.4 to write *preds* in terms of a catamorphism.

3.7 The factorial function can be defined as

$$fact \ = \ product \cdot preds,$$

where *product* returns the product of a list of numbers. Use the preceding exercise and fusion to obtain a more efficient solution.

3.8 Prove *Fokkinga's mutual recursion theorem*:

$$\begin{array}{ll} & f \cdot \alpha = h \cdot \mathsf{F}\langle f, g\rangle \ \wedge \ g \cdot \alpha = k \cdot \mathsf{F}\langle f, g\rangle \\ \equiv & \\ & \langle f, g\rangle = (\!|\langle h, k\rangle|\!). \end{array}$$

It may be helpful to start by drawing a diagram of the types involved. Show that the banana-split law and Exercise 3.4 are special cases of the mutual recursion theorem.

3.2 Ruby triangles and Horner's rule

The initial type of cons-lists is the basis of the circuit design language Ruby (Jones and Sheeran 1990), which is in many ways similar to the calculus used in this book. Ruby does, however, have a number of additional primitives. One of these primitives is called *triangle*. For any function $f : A \leftarrow A$, the function $\mathit{tri}\, f : \mathit{listr}\, A \leftarrow \mathit{listr}\, A$ is defined informally by

$$\mathit{tri}\, f\, [a_0, a_1, \ldots, a_i, \ldots, a_n] = [a_0, f\, a_1, \ldots, f^i\, a_i, \ldots, f^n\, a_n].$$

In Ruby the single most important result for reasoning about triangles is the following one. For all f and c,

$$(\!|c, g|\!) \cdot \mathit{tri}\, f = (\!|c, g \cdot (\mathit{id} \times f)|\!) \quad \Leftarrow \quad f \cdot c = c \text{ and } f \cdot g = g \cdot (f \times f).$$

In Ruby, this fact is called *Horner's rule*, because it generalises the well-known method for evaluating polynomials. If we take $c = 0$, $g(a, b) = a+b$, and $f\, a = a\times x$, then the above equation states that because

$$\begin{array}{rcl} 0 \times x & = & 0 \\ (a + b) \times x & = & a \times x + b \times x, \end{array}$$

we have

$$\begin{array}{ll} & a_0 + a_1 \times x + a_2 \times x^2 + \cdots + a_n \times x^n \\ = & a_0 + (a_1 + (a_2 + \cdots (a_n + 0) \times x \cdots) \times x). \end{array}$$

In Ruby, Horner's rule is stated only for the type of lists built up from *nil* and *cons*. The purpose of this section is to generalise Horner's rule to arbitrary initial types, and then to illustrate it with a small, familiar programming problem.

First, let us define $\mathit{tri}\, f : \mathit{listr}\, A \leftarrow \mathit{listr}\, A$ formally: we have

$$\mathit{tri}\, f \quad = \quad (\!|\mathit{nil}, \mathit{cons} \cdot (\mathit{id} \times \mathit{listr}\, f)|\!).$$

The base functor of cons-lists is $\mathsf{F}(A, B) = 1 + A \times B$, and the initial algebra

$\alpha = [nil, cons]$, so we can write the above in the form

$$tri\,f \quad = \quad (\!|\alpha \cdot \mathsf{F}(id, listr\,f)|\!).$$

This immediately gives the abstract definition: let F be a bifunctor with initial type (α, T); then

$$tri\,f \quad = \quad (\!|\alpha \cdot \mathsf{F}(id, \mathsf{T}f)|\!).$$

For the definition to make sense we require f to be of type $A \leftarrow A$ for some A, in which case $tri\,f : \mathsf{T}A \leftarrow \mathsf{T}A$. We aim to generalise Horner's rule by finding conditions such that

$$(\!|g|\!) \cdot tri\,f \quad = \quad (\!|g \cdot \mathsf{F}(id, f)|\!).$$

The type information is illustrated in the following diagram:

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{F}(A,A) \arrow[d, "g"'] & \mathsf{T}A \arrow[dr, "(\!|g|\!)"'] & & \mathsf{T}A \arrow[ll, "tri\,f"'] \arrow[dl, "(\!|g \cdot \mathsf{F}(id,f)|\!)"] \\
A & A & & A &
\end{tikzcd}$$

By fusion it suffices to find conditions such that

$$(\!|g|\!) \cdot \alpha \cdot \mathsf{F}(id, \mathsf{T}f) \quad = \quad g \cdot \mathsf{F}(id, f) \cdot \mathsf{F}(id, (\!|g|\!)).$$

We calculate:

$$\begin{array}{cl}
& (\!|g|\!) \cdot \alpha \cdot \mathsf{F}(id, \mathsf{T}f) \\
= & \quad \{\text{catamorphisms}\} \\
& g \cdot \mathsf{F}(id, (\!|g|\!)) \cdot \mathsf{F}(id, \mathsf{T}f) \\
= & \quad \{\mathsf{F}\ \text{bifunctor}\} \\
& g \cdot \mathsf{F}(id, (\!|g|\!) \cdot \mathsf{T}f) \\
= & \quad \{\text{type functor fusion (2.14)}\} \\
& g \cdot \mathsf{F}(id, (\!|g \cdot \mathsf{F}(f, id)|\!)) \\
= & \quad \{\text{claim: see below}\} \\
& g \cdot \mathsf{F}(id, f \cdot (\!|g|\!)) \\
= & \quad \{\mathsf{F}\ \text{bifunctor}\} \\
& g \cdot \mathsf{F}(id, f) \cdot \mathsf{F}(id, (\!|g|\!)).
\end{array}$$

The claim is that $(\!|g \cdot \mathsf{F}(f, id)|\!) = f \cdot (\!|g|\!)$. Appealing to fusion a second time, this equation holds if

$$f \cdot g \quad = \quad g \cdot \mathsf{F}(f, id) \cdot \mathsf{F}(id, f).$$

Since the right-hand side equals $g \cdot \mathsf{F}(f,f)$, we have shown that

$$(\!|g|\!) \cdot \mathit{tri}\, f = (\!|g \cdot \mathsf{F}(\mathit{id},f)|\!) \quad \Leftarrow \quad f \cdot g = g \cdot \mathsf{F}(f,f).$$

For the special case of lists this is precisely the statement of Horner's rule in Ruby.

Depth of a tree

Now consider the problem of computing the depth of a binary tree. We define such trees with the declaration

$$\mathit{tree}\, A \quad ::= \quad \mathit{tip}\, A \mid \mathit{bin}\, (\mathit{tree}\, A, \mathit{tree}\, A).$$

The base functor F of this definition is $\mathsf{F}(A,B) = A + B \times B$, and the initial type of F is $([\mathit{tip}, \mathit{bin}], \mathit{tree})$. We have that $f = (\!|g, h|\!)$ is the unique solution of the equations

$$\begin{aligned} f\,(\mathit{tip}\, a) &= g\, a \\ f\,(\mathit{bin}\,(x,y)) &= h(f\,x, f\,y), \end{aligned}$$

so $(\!|g, h|\!)$ is the generic fold operator for binary trees. In particular, the map operator $\mathit{tree}\, f$ for binary trees is defined by

$$\mathit{tree}\, f \quad = \quad (\!|[\mathit{tip}, \mathit{bin}] \cdot \mathsf{F}(f, \mathit{id})|\!).$$

At the point level, this equation translates into two equations

$$\begin{aligned} \mathit{tree}\, f\,(\mathit{tip}\, a) &= \mathit{tip}\,(f\, a) \\ \mathit{tree}\, f\,(\mathit{bin}\,(x,y)) &= \mathit{bin}\,(\mathit{tree}\, f\, x, \mathit{tree}\, f\, y). \end{aligned}$$

The function $\mathit{max} : N \leftarrow \mathit{tree}\, N$ returns the maximum of a tree of numbers:

$$\mathit{max} \quad = \quad (\!|\mathit{id}, \mathit{bmax}|\!),$$

where $\mathit{bmax}(a,b)$ returns the maximum of a and b. The function $\mathit{depths} : \mathit{tree}\, N \leftarrow \mathit{tree}\, A$ takes a tree and replaces every tip by its depth in the tree:

$$\mathit{depths} \quad = \quad \mathit{tri}\, \mathit{succ} \cdot \mathit{tree}\, \mathit{zero},$$

where zero is the constant function returning 0, and succ is the successor function. Finally, we specify the depth of a tree by

$$\mathit{depth} \quad = \quad \mathit{max} \cdot \mathit{depths}.$$

A direction implementation of depth will require time that is quadratic in the number of tips. For an unbalanced tree of n tips with a single tip at every positive depth, the computation of depths requires evaluation of succ^i for $1 \leq i \leq n$ and this takes $O(n^2)$ steps. We aim to improve the efficiency by applying the generalised statement of Horner's rule to the term $max \cdot tri\ succ$. The proviso of Horner's rule in this case is that

$$succ \cdot [id, bmax] \quad = \quad [id, bmax] \cdot (succ + succ \times succ).$$

Since $succ \cdot id = id \cdot succ$ we require

$$succ \cdot bmax \quad = \quad bmax \cdot (succ \times succ),$$

but this is equivalent to the fact that $succ$ is monotonic. Therefore, we obtain

$$\begin{array}{cl} & depth \\ = & \quad \{\text{definitions}\} \\ & max \cdot tri\ succ \cdot tree\ zero \\ = & \quad \{\text{Horner's rule}\} \\ & (\![[id, bmax] \cdot (id + succ \times succ)]\!) \cdot tree\ zero \\ = & \quad \{\text{coproducts}\} \\ & (\![id, bmax \cdot (succ \times succ)]\!) \cdot tree\ zero \\ = & \quad \{\text{since } bmax \cdot (succ \times succ) = succ \cdot bmax\} \\ & (\![id, succ \cdot bmax]\!) \cdot tree\ zero \\ = & \quad \{\text{type functor fusion}\} \\ & (\![zero, succ \cdot bmax]\!). \end{array}$$

This is the obvious linear-time program for computing the maximum depth of a tree.

The moral of this example is that the categorical proof of familiar laws about lists (such as Horner's rule) are free of the syntactic clutter that a specialised proof would require. Furthermore, the categorically formulated law sometimes applies to programming examples that have nothing to do with lists.

Exercises

3.9 The function $slice :: list\,(listr^+ A) \leftarrow list\,(listr^+ A)$ is given informally by

$$slice\,[x_0, x_1, \ldots, x_{n-1}] \quad = \quad [drop\,0\,x_0, drop\,1\,x_1, \ldots, drop\,(n-1)\,x_{n-1}],$$

where $drop\,n\,x$ drops the first n elements from the list x. Define the function $slice$ in terms of tri.

3.10 The *binary hyperproduct* of a sequence of numbers $[a_0, a_1, \ldots, a_{n-1}]$ is given by $\prod_{i=0}^{n-1} a_i^{2^i}$. Using Horner's rule, derive an efficient program for computing binary hyperproducts.

3.11 Horner's rule can be generalised as follows. If $h \cdot g = g \cdot \mathsf{F}(f, h)$, then

$$(\!|g|\!) \cdot \mathit{tri}\, f \quad = \quad (\!|g \cdot \mathsf{F}(id, h)|\!).$$

Draw a diagram of the types involved and prove the new rule.

3.12 Show that, when the new rule of the preceding exercise is applied to polynomial evaluation, there is only one possible choice for h.

3.13 Specify the problem of computing $\sum_{i=0}^{n-1} ia_i$ in terms of *tri*. Horner's rule is not immediately applicable, but it is if you consider computing $(\sum_{i=0}^{n-1} ia_i, \sum a_i)$ instead. Work out the details of this application.

3.14 Consider binary trees of type

$$\mathit{tree}\ A \quad ::= \quad \mathit{tip}\ A \mid \mathit{node}\,(\mathit{tree}\ A, \mathit{tree}\ A).$$

The *weighted path length* of a tree of numbers is obtained by multiplying each tip by its depth, and then summing the tips. Define a function $\mathit{wpl} : \mathit{Nat} \leftarrow \mathit{tree}\ \mathit{Nat}$ that returns the weighted path length of a tree, using *tri*. Using Horner's rule, improve the efficiency of the definition.

3.3 The TEX problem – part one

The TEX problem (Knuth 1990; Gries 1990a) is to do with converting between binary and decimal numbers in Knuth's text processing system TEX (used to produce this book). TEX uses integer arithmetic, with all fractions expressed as integer multiples of 2^{-16}. Since the input language of TEX documents is decimal, there is the problem of converting between decimal fractions and their nearest representable binary equivalents.

Here, we are interested only in the decimal to binary problem; the converse problem, which is more difficult, will be dealt with in Chapter 10. Let x denote the decimal fraction $0.d_1 d_2 \ldots d_k$ and let

$$\mathit{val}(x) = \sum_{j=1}^{j=k} d_j / 10^j \tag{3.1}$$

be the corresponding real number. The problem is to find the integer multiple of 2^{-16} nearest to $\mathit{val}(x)$, that is, to round $2^{16}\mathit{val}(x)$ to the nearest integer. If two integers are equally near this quantity, we will take the larger; so we want $n = \lfloor 2^{16} val(x) + 1/2 \rfloor$. The value n will lie in the range $0 \leq n \leq 2^{16}$.

So far, so good. But it is required to use integer arithmetic only in the calculation and to keep intermediate results reasonably small, so there is a programming problem to get round.

To formulate (3.1) in programming terms we will need the datatype

$$Decimal \quad ::= \quad nil \mid cons\,(Digit, Decimal).$$

The function $val : Unit \leftarrow Decimal$, where $Unit$ denotes the set of real numbers r in the unit interval $0 \leq r < 1$, is then given by the catamorphism

$$\begin{array}{rcl} val & = & (\![zero, shift]\!) \\ shift\,(d, r) & = & (d + r)/10. \end{array}$$

For example, with $x = [d_1, d_2, d_3]$ we obtain that $val\, x$ is the number

$$(d_1 + (d_2 + (d_3 + 0)/10)/10)/10 \quad = \quad d_1/10 + d_2/100 + d_3/1000.$$

Writing $[0, 2^{16}]$ for the set of integers n in the range $0 \leq n \leq 2^{16}$, our problem is to compute $intern : [0, 2^{16}] \leftarrow Decimal$, where

$$\begin{array}{rcl} intern & = & round \cdot val \\ round\, r & = & \lfloor (2^{17} r + 1)/2 \rfloor, \end{array}$$

under the restriction that only integer arithmetic is allowed.

For completeness, we specify the converse problem, which is to compute a function $extern : Decimal \leftarrow [0, 2^{16})$, where $[0, 2^{16})$ denotes the set of integers n in the range $0 \leq n < 2^{16}$. The function $extern$ is defined by the condition that for all arguments n the value of $extern\, n$ should be a shortest decimal fraction satisfying $intern\,(extern\, n) = n$. We cannot yet formalise this specification, let alone solve the problem, since the definition does not identify a unique decimal fraction, and so $extern$ cannot be described solely within a functional framework. On the other hand, $extern$ can be specified using relations, a point that motivates the remainder of the book.

Let us return to the problem of computing $intern$. Given its definition, it is tempting to try and use the fusion law for catamorphisms, promoting the computation of $round$ into the catamorphism. However, this idea does not quite work. To solve the problem, we need to make use of the following 'rule of floors': for integers a and b, with $b > 0$, and real r we have

$$\lfloor (a + r)/b \rfloor \quad = \quad \lfloor (a + \lfloor r \rfloor)/b \rfloor.$$

Applied to the function *round*, the rule of floors gives that

$$\begin{aligned} \mathit{round} &= \mathit{halve} \cdot \mathit{convert} \\ \mathit{halve}\, n &= (n+1) \operatorname{div} 2 \\ \mathit{convert}\, r &= \lfloor 2^{17} r \rfloor . \end{aligned}$$

This division of *round* into two components turns out to be necessary because, as we shall see, we can apply fusion with *convert* but not with *halve*.

To see if we can apply fusion with *convert*, we calculate:

$$\begin{aligned} & (\mathit{convert} \cdot \mathit{shift})\,(d, r) \\ = \quad & \quad \{\text{definitions of } \mathit{convert} \text{ and } \mathit{shift}\} \\ & \lfloor 2^{17}(d + r)/10 \rfloor \\ = \quad & \quad \{\text{rule of floors, since } 2^{17} d \text{ is an integer}\} \\ & \lfloor (2^{17} d + \lfloor 2^{17} r \rfloor)/10 \rfloor \\ = \quad & \quad \{\text{definition of } \mathit{convert}\} \\ & \lfloor (2^{17} d + \mathit{convert}(r))/10 \rfloor \\ = \quad & \quad \{\text{introducing } \mathit{cshift}\text{; see below}\} \\ & \mathit{cshift}(d, \mathit{convert}(r)), \end{aligned}$$

where $\mathit{cshift}(d, n) = (2^{17} d + n) \operatorname{div} 10$. Since we also have $\mathit{convert}(0) = 0$, we now obtain

$$\mathit{convert} \cdot [\mathit{zero}, \mathit{shift}] \quad = \quad [\mathit{zero}, \mathit{cshift}] \cdot (\mathit{id} + (\mathit{id} \times \mathit{convert})),$$

and hence, by fusion, $\mathit{intern} = \mathit{halve} \cdot (\![\mathit{zero}, \mathit{cshift}]\!)$. This concludes the derivation.

Two further remarks are in order. The first is a small calculation to show that the expression $\mathit{halve} \cdot (\![\mathit{zero}, \mathit{cshift}]\!)$ cannot be optimised by a second appeal to fusion. We have

$$\begin{aligned} & (\mathit{halve} \cdot \mathit{cshift})\,(d, n) \\ = \quad & \quad \{\text{definitions of } \mathit{halve} \text{ and } \mathit{cshift}\} \\ & \lfloor (\lfloor (2^{17} d + n)/10 \rfloor + 1)/2 \rfloor \\ = \quad & \quad \{\text{arithmetic}\} \\ & \lfloor (2^{17} d + n + 10)/20 \rfloor \end{aligned}$$

Now, in order to appeal to fusion, we have to write this last expression in the form $f\,(d, \mathit{halve}\, n)$ for some function f. Since $\mathit{halve}(2k) = \mathit{halve}(2k-1)$ for all $k > 0$, we therefore require that

$$f\,(d, \mathit{halve}\,(2k)) \quad = \quad f\,(d, \mathit{halve}\,(2k-1)).$$

In other words, we need

$$\lfloor (2^{17}d + 2k + 10)/20 \rfloor \quad = \quad \lfloor (2^{17}d + 2k + 9)/20 \rfloor$$

for all $k > 0$. But, taking $d = 0$ and $k = 5$, this gives $1 = 0$, so no function f can exist and the attempt to use fusion a second time fails.

The second remark concerns the fact that nowhere above have we exploited any property of 2^{17} except that it was a non-negative integer. For the particular value 2^{17}, the algorithm can be optimised: except for the first 17, all digits of the given decimal can be discarded since they do not affect the answer. A proof of this fact can be found in (Knuth 1990).

Exercises

3.15 Taking *Decimal* = *listr Digit* (why is it valid to do so?), the function *val* could be specified

$$val \quad = \quad sum \cdot tri\,(/10) \cdot listr\,(/10).$$

Derive the catamorphism in the text.

3.16 Supposing we take 2^2 rather than 2^{16}, characterise those decimals whose *intern* values are n, for $0 \leq n \leq 4$.

3.17 Show that $intern = intern \cdot take\,17$.

3.18 The rule of *indirect equality* states that two integers m and n are equal iff

$$k \leq m \quad \equiv \quad k \leq n \;, \qquad \text{for all } k.$$

Prove the rule of indirect equality. Can you generalise the rule to arbitrary ordered sets?

3.19 The *floor* of a real number x is defined by the property that, for all integers n,

$$n \leq x \quad \equiv \quad n \leq \lfloor x \rfloor.$$

Prove the rule of floors using this definition and the rule of indirect equality.

3.20 Show that the rule of floors is not valid when a or b is not an integer.

3.21 Show that if $f : A \leftarrow B$ is injective, then for any binary operator $(\oplus) : B \leftarrow C \times B$ there exists a binary operator $(\otimes) : A \leftarrow A \times C$ such that

$$f\,(c \oplus b) \quad = \quad c \otimes f\,b.$$

(*Cf.* Exercise 2.34.)

3.22 Let $f : A \leftarrow B$ and $(\oplus) : B \leftarrow C \times B$. To prove that there exists no binary operator $(\otimes) : A \leftarrow C \times A$ such that

$$f\,(c \oplus b) \quad = \quad c \otimes f\, b,$$

it suffices to find c, b_0 and b_1 such that

$$f\, b_0 = f\, b_1 \qquad \text{and} \qquad f(c \oplus b_0) \neq f(c \oplus b_1).$$

Apply this strategy to prove that fusion does *not* apply to $round \cdot val$.

3.4 Conditions and conditionals

We have already shown how many features of current functional programming languages can be expressed and characterised in a purely categorical setting. But there are two important omissions: definition by cases and currying. Currying will be dealt with in the following section; here we are concerned with how to characterise definition by cases.

The coproduct construction permits a restricted kind of definition by cases, essentially definition by pattern-matching. As we have seen, this is sufficient for the description of many functions. However, programmers also make use of conditionals; for example, the function *filter p* is defined using a mixture of pattern-matching and case analysis:

$$\begin{aligned} \mathit{filter}\, p\,[\,] &= [\,] \\ \mathit{filter}\, p\,(\mathit{cons}\,(a, x)) &= \begin{cases} \mathit{cons}(a, \mathit{filter}\, p\, x), & \text{if } p\, a \\ \mathit{filter}\, p\, x, & \text{otherwise.} \end{cases} \end{aligned}$$

Given the McCarthy conditional form $(p \to f, g)$ for writing conditionals, we can express *filter p* as a catamorphism on cons-lists:

$$\begin{aligned} \mathit{filter}\, p &= (\![\mathit{nil}, \mathit{test}\, p]\!) \\ \mathit{test}\, p &= (p \cdot \mathit{outl} \to \mathit{cons}, \mathit{outr}). \end{aligned}$$

The question thus arises: how can we express and characterise the conditional form $(p \to f, g)$ in a categorical setting?

In functional programming the datatype *Bool* is declared by

$$\mathit{Bool} \;::=\; \mathit{true} \mid \mathit{false}.$$

Thus, $\mathit{Bool} = 1 + 1$, with injection functions $\mathit{inl} = \mathit{true}$ and $\mathit{inr} = \mathit{false}$. Using this datatype, we can define the function $\mathit{not} : \mathit{Bool} \leftarrow \mathit{Bool}$ by

$$\mathit{not} \quad = \quad [\mathit{false}, \mathit{true}].$$

The negation of a condition $p : Bool \leftarrow A$ can now be defined as $not \cdot p$. Although this is straightforward enough, the construction of binary operators such as *and* and *or* is a little more problematic. As we shall see, we need the assumption that the underlying category is *distributive*. In a distributive category one can also construct conditionals.

Distributive categories

In any category with products and coproducts there is a natural transformation

$$undistr : A \times (B + C) \leftarrow (A \times B) + (A \times C)$$

defined by $undistr = [id \times inl, id \times inr]$ (*undistr* is short for 'un-distribute-right'). Thus,

$$(f \times (g + h)) \cdot undistr \quad = \quad undistr \cdot ((f \times g) + (f \times h))$$

for all f, g and h of the appropriate types. In a distributive category *undistr* is, by assumption, a natural *isomorphism*. This means that there is an arrow

$$distr \quad : \quad (A \times B) + (A \times C) \leftarrow A \times (B + C)$$

such that $distr \cdot undistr = id$ and $undistr \cdot distr = id$.

There is a second requirement on a distributive category. In any category with products and initial objects, there is a (unique) arrow

$$unnull : A \times 0 \leftarrow 0$$

for each A. In a distributive category *unnull*, like *undistr*, is assumed to be an isomorphism. Thus, there is an arrow $null : 0 \leftarrow A \times 0$ such that $null \cdot unnull = id$ and $unnull \cdot null = id$.

In other words, in a distributive category we have the natural isomorphisms

$$\begin{aligned} A \times (B + C) &\cong (A \times B) + (A \times C) \\ A \times 0 &\cong 0, \end{aligned}$$

as well as the natural isomorphisms

$$\begin{aligned} A \times (B \times C) \cong (A \times B) \times C &\qquad A + (B + C) \cong (A + B) + C \\ A \times B \cong B \times A &\qquad A + B \cong B + A \\ A \times 1 \cong A &\qquad A + 0 \cong A, \end{aligned}$$

described in Exercise 2.26. Below we shall sometimes omit brackets in products and coproducts that consist of more than two components.

One consequence of a category being distributive is that there are non-trivial arrows whose source is a product, the trivial arrows being the identity arrow and the projections. In particular, there is an isomorphism

$$quad : 1 + 1 + 1 + 1 \leftarrow Bool^2.$$

We will leave the proof as an exercise. It follows that we can define arrows of type $Bool \leftarrow Bool^2$ in terms of arrows of type $Bool \leftarrow 1 + 1 + 1 + 1$. For example, we can define

$$and \quad = \quad [true, false, false, false] \cdot quad$$

and the conjunction of $p, q : Bool \leftarrow A$ by $and \cdot \langle p, q \rangle$. Other boolean connectives can also be defined by such 'truth tables'.

A distributive category also gives us the means to construct, given a function $p : Bool \leftarrow A$ and two functions $f, g : B \leftarrow A$, a conditional function $(p \rightarrow f, g) : B \leftarrow A$. The idea is to associate with each condition $p : Bool \leftarrow A$ an arrow $p? : A + A \leftarrow A$, for then we can define

$$(p \rightarrow f, g) \quad = \quad [f, g] \cdot p?. \tag{3.2}$$

The arrow $p?$ is defined by

$$p? \quad = \quad (unit + unit) \cdot distr \cdot \langle id, p \rangle.$$

The types are shown by the following diagram:

$$\begin{tikzcd}
A \arrow[r, "\langle id, p \rangle"] \arrow[d, "p?"'] & A \times Bool \arrow[d, "distr"] \\
A + A & A \times 1 + A \times 1 \arrow[l, "unit + unit"]
\end{tikzcd}$$

The association of conditions p with arrows $p?$ is injective (see Exercise 3.25). Using definition (3.2), let us now show that the following three properties of conditionals hold:

$$h \cdot (p \rightarrow f, g) \quad = \quad (p \rightarrow h \cdot f, h \cdot g) \tag{3.3}$$
$$(p \rightarrow f, g) \cdot h \quad = \quad (p \cdot h \rightarrow f \cdot h, g \cdot h) \tag{3.4}$$
$$(p \rightarrow f, f) \quad = \quad f. \tag{3.5}$$

Equation (3.3) is immediate from (3.2) using the distributivity property of coproducts. For (3.4) it is sufficient to show

$$(h + h) \cdot (p \cdot h)? \quad = \quad p? \cdot h.$$

The proof is:

$$\begin{array}{cl}
& (h+h)\cdot(p\cdot h)? \\
= & \quad \{\text{definition}\} \\
& (h+h)\cdot(unit+unit)\cdot distr\cdot\langle id, p\cdot h\rangle \\
= & \quad \{\text{naturality of } distr \text{ and } unit\} \\
& (unit+unit)\cdot distr\cdot(h\times id)\cdot\langle id, p\cdot h\rangle \\
= & \quad \{\text{products}\} \\
& (unit+unit)\cdot distr\cdot\langle id, p\rangle\cdot h \\
= & \quad \{\text{definition}\} \\
& p?\cdot h.
\end{array}$$

For (3.5) it is sufficient to show that $[f,f]\cdot p? = f$. We argue:

$$\begin{array}{cl}
& [f,f]\cdot p? \\
= & \quad \{\text{definition}\} \\
& [f,f]\cdot(unit+unit)\cdot distr\cdot\langle id, p\rangle \\
= & \quad \{\text{coproducts; naturality of } unit\} \\
& unit\cdot[f\times id, f\times id]\cdot distr\cdot\langle id, p\rangle \\
= & \quad \{\text{claim: see below}\} \\
& unit\cdot(f\times id)\cdot\langle id, p\rangle \\
= & \quad \{\text{products}\} \\
& unit\cdot\langle f, p\rangle \\
= & \quad \{\text{since } unit = outl\} \\
& f.
\end{array}$$

The claim is an instance of the fact that

$$f\times[g,h] \quad = \quad [f\times g, f\times h]\cdot distr.$$

The proof, which we leave as a short exercise, uses the definition of *undistr* and the fact that $undistr\cdot distr = id$.

Exercises

3.23 Show that if

$$\begin{tikzcd}
A & & A\times 0 \arrow[ll, "outl"'] \arrow[dl, "outr"] \\
& 0 \arrow[ul, "¡"] &
\end{tikzcd}$$

commutes, then there must exist an arrow *unnull* such that $unnull\cdot null = id$ and $null\cdot unnull = id$:

3.24 Is *Rel* a distributive category?

3.25 Prove that $(\_)?$ is injective with inverse $(\_)¿$ defined by

$$t¿ \quad = \quad (! + !) \cdot t.$$

Hint: first show that $(! + !) \cdot \mathit{distr} = (! + !) \cdot \mathit{outr}$.

3.26 Prove that

$$(\mathit{unit} + \mathit{unit}) \cdot \mathit{distr} \quad : \quad \mathsf{F} \leftarrow \mathsf{G},$$

where $\mathsf{F}A = A + A$ and $\mathsf{G}A = A \times \mathit{Bool}$.

3.27 Suppose in a distributive category that there is an arrow $h : 0 \leftarrow A$. Show that h is an isomorphism and hence that A is also an initial object.

3.28 Prove that $f \times [g, h] = [f \times g, f \times h] \cdot \mathit{distr}$.

3.29 Show that $\mathit{Bool}^2 \cong 1 + 1 + 1 + 1$.

3.30 Prove that $\mathit{filter}\, p \cdot \mathit{listr}\, f = \mathit{listr}\, f \cdot \mathit{filter}\,(p \cdot f)$ using the following definition of *filter*:

$$\mathit{filter}\, p \quad = \quad \mathit{concat} \cdot \mathit{listr}\,(p \rightarrow \mathit{wrap}, \mathit{nil}),$$

where $\mathit{wrap}\, a = [a]$ and $\mathit{nil} : \mathit{listr}\, A \leftarrow A$ is a constant returning the empty list.

3.5 Concatenation and currying

Consider once more the type $\mathit{listr}\, A$ of cons-lists over A. In functional programming the function $\mathit{cat} : \mathit{listr}\, A \leftarrow \mathit{listr}\, A \times \mathit{listr}\, A$ is written as an infix operator $+\!\!+$ and defined by the equations

$$\begin{aligned} [\,] +\!\!+ y &= y \\ \mathit{cons}\,(a, x) +\!\!+ y &= \mathit{cons}\,(a, x +\!\!+ y). \end{aligned}$$

In terms of our categorical combinators these equations become

$$\begin{aligned} \mathit{cat} \cdot (\mathit{nil} \times \mathit{id}) &= \mathit{outr} \\ \mathit{cat} \cdot (\mathit{cons} \times \mathit{id}) &= \mathit{cons} \cdot (\mathit{id} \times \mathit{cat}) \cdot \mathit{assocr}, \end{aligned}$$

where *assocr* is the natural isomorphism $\mathit{assocr} : A \times (B \times C) \leftarrow (A \times B) \times C$ described in Exercise 2.26. We can combine the two equations for *cat* into one:

$$\mathit{cat} \cdot ([\mathit{nil}, \mathit{cons}] \times \mathit{id}) \quad = \quad [\mathit{outr}, \mathit{cons}] \cdot (\mathit{id} + \mathit{id} \times \mathit{cat}) \cdot \phi, \tag{3.6}$$

where $\phi : (1 \times C) + A \times (B \times C) \leftarrow (1 + A \times B) \times C$ is given by

$$\phi \quad = \quad (id + assocr) \cdot distl$$

and *distl* is the natural isomorphism $(A \times C) + (B \times C) \leftarrow (A + B) \times C$ whose companion *distr* was described in the preceding section.

But how do we know that equation (3.6) defines *cat* uniquely? The function *cat* is not a catamorphism, in spite of its name, because it has two arguments, so we cannot appeal to the unique solution property of catamorphisms.

The answer is to consider a variant *ccat* of *cat* in which the arguments are curried. Suppose we define $ccat : (listr\ A \leftarrow listr\ A) \leftarrow listr\ A$ by $ccat\ x\ y = x +\!\!+ y$. Then we have

$$\begin{aligned} ccat\,[\,] &= id \\ ccat\,(cons\,(a, x)) &= ccons\ a \cdot ccat\ x, \end{aligned}$$

where we take $ccons : (listr\ A \leftarrow listr\ A) \leftarrow A$ to be a curried version of *cons*. This version of *cat* is a catamorphism, for we have

$$ccat \quad = \quad (\!|const\ id, compose \cdot (ccons \times id)|\!),$$

where $const\,f$ is a constant returning f and $compose(f, g) = f \cdot g$. Just to check this, we expand the catamorphism to two equations:

$$\begin{aligned} ccat \cdot nil &= const\ id \\ ccat \cdot cons &= compose \cdot (ccons \times ccat). \end{aligned}$$

Applying the first equation to the element of the terminal object, and the second to (a, x), we obtain the pointwise versions

$$\begin{aligned} ccat\ nil &= id \\ ccat\,(cons\,(a, x)) &= compose\,(ccons\ a, ccat\ x), \end{aligned}$$

which is what we had before. The conclusion is that since the curried version of *cat* is uniquely defined by this translation, the original version is, too.

All this leads to a more general problem: consider a functor F with initial type (α, T), another functor G, and a transformation $\phi_{A,B} : \mathsf{G}(A \times B) \leftarrow \mathsf{F}A \times B$. What conditions on ϕ guarantee that the recursion

$$f \cdot (\alpha \times id) \quad = \quad h \cdot \mathsf{G}f \cdot \phi \tag{3.7}$$

defines a unique function f for each choice of h?

In a diagram we have

$$\begin{tikzcd} \mathsf{T}A \times B \arrow[d, "f"'] & \mathsf{FT}A \times B \arrow[l, "\alpha \times id"'] \arrow[r, "\phi"] & \mathsf{G}(\mathsf{T}A \times B) \arrow[d, "\mathsf{G}f"] \\ B & & \mathsf{G}B \arrow[ll, "h"] \end{tikzcd}$$

To solve the problem we use the same idea as before and curry the function f. To do this we need the idea of a function space object $A \leftarrow B$, more usually written in the form A^B. Function space objects are called *exponentials.*

Exponentials

Let $\mathbf{C}$ be a category with terminal object and products. An *exponential* of two objects A and B is an object A^B and an arrow $apply : A \leftarrow A^B \times B$ such that for each $f : A \leftarrow C \times B$ there is a unique arrow $curry\, f : A^B \leftarrow C$ such that

$$apply \cdot (curry\, f \times id) \quad = \quad f.$$

In other words, we have the universal property

$$g = curry\, f \quad \equiv \quad apply \cdot (g \times id) = f.$$

For fixed A and B, this definition can be regarded as defining a terminal object in the category **Exp**, constructed as follows. The objects of **Exp** are arrows $A \leftarrow C \times B$ in $\mathbf{C}$. An arrow $h \leftarrow k$ of **Exp** is an arrow $f : C \times B \leftarrow D \times B$ of $\mathbf{C}$ just when the following diagram commutes:

$$\begin{tikzcd} C \times B \arrow[dr, "h"'] & & D \times B \arrow[ll, "f \times id"'] \arrow[dl, "k"] \\ & A & \end{tikzcd}$$

The terminal object of **Exp** is $apply : A \leftarrow A^B \times B$, and $!_f$ is given by $curry\, f$. The reflection law of terminal objects translates in this case to

$$curry\; apply \quad = \quad id,$$

and the fusion law reads

$$curry\, f \cdot g = curry\,(f \cdot (g \times id)).$$

If a category has finite products, and for every pair of objects A and B the exponential A^B exists, the category is said to be *cartesian closed.* In what follows, we assume that we are working in a cartesian closed category.

Returning to the problem of solving equation (3.7), the fusion law gives us that

$$f \cdot (\alpha \times id) = h \cdot \mathsf{G} f \cdot \phi \quad \equiv \quad curry\, f \cdot \alpha = curry\,(h \cdot \mathsf{G} f \cdot \phi).$$

Our aim now is to find a k so that

$$curry\,(h \cdot \mathsf{G} f \cdot \phi) \quad = \quad k \cdot \mathsf{F}(curry\, f),$$

in which case we obtain $curry\, f = (\![k]\!)$. We reason:

$$\begin{aligned}
& curry\,(h \cdot \mathsf{G} f \cdot \phi) \\
= & \quad \{\text{curry cancellation}\} \\
& curry\,(h \cdot \mathsf{G}\,(apply \cdot (curry\, f \times id)) \cdot \phi) \\
= & \quad \{\text{functor}\} \\
& curry\,(h \cdot \mathsf{G}\, apply \cdot \mathsf{G}\,(curry\, f \times id) \cdot \phi) \\
= & \quad \{\text{assumption: } \phi \text{ natural}\} \\
& curry\,(h \cdot \mathsf{G}\, apply \cdot \phi \cdot (\mathsf{F}(curry\, f) \times id)) \\
= & \quad \{\text{curry fusion law (backwards)}\} \\
& curry\,(h \cdot \mathsf{G}\, apply \cdot \phi) \cdot \mathsf{F}(curry\, f).
\end{aligned}$$

Hence we can take $k = curry\,(h \cdot \mathsf{G}\, apply \cdot \phi)$. The only assumption in the argument above was that ϕ is natural in the following sense:

$$\mathsf{G}(h \times id) \cdot \phi \quad = \quad \phi \cdot (\mathsf{F}h \times id).$$

In summary, we have proved the following structural recursion theorem.

Theorem 3.1 If ϕ is natural in the sense that $\mathsf{G}(h \times id) \cdot \phi = \phi \cdot (\mathsf{F}h \times id)$, then

$$f \cdot (\alpha \times id) \quad = \quad h \cdot \mathsf{G}f \cdot \phi$$

if and only if

$$f \quad = \quad apply \cdot ((\![curry\,(h \cdot \mathsf{G}apply \cdot \phi)]\!) \times id).$$

Let us now see what this gives in the case of cat. We started with

$$cat \cdot (\alpha \times id) \quad = \quad [outr, cons] \cdot (id + id \times cat) \cdot \phi,$$

where $\phi = (id + assocr) \cdot distl$. So, $h = [outr, cons]$ and $\mathsf{G}f = (id + id \times f)$. The naturality condition on ϕ is

$$(id + id \times (h \times id)) \cdot \phi \quad = \quad \phi \cdot ((id + (id \times h) \times id)),$$

which is easily checked. Hence we find that

$$cat \quad = \quad (\!|curry\,([outr, cons] \cdot (id + id \times apply) \cdot (id + assocr) \cdot distl)|\!),$$

which simplifies to

$$cat \quad = \quad (\!|curry\,([outr, cons \cdot (id \times apply) \cdot assocr] \cdot distl)|\!).$$

We leave it as an instructive exercise to recover the pointwise definition of *cat* from the above catamorphism.

Tree traversal

Let us look at another illustration. Consider again the initial type of trees introduced in Section 3.2. The function *tips* returns the list of tips of a given tree:

$$tips \quad = \quad (\!|wrap, cat|\!).$$

Here $cat\,(x, y) = x +\!\!+\, y$ is the concatenation function on lists from the last section, and *wrap* is the function that converts an element into a singleton list, so $wrap\ a = [a]$. In most functional languages, the computation of $x +\!\!+\, y$ takes time proportional to the length of x. Therefore, when we attempt to implement the above definition directly in such a language, the result is a quadratic-time program.

To improve the efficiency, we aim to design a curried function *tipcat* such that

$$tipcat\ t\ x \quad = \quad tips\ t +\!\!+\, x.$$

Since the empty list is the unit of concatenation we have $tips\ t = tipcat\ t\,[\,]$, so *tipcat* is a generalisation of our problem. The addition of an extra parameter such as x is known as *accumulation* and is a well-known technique for improving the efficiency of functional programs.

Using *curry*, we can write the above definition of *tipcat* more briefly as

$$tipcat \quad = \quad curry\ cat \cdot tips.$$

This suggests an application of the fusion law. Can we find an f and op so that both of the following equations hold?

$$\begin{aligned} curry\ cat \cdot wrap \quad &= \quad f \\ curry\ cat \cdot cat \quad &= \quad op \cdot (curry\ cat \times curry\ cat) \end{aligned}$$

Well, since $cons\,(a, x) \ = \ cat([a], x)$, we can take $f \ = \ curry\ cons$. To find op we reason as follows:

$$
\begin{array}{cl}
& (\mathit{curry\ cat} \cdot \mathit{cat})\,(x, y)\,z \\
= & \quad \{\text{application}\} \\
& (x +\!\!+\, y) +\!\!+\, z \\
= & \quad \{\text{since } (+\!\!+) \text{ is associative}\} \\
& (x +\!\!+\, (y +\!\!+\, z)) \\
= & \quad \{\text{application}\} \\
& (\mathit{curry\ cat}\ x \cdot \mathit{curry\ cat}\ y)\,z \\
= & \quad \{\text{introducing } \mathit{compose}\,(h, k) = h \cdot k\} \\
& (\mathit{compose} \cdot (\mathit{curry\ cat} \times \mathit{curry\ cat}))\,(x, y)\,z.
\end{array}
$$

Hence we have

$$
\mathit{tips}\ t \;=\; (\!|\mathit{curry\ cons}, \mathit{compose}|\!)\ t\ \mathit{nil}.
$$

In contrast to the original definition of *tips*, this equation can be implemented directly as a linear-time program.

Exercises

3.31 Show that

$$
\begin{array}{rcl}
\mathit{cat} \cdot (\mathit{nil} \times \mathit{id}) & = & \mathit{outr} \\
\mathit{cat} \cdot (\mathit{cons} \times \mathit{id}) & = & \mathit{cons} \cdot (\mathit{id} \times \mathit{cat}) \cdot \mathit{assocr}
\end{array}
$$

is equivalent to equation (3.6), using properties of products and coproducts only.

3.32 Prove that any cartesian closed category that has coproducts is distributive.

3.33 Construct the following isomorphisms:

$$
A^0 \cong 1 \quad A^1 \cong A \quad A^{B+C} \cong A^B \times A^C.
$$

3.34 Construct a bijection between arrows of type $A \leftarrow B$ and arrows of type $A^B \leftarrow 1$.

3.35 What does it mean for a preorder to be cartesian closed? (See Exercise 2.6 for the interpretation of preorders as categories.)

3.36 Let B be an object in a cartesian closed category. Show how $(\_)^B$ can be made into a functor by defining f^B for an arbitrary arrow f.

3.37 Show that if $\mathbf{A}$ is cartesian closed, then so is $\mathbf{A}^{\mathbf{B}}$. (See Exercise 2.19 for the definition of $\mathbf{A}^{\mathbf{B}}$.)

3.38 The *map* function (as in functional programming) is a collection of arrows

$$map_{A,B} \quad : \quad listr\, A^{listr\, B} \leftarrow A^B$$

such that $map_{A,B} f = listr\, f$. Between what functors is *map* a natural transformation. Write out the naturality condition and prove that it is satisfied.

3.39 The function *cpr* (short for 'cartesian product, right') with type

$$cpr \quad : \quad listr\,(A \times B) \leftarrow A \times listr\, B$$

is defined by the list comprehension

$$cpr\,(x, b) \quad = \quad [(a, b) \mid a \leftarrow x].$$

Give a point-free definition of *cpr* in terms of *listr*.

3.40 A functor F is said to be *strong* if there exists a corresponding natural transformation

$$map_{A,B} \quad : \quad \mathsf{F}A^{\mathsf{F}B} \leftarrow A^B\ .$$

Show that every functor of *Fun* is strong. Give an example of a functor that is *not* strong. (*Warning:* in the literature, strength usually involves a number of additional conditions. Interested readers should consult the references at the end of this chapter.)

3.41 What conditions guarantee that

$$f \cdot (id \times \alpha) \quad = \quad h \cdot \mathsf{G}f \cdot \phi$$

has a unique solution for each choice of h?

3.42 Show that the following equations uniquely determine $iter\,(g, h) : A \leftarrow (Nat \times B)$, for each choice of $g : A \leftarrow B$ and $h : A \leftarrow A$:

$$\begin{aligned} iter\,(g, h) \cdot (zero \times id) &= g \cdot outr \\ iter\,(g, h) \cdot (succ \times id) &= h \cdot iter\, g\, h. \end{aligned}$$

How can addition be expressed in terms of *iter*?

3.43 Continuing the preceding exercise, show that

$$\begin{tikzcd} Nat \times A \arrow[d, "iter\,(id,h)"'] & & Nat \times (Nat \times A)) \arrow[ll, "id \times iter\,(id,h)"'] \arrow[d, "assocl"] \\ Nat & Nat \times A \arrow[l, "iter\,(id,h)"] & (Nat \times Nat) \times A \arrow[l, "plus \times id"] \end{tikzcd}$$

commutes for all $h : A \leftarrow A$.

3.44 Consider the type definition

$$\mathit{tree}\,A \quad ::= \quad \mathit{tip}\,A \mid \mathit{node}\,(\mathit{tree}\,A)^A$$

Does this definition make sense in *Fun*? Could you write it in your favourite functional programming language?

3.45 The introduction of an *accumulation parameter* in the tree traversal example can be summarised as follows. Suppose that we have a function k and a value e such that $k\;a\;e = a$ (all a) and $k \cdot f = g \cdot Fk$. Then for all x, we have $(\!|f|\!)\;x = (\!|g|\!)\;x\;e$. Prove this general statement. The following four exercises aim to apply this strategy to other examples.

3.46 Recall the function $\mathit{convert} : \mathit{listr}\,A \leftarrow \mathit{listl}\,A$ which produces the *cons*-list corresponding to a given *snoc* list. It is defined by

$$\mathit{convert} \quad = \quad (\!|\mathit{nil}, \mathit{snocr}|\!),$$

where $\mathit{snocr}\,(x, a) = x \mathbin{+\!\!+} [a]$. Improve the efficiency of *convert* by introducing an accumulation parameter.

3.47 Using the type of cons-lists, define

$$\mathit{reverse} \quad = \quad (\!|\mathit{nil}, \mathit{snocr}|\!),$$

where *snocr* was defined above. Improve the efficiency of *reverse* by introducing an accumulation parameter.

3.48 The function *depths*, as defined in terms of *tri*, takes quadratic time. Derive a linear-time implementation by introducing an accumulation parameter. *Hint:* take $k\;a\;n = \mathit{tree}\,(+n)\;a$, and $e = 0$.

3.49 In analogy with the depth of a tree example, we can also define the minimum depth, and the minimum depth can be written as a catamorphism. Direct evaluation of the catamorphism is inefficient because it will explore subtrees all the way down to the tips, even if it has already found a tip at a lesser depth. Improve the efficiency by introducing an accumulation parameter. *Hint:* take $k\;a\,(n, m) = \mathit{min}(a + n, m)$ and $e = (0, \infty)$.

3.50 Consider the recursion scheme:

$$\mathit{loop}\,h \cdot (\alpha \times \mathit{id}) \quad = \quad [\mathit{id}, \mathit{loop}\,h \cdot (\mathit{id} \times h) \cdot \mathit{assocr}] \cdot \mathit{distl},$$

where $\alpha = [\mathit{nil}, \mathit{snoc}]$. Show that for any choice of h the function $\mathit{loop}\,h$ is determined uniquely.

3.51 Using the preceding exercise and Exercise 3.46, check that $convcat\, x\, y = convert\, x +\!\!\!+\, y$ satisfies the equation

$$uncurry\ convcat \quad = \quad loop\ cons.$$

Hence show how cons-list catamorphisms can be implemented on snoc-lists by

$$(\!|e, f|\!) \cdot convert \quad = \quad loop\, f \cdot \langle id, e \cdot ! \rangle.$$

How can snoc-list catamorphisms be implemented by a loop over cons-lists?

Bibliographical remarks

The banana-split law was first recorded by (Fokkinga 1992a), who attributes its catchy name to Meertens and Van der Woude. Of course, similar transformations have been studied in other contexts; they are usually classified under the names *tupling* (Pettorossi 1984) or *parallel loop fusion.*

Our exposition on Horner's rule in Ruby does not do justice either to Ruby or to the use of this particular rule. We entirely ignored several important aspects of Ruby, partly because we can only introduce these once relations have been discussed. The standard introduction to Ruby is (Jones and Sheeran 1990). Other references are (Jones and Sheeran 1993; Hutton 1992; Sheeran 1987, 1990). (Harrison 1991) describes a categorical approach to the synthesis of static parallel algorithms which is very similar to the theory described here, and is also similar to Ruby. (Skillicorn 1995) considers the categorical view of datatypes as an appropriate setting for reasoning about architecture-independent parallel programs. In (Gibbons, Cai, and Skillicorn 1994), some parallel tree-based algorithms are discussed.

Distributive categories are the subject of new and exciting developments on the border between computation and category theory. The exposition given here was heavily influenced by the text (Walters 1992a), as well as by a number of research papers (Carboni, Lack, and Walters 1993; Cockett 1993; Walters 1989, 1992b). The connection between distributive categories and the algebra of conditionals was definitively explored by Robin Cockett (Cockett 1991).

The subject of cartesian closed categories is rich and full of deep connections to computation. Almost all introductory books on category theory mention cartesian closed categories; the most comprehensive treatment can be found in (Lambek and Scott 1986). The trick of using currying to define such operations as concatenation in terms of catamorphisms goes at least back to Lawvere's Recursion theorem for natural numbers: see *e.g.* (Lambek and Scott 1986). In (Cockett 1990; Cockett and Spencer 1992) it is considered how the same effect can be achieved in categories that do not have exponentials. The key is to concentrate on functors that have *tensorial strength*, that is a natural transformation $\theta : \mathsf{F}(A \times B) \leftarrow \mathsf{F}A \times B$ satisfying certain coherence conditions. For more information on strength, the interested reader is also referred to (Kock 1972; Moggi 1991). Some interesting applications of these categorical concepts to programming languages and program construction can be found in (Jay 1994; Jay and Cockett 1994; Jay 1995).

The interplay between fusion and accumulation parameters was first studied in (Bird 1984). Our appreciation of the connection with currying grew while reading (Meijer 1992; Meijer and Hutton 1995), and by discussions with Masato Takeichi and his colleagues at Tokyo University (Hu, Iwasaki, and Takeichi 1996).

Chapter 4

Relations and Allegories

We now generalise from functions to relations. There are a number of reasons for this step. First, like the move from real numbers to complex ones, the move to relations increases our powers of expression. Relations, unlike functions, are essentially nondeterministic and one can employ them to specify nondeterministic problems. For instance, an optimisation problem can be specified in terms of finding an optimal solution among a set of candidates without also having to specify precisely which one should be chosen. Every relation has a well-defined converse, so one can specify problems in terms of converses of other problems.

A second reason concerns the structure of certain proofs. There are deterministically specified programming problems with deterministic solutions where, nevertheless, it is helpful to consider nondeterministic expressions in passing from the former to the latter. The proofs become easier, more structure is revealed, and directions for useful generalisation are clearly signposted. So it is with problems about functions of real variables that are solved more easily in the complex plane.

On the other hand, in the hundred years or so of its existence, the calculus of relations has gained a good deal of notoriety for the apparently enormous number of operators and laws that one has to memorise in order to do proofs effectively. In this chapter we aim to cope with this problem by presenting the calculus in five successive stages, each of which is motivated by categorical considerations and is sufficiently small to be studied as a unit. We will see how these parts interact, and how they can be put to use in developing a concise and effective style of reasoning.

4.1 Allegories

Allegories are to the algebra of relations as categories are to the algebra of functions. An *allegory* **A** is a category endowed with three operators in addition to target, source, composition and identities. These extra operators are inspired by the category **Rel** of sets and relations. Briefly, we can compare relations with a partial order $\subseteq$, take the intersection of two relations with $\cap$, and take a relation to its converse with the unary operator $(\_)^\circ$. The purpose of this section is to describe these operators axiomatically.

Inclusion

The first assumption is that any two arrows with the same source and target can be compared with a partial order $\subseteq$, and that composition is monotonic with respect to this order: that is,

$$(S_1 \subseteq S_2) \text{ and } (T_1 \subseteq T_2) \quad \text{implies} \quad (S_1 \cdot T_1) \subseteq (S_2 \cdot T_2).$$

In **Rel**, where a relation $R : A \leftarrow B$ is interpreted as a subset $R \subseteq A \times B$, inclusion of relations is the same as set-theoretic inclusion; thus

$$R \subseteq S \quad \equiv \quad (\forall a, b : aRb \Rightarrow aSb).$$

Monotonicity of composition is so fundamental that we often apply it tacitly in proofs. An expression of the form $S \subseteq T$ is called an *inequation*, and most of the laws in the relational calculus are inequations rather than equations. The proof format used in the preceding chapter adapts easily to reasoning about inequations, as long as we don't mix reasoning with $\subseteq$ and reasoning with $\supseteq$. A proof of $R = S$ by two separate proofs, one of $R \subseteq S$ and one of $S \subseteq R$, is sometimes called a *ping-pong* argument. Use of ping-pong arguments can often be avoided either by direct equational reasoning, or by an *indirect* proof in which the following equivalence is exploited:

$$R = S \quad \equiv \quad (X \subseteq R \ \equiv \ X \subseteq S) \qquad \text{for all } X.$$

Thus, an indirect proof is equational reasoning with $\equiv$.

It will occasionally be helpful to illustrate inequations by diagrams similar to those given in the preceding chapter. The fact that a diagram illustrates an inequation rather than an equation is signalled by inserting an inclusion sign at an appropriate point. For instance, the diagram

$$\begin{tikzcd} A \arrow[d, "S_1"'] \arrow[dr, phantom, "\subseteq"] & B \arrow[l, "T_1"'] \arrow[d, "T_2"] \\ C & D \arrow[l, "S_2"] \end{tikzcd}$$

depicts the inequation $S_1 \cdot T_1 \subseteq S_2 \cdot T_2$. In such cases, one says that a diagram *semi-commutes*.

Meet

The second assumption is that for all arrows $R, S : A \leftarrow B$ there is an arrow $R \cap S : A \leftarrow B$, called the *meet* of R and S, and characterised by the universal property

$$X \subseteq (R \cap S) \quad \equiv \quad (X \subseteq R) \text{ and } (X \subseteq S), \tag{4.1}$$

for all $X : A \leftarrow B$. In words, $R \cap S$ is the greatest lower bound of R and S. Using this universal property of meet it can easily be established that meet is commutative, associative and idempotent. In symbols:

$$\begin{aligned} R \cap S &= S \cap R \\ R \cap (S \cap T) &= (R \cap S) \cap T \\ R \cap R &= R. \end{aligned}$$

Using meet, we can restate the axiom of monotonicity as two inclusions:

$$\begin{aligned} R \cdot (S \cap T) &\subseteq (R \cdot S) \cap (R \cdot T) \\ (R \cap S) \cdot T &\subseteq (R \cdot T) \cap (S \cdot T). \end{aligned}$$

Given $\cap$ as an associative, commutative, and idempotent operation, we need not postulate inclusion of arrows as a primitive concept, for $R \subseteq S$ can be defined as an abbreviation for $R \cap S = R$.

Converse

Finally, for each arrow $R : A \leftarrow B$ there is an arrow $R^\circ : B \leftarrow A$ called the *converse* of R (and also known as the *reverse* or *reciprocal* of R). The converse operator has three properties. First, it is an involution:

$$(R^\circ)^\circ \quad = \quad R. \tag{4.2}$$

Second, it is order-preserving:

$$R \subseteq S \quad \equiv \quad R^\circ \subseteq S^\circ. \tag{4.3}$$

Third, it is contravariant:

$$(R \cdot S)^\circ \quad = \quad S^\circ \cdot R^\circ. \tag{4.4}$$

Using (4.2) and (4.3), together with the universal property (4.1), we obtain that converse distributes over meet:

$$(R \cap S)^\circ \quad = \quad R^\circ \cap S^\circ. \tag{4.5}$$

Use of these four properties in calculations will usually be signalled just by the hint *converse*.

The modular law

There is one more axiom that connects all three operators in an allegory. The axiom is called the *modular law* and states that

$$(R \cdot S) \cap T \quad \subseteq \quad R \cdot (S \cap (R^\circ \cdot T)). \tag{4.6}$$

The modular law is also known as *Dedekind's* rule. The modular law holds in **Rel**, the proof being as follows:

$$\begin{array}{ll} & (\exists b : aRb \wedge bSc) \wedge aTc \\ \equiv & \quad \{\text{predicate calculus}\} \\ & (\exists b : aRb \wedge bSc \wedge aTc) \\ \Rightarrow & \quad \{\text{since } aRb \wedge aTc \Rightarrow b(R^\circ \cdot T)c\} \\ & (\exists b : aRb \wedge bSc \wedge b(R^\circ \cdot T)c) \\ \equiv & \quad \{\text{meet}\} \\ & (\exists b : aRb \wedge b(S \cap (R^\circ \cdot T))c). \end{array}$$

One can think of the modular law as a weak converse of the distributivity of composition over meet.

By applying converse to both sides of the modular law and renaming, we obtain the dual variant

$$(R \cdot S) \cap T \quad \subseteq \quad (R \cap (T \cdot S^\circ)) \cdot S. \tag{4.7}$$

In fact, the modular law can be stated symmetrically in R and S:

$$(R \cdot S) \cap T \quad \subseteq \quad (R \cap (T \cdot S^\circ)) \cdot (S \cap (R^\circ \cdot T)). \tag{4.8}$$

Let us prove that (4.8) is equivalent to the preceding two versions. First, monotonicity of composition gives at once that (4.8) implies both (4.6) and (4.7). For the other direction, we reason:

$$\begin{array}{ll} & (R \cdot S) \cap T \\ = & \quad \{\text{meet idempotent}\} \\ & (R \cdot S) \cap T \cap T \\ \subseteq & \quad \{\text{inequation (4.7), writing } U = R \cap (T \cdot S^\circ)\} \\ & (U \cdot S) \cap T \\ \subseteq & \quad \{\text{inequation (4.6)}\} \\ & U \cdot (S \cap (U^\circ \cdot T)) \\ \subseteq & \quad \{\text{since } U \subseteq R\text{; converse; monotonicity}\} \\ & U \cdot (S \cap (R^\circ \cdot T)). \end{array}$$

In particular, taking $T = id$ and replacing R by R° in (4.8), we obtain

$$(R^\circ \cdot S) \cap id \;\subseteq\; (R \cap S)^\circ \cdot (R \cap S). \tag{4.9}$$

This inclusion is useful when reasoning about the range operator, defined below.

A proof similar to the above one gives that

$$R \;\subseteq\; R \cdot R^\circ \cdot R. \tag{4.10}$$

This completes the formal definition of an allegory. Note that neither the join operation ($\cup$) nor the complementation operator ($\neg$) on arrows are part of the definition of an allegory, even though both are meaningful in **Rel**.

To explore the consequences of the axiomatisation we will need some additional concepts and notation, and we turn to these next.

Exercises

4.1 Using an indirect proof and the universal property of meet, prove that meet is associative: $R \cap (S \cap T) = (R \cap S) \cap T$.

4.2 Translate the following semi-commutative diagrams into inequations:

$$\begin{tikzcd}
A \arrow[dr, "S"'] & & B \arrow[ll, "R"'] \arrow[dl, "T"] \\
& C &
\arrow[from=1-2, to=2-2, phantom, "\subseteq"]
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
A \arrow[d, "S"'] & & B \arrow[ll, "R"'] \arrow[dll, "T"] \\
C & & D \arrow[ll, "U"] \arrow[u, "V"']
\arrow[from=1-1, to=2-2, phantom, "\subseteq"]
\arrow[from=1-2, to=2-3, phantom, "\supseteq"]
\end{tikzcd}$$

4.3 Find a counter-example for $(R \cdot S) \cap (R \cdot T) \subseteq R \cdot (S \cap T)$.

4.4 The term *universal property of meet* suggests that $R \cap S$ is the terminal object in a certain category. Is it?

4.5 Show that $R \cap (S \cdot T) = R \cap S \cdot ((S^\circ \cdot R) \cap T)$.

4.6 Prove that $R \subseteq R \cdot R^\circ \cdot R$.

4.7 Prove that if **A** and **B** are allegories, then so is **A** $\times$ **B**.

4.2 Special properties of arrows

Various properties of relations that relate to order are familiar from ordinary set theory, and can be stated very concisely in the language of relations and allegories.

An arrow $R : A \leftarrow A$ is said to be *reflexive* if $id_A \subseteq R$ and *transitive* if $R \cdot R \subseteq R$. An arrow that is both reflexive and transitive is called a *preorder*. The converse of a preorder is again a preorder, and monotonicity of composition gives us that the meet of two preorders is a preorder as well.

An arrow $R : A \leftarrow A$ is said to be *symmetric* if $R \subseteq R^\circ$. Because converse is a monotonic involution, this is the same as saying that $R = R^\circ$. Again it is easy to check that the meet of two symmetric arrows is symmetric.

An arrow $R : A \leftarrow A$ is said to be *anti-symmetric* if $R \cap R^\circ \subseteq id_A$. An anti-symmetric preorder is called a *partial order*. A symmetric preorder is called an *equivalence*. If R is a preorder, then $R \cap R^\circ$ is an equivalence.

A less familiar notion, but one which turns out to be extremely useful, is that of a *coreflexive* arrow. An arrow $C : A \leftarrow A$ is called coreflexive if $C \subseteq id_A$. One can think of C as a subset of A. Every coreflexive arrow is both transitive and symmetric. Here is a proof of symmetry:

$$\begin{array}{ll} & C \\ \subseteq & \quad \{\text{inequation (4.10)}\} \\ & C \cdot C^\circ \cdot C \\ \subseteq & \quad \{\text{since } C \text{ coreflexive}\} \\ & id \cdot C^\circ \cdot id \\ = & \quad \{\text{identity arrows}\} \\ & C^\circ \end{array}$$

The proof of transitivity is even easier and is left as an exercise.

Range and domain

Associated with every arrow $R : A \leftarrow B$ are two coreflexives $ran\, R : A \leftarrow A$ and $dom\, R : B \leftarrow B$, called the *range* and *domain* of R respectively. Below we shall only discuss range; the properties of domain follow by duality since, by definition,

$$dom\, R \quad = \quad ran\, R^\circ.$$

One way of defining $ran\, R : A \leftarrow A$ is by the universal property

$$ran\, R \subseteq X \quad \equiv \quad R \subseteq X \cdot R \qquad \text{for all } X \subseteq id_A. \tag{4.11}$$

The intended interpretation in **Rel** is that $a(\mathit{ran}\ R)a$ holds if there exists an element b such that aRb.

We can also define $\mathit{ran}\ R$ directly:

$$\mathit{ran}\ R \quad = \quad (R \cdot R^\circ) \cap id. \tag{4.12}$$

To prove that (4.12) implies (4.11), note that

$$R \quad = \quad R \cap R \quad \subseteq \quad ((R \cdot R^\circ) \cap id) \cdot R \quad = \quad \mathit{ran}\ R \cdot R$$

by the modular law and so, by monotonicity, we obtain

$$\mathit{ran}\ R \subseteq X \quad \Rightarrow \quad R \subseteq X \cdot R$$

for any X. Conversely,

$$\begin{array}{cl} & (R \cdot R^\circ) \cap id \\ \subseteq & \quad \{\text{assume } R \subseteq X \cdot R\} \\ & (X \cdot R \cdot R^\circ) \cap id \\ \subseteq & \quad \{\text{modular law}\} \\ & X \cdot ((R \cdot R^\circ) \cap X^\circ) \\ \subseteq & \quad \{\text{meet}\} \\ & X \cdot X^\circ \\ \subseteq & \quad \{\text{assuming } X \text{ is a coreflexive}\} \\ & X, \end{array}$$

completing the proof.

If X is coreflexive, then $X \cdot R \subseteq R$, and so $R \subseteq X \cdot R$ if and only if $R = X \cdot R$. In particular, taking $X = \mathit{ran}\ R$ in (4.11) we obtain

$$R \quad = \quad \mathit{ran}\ R \cdot R. \tag{4.13}$$

Taking $R = S \cdot T$ and $X = \mathit{ran}\ S$ in (4.11), and using (4.13), we obtain $\mathit{ran}\ (S \cdot T) \subseteq \mathit{ran}\ S$. In fact, this result can be sharpened: we have

$$\mathit{ran}\ (R \cdot S) \quad = \quad \mathit{ran}\ (R \cdot \mathit{ran}\ S). \tag{4.14}$$

In one direction the proof is

$$\mathit{ran}\ (R \cdot S) = \mathit{ran}\ (R \cdot \mathit{ran}\ S \cdot S) \subseteq \mathit{ran}\ (R \cdot \mathit{ran}\ S).$$

The other direction follows from (4.12).

Finally, let us consider briefly how the range operator interacts with meet. From the direct definition of range (4.12) and monotonicity, we have

$$ran\,(R \cap S) \quad \subseteq \quad id \cap (R \cdot S^\circ).$$

The converse inequation also holds, by (4.9), and therefore

$$ran\,(R \cap S) \quad = \quad id \cap (R \cdot S^\circ). \tag{4.15}$$

Simple and entire arrows

An allegory has three subcategories of special interest, the categories formed by taking just: (i) the simple arrows, also called partial functions; (ii) the entire arrows, also called total relations; and (iii) those arrows that are both simple and entire, that is, functions. We now examine each of these subcategories in some detail.

An arrow $S : A \leftarrow B$ is said to be *simple* if

$$S \cdot S^\circ \quad \subseteq \quad id_A.$$

Simple arrows are also known as *imps* (short for *implementations*) or *partial functions.* In set-theoretic terms, S is simple if for every b there exists at most one a such that aSb. Simple arrows satisfy various algebraic properties not enjoyed by arbitrary arrows. For example, the modular law can be strengthened to an identity:

$$(S \cdot R) \cap T \quad = \quad S \cdot (R \cap (S^\circ \cdot T)) \quad \textit{provided } S \textit{ is simple.} \tag{4.16}$$

The inclusion ($\supseteq$) is proved as follows:

$$S \cdot (R \cap (S^\circ \cdot T)) \subseteq (S \cdot R) \cap (S \cdot S^\circ \cdot T) \subseteq (S \cdot R) \cap T.$$

We also have that composition of simple arrows right-distributes through meets:

$$(R \cap T) \cdot S \quad = \quad (R \cdot S) \cap (T \cdot S) \quad \textit{provided } S \textit{ is simple.} \tag{4.17}$$

Again, the proof makes essential use of the modular law.

An arrow $R : A \leftarrow B$ is said to be *entire* if

$$id_B \quad \subseteq \quad R^\circ \cdot R.$$

Equivalently, R is entire when $dom\,R = id_B$. In set-theoretic terms R is entire if for every b there exists at least one a such that aRb. Since $dom\,(R \cdot S) \subseteq dom\,S$ we have that S is entire whenever $R \cdot S$ is for any R. Also clear is the fact that if R is entire and $R \subseteq S$, then S is entire. Finally, using (4.15) it is easy to show that

$$R \cap S \;\; \text{entire} \quad \equiv \quad id \subseteq R^\circ \cdot S. \tag{4.18}$$

This condition will be useful below.

An arrow that is both simple and entire is said to be a *function*. Single lower-case letters will always denote functions, even if we do not say so explicitly. For any allegory **A**, its subcategory of functions will be denoted by $\mathit{Fun}(\mathbf{A})$. In particular, $\mathit{Fun}(\mathbf{Rel}) = \mathbf{Fun}$.

The following two *shunting rules* for functions are very useful:

$$f \cdot R \subseteq S \quad \equiv \quad R \subseteq f^\circ \cdot S \tag{4.19}$$

$$R \cdot f^\circ \subseteq S \quad \equiv \quad R \subseteq S \cdot f. \tag{4.20}$$

To prove (4.19) we reason:

$$\begin{array}{ll} & f \cdot R \subseteq S \\ \Rightarrow & \quad \{\text{monotonicity}\} \\ & f^\circ \cdot f \cdot R \subseteq f^\circ \cdot S \\ \Rightarrow & \quad \{\text{since } f \text{ is entire}\} \\ & R \subseteq f^\circ \cdot S \\ \Rightarrow & \quad \{\text{monotonicity}\} \\ & f \cdot R \subseteq f \cdot f^\circ \cdot S \\ \Rightarrow & \quad \{\text{since } f \text{ is simple}\} \\ & f \cdot R \subseteq S. \end{array}$$

The dual form (4.20) is obtained by taking converses. Any arrow f satisfying either (4.19) or (4.20) for all R and S is necessarily a function; the proof is left to the reader.

An easy consequence of the shunting rules is the fact that inclusion of functions reduces to equality:

$$(f \subseteq g) \quad \equiv \quad (f = g) \quad \equiv \quad (f \supseteq g).$$

We reason:

$$\begin{array}{ll} & f \subseteq g \\ \equiv & \quad \{\text{shunting}\} \\ & id \subseteq f^\circ \cdot g \\ \equiv & \quad \{\text{shunting}\} \\ & g^\circ \subseteq f^\circ \\ \equiv & \quad \{\text{converse is a monotonic involution}\} \\ & g \subseteq f. \end{array}$$

This fact is used frequently.

Functions can also be characterised without making explicit use of the converse operator. This result will be of fundamental importance in the following chapter, so we record it as a proposition.

Proposition 4.1 Suppose that $R : A \leftarrow B$ and $S : B \leftarrow A$ satisfy $R \cdot S \subseteq id$ and $id \subseteq S \cdot R$. Then $S = R^\circ$, and so R is a function.

Proof. First observe that $id \subseteq S \cdot R$ implies that $S \cdot R$ is entire. Hence R is entire as well.

We now reason:

$$
\begin{array}{cl}
 & S \\
\subseteq & \quad \{\text{since } R \text{ is entire}\} \\
 & R^\circ \cdot R \cdot S \\
\subseteq & \quad \{\text{since } R \cdot S \subseteq id\} \\
 & R^\circ .
\end{array}
$$

By taking $R = S^\circ$ and $S = R^\circ$ in the above argument, we also have $R^\circ \subseteq S$, and so $S = R^\circ$.

□

Exercises

4.8 Prove that coreflexives are transitive.

4.9 Let A and B be coreflexive arrows. Prove that $A \cdot B = A \cap B$.

4.10 Let C be coreflexive. Prove that $(C \cdot R) \cap S = C \cdot (R \cap S)$.

4.11 Let C be coreflexive. Prove that

$$
\begin{aligned}
(C \cdot X) \cap id &= (X \cdot C) \cap id \\
&= (C \cdot X \cdot C) \cap id \\
&= C \cdot (X \cap id) \\
&= (X \cap id) \cdot C.
\end{aligned}
$$

4.12 Show that, when C is coreflexive, $ran(C \cdot R) = C \cdot ranR$.

4.13 An arrow is said to be idempotent if $R \cdot R = R$. Prove that an arrow which is both symmetric and transitive is idempotent.

4.14 Prove that R is symmetric and transitive if and only if $R = R \cdot R^\circ$.

4.15 Prove that if S is simple, $S = S \cdot S^\circ \cdot S$. Does this equation imply simplicity?

4.16 Prove that $ran\,(R \cap (S \cdot T)) = ran\,((R \cdot T^\circ) \cap S)$.

4.17 Prove that $dom\,R \cdot f = f \cdot dom\,(R \cdot f)$.

4.18 A *locale* is a partial order $(\sqsubseteq, V)$ in which every subset $X \subseteq V$ has a least upper bound $\bigsqcup X$, and any two elements a, b have a greatest lower bound $a \sqcap b$. Furthermore, it is required that

$$(\bigsqcup X) \sqcap b \quad = \quad \bigsqcup\{\, a \sqcap b \mid a \in X\,\}.$$

A *V-valued relation* of type $A \leftarrow B$ is a function $V \leftarrow (A \times B)$. Show that V-valued relations form an allegory.

4.3 Tabular allegories

The definition of an allegory is very general and admits models that are quite different from set-theoretic relations. Surprisingly, however, one only needs to make two additional assumptions, the existence of *tabulations* and a *unit*, to get very close to set-theoretic relations, at least in terms of proofs. The existence of tabulations makes it possible to mimic pointwise proofs in a categorical setting. In a pointwise proof we reason about relations as binary predicates, manipulating aRb instead of R itself. In some cases pointwise proofs are more effective than point-free proofs; indeed it may even happen that no point-free proof is available. Tabulations give us a means of overcoming this phenomenon and thus the best of both worlds.

Tabulations

Let $R : A \leftarrow B$. A pair of functions $f : A \leftarrow C$ and $g : B \leftarrow C$ is said to be a *tabulation* of R if

$$R = f \cdot g^\circ \qquad \text{and} \qquad (f^\circ \cdot f) \cap (g^\circ \cdot g) = id.$$

An allegory is said to be *tabular* if every arrow has a tabulation.

In particular, the allegory **Rel** is tabular. In **Rel** a relation $R : A \leftarrow B$ can be identified with a subset C of $A \times B$. Taking f and g to be the projection functions $outl : A \leftarrow C$ and $outr : B \leftarrow C$, we obtain $R = outl \cdot outr^\circ$. Moreover, in **Rel** the projection functions satisfy

$$(outl^\circ \cdot outl) \cap (outr^\circ \cdot outr) \quad = \quad id,$$

as one can easily check, so the second condition is satisfied as well.

In any tabular allegory, the condition $(f^\circ \cdot f) \cap (g^\circ \cdot g) = id$ is equivalent to saying that the pair of functions (f, g) is *jointly monic*, that is, if for all functions h and k we have

$$h = k \quad \equiv \quad (f \cdot h = f \cdot k \text{ and } g \cdot h = g \cdot k).$$

In one direction we reason:

$$\begin{array}{ll}
& f \cdot h = f \cdot k \text{ and } g \cdot h = g \cdot k \\
\equiv & \quad \{\text{shunting of functions}\} \\
& h \cdot k^\circ \subseteq f^\circ \cdot f \text{ and } h \cdot k^\circ \subseteq g^\circ \cdot g \\
\equiv & \quad \{\text{meet}\} \\
& h \cdot k^\circ \subseteq (f^\circ \cdot f) \cap (g^\circ \cdot g) \\
\equiv & \quad \{\text{assumption}\} \\
& h \cdot k^\circ \subseteq id \\
\equiv & \quad \{\text{shunting of functions}\} \\
& h \subseteq k \\
\equiv & \quad \{\text{inclusion of functions is equality}\} \\
& h = k.
\end{array}$$

For the other direction, assume that (h, k) is a tabulation of $(f^\circ \cdot f) \cap (g^\circ \cdot g)$:

$$\begin{array}{ll}
& h \cdot k^\circ \subseteq (f^\circ \cdot f) \cap (g^\circ \cdot g) \\
\equiv & \quad \{\text{as before}\} \\
& f \cdot h = f \cdot k \text{ and } g \cdot h = g \cdot k \\
\equiv & \quad \{\text{assuming } (f, g) \text{ is jointly monic}\} \\
& h = k,
\end{array}$$

and so $(f^\circ \cdot f) \cap (g^\circ \cdot g) = h \cdot h^\circ \subseteq id$. But since f and g are entire, this inclusion can be strengthened to an equality.

The kind of reasoning seen in the last part of this proof is typical: we are essentially doing pointwise proofs in a relational framework. Similar reasoning arises in the proof of the next result, which gives a characterisation of inclusion in terms of functions.

Proposition 4.2 Let (f, g) be a tabulation of R. Then $h \cdot k^\circ \subseteq R$ if and only if there exists a (necessarily unique) function m such that $h = f \cdot m$ and $k = g \cdot m$.

Proof. Given the existence of function m such that

$$\begin{tikzcd} & A \arrow[dl, "h"'] \arrow[dr, "k"] \arrow[dd, "m"] & \\ B & & C \\ & D \arrow[ul, "f"] \arrow[ur, "g"'] & \end{tikzcd}$$

commutes, we have

$$\begin{array}{cl} & h \cdot k^\circ \\ = & \quad \{\text{assumption, converse}\} \\ & f \cdot m \cdot m^\circ \cdot g^\circ \\ \subseteq & \quad \{m \text{ simple}\} \\ & f \cdot g^\circ \\ = & \quad \{(f, g) \text{ tabulates } R\} \\ & R. \end{array}$$

In the other direction, define $m = (f^\circ \cdot h) \cap (g^\circ \cdot k)$. We first show that m is simple:

$$\begin{array}{cl} & m \cdot m^\circ \\ = & \quad \{\text{definition and converse}\} \\ & ((f^\circ \cdot h) \cap (g^\circ \cdot k)) \cdot ((h^\circ \cdot f) \cap (k^\circ \cdot g)) \\ \subseteq & \quad \{\text{monotonicity}\} \\ & (f^\circ \cdot h \cdot h^\circ \cdot f) \cap (g^\circ \cdot k \cdot k^\circ \cdot g) \\ \subseteq & \quad \{h \text{ and } k \text{ are simple}\} \\ & (f^\circ \cdot f) \cap (g^\circ \cdot g) \\ \subseteq & \quad \{(f, g) \text{ tabulates } R\} \\ & id. \end{array}$$

To show that m is entire, we can appeal to (4.18) and prove $id \subseteq h^\circ \cdot f \cdot g^\circ \cdot k$. The argument is

$$\begin{array}{cl} & id \\ \subseteq & \quad \{h \text{ and } k \text{ are entire}\} \\ & h^\circ \cdot h \cdot k^\circ \cdot k \\ \subseteq & \quad \{\text{assumption and } (f, g) \text{ tabulates } R\} \\ & h^\circ \cdot f \cdot g^\circ \cdot k. \end{array}$$

Since we now know m is a function and

$$f \cdot m \subseteq f \cdot f^{\circ} \cdot h \subseteq h,$$

we obtain that $f \cdot m = h$ because inclusion of functions is equality. By symmetry, it is also the case that $g \cdot m = k$. Finally, the fact that m is uniquely defined by $f \cdot m = h$ and $g \cdot m = k$ follows at once from the fact that (f, g) is jointly monic.

□

One import of this result is that tabulations are unique up to unique isomorphism, that is, if both (f, g) and (h, k) tabulate R, then there is a unique isomorphism m such that $h = f \cdot m$ and $k = g \cdot m$.

Unit

A *unit* in an allegory is an object U with two properties. First, id_U is the largest arrow of type $U \leftarrow U$, that is,

$$R \subseteq id_U \quad \Leftarrow \quad R : U \leftarrow U.$$

In other words, every arrow $U \leftarrow U$ is a coreflexive. The second property is that for every object A there exists an entire arrow $p_A : U \leftarrow A$. This entire arrow is necessarily a function because $p_A \cdot p_A{}^{\circ} \subseteq id_U$ by the first condition, so p_A is simple as well. An allegory possessing a unit is called *unitary*.

The allegory **Rel** is unitary: a unit is a singleton set and p_A is the unique function mapping every element of A to the sole inhabitant of the singleton. Recall that a singleton set in **Rel** is a terminal object in **Fun**, its subcategory of functions.

In a unitary allegory we have, for any relation $R : A \leftarrow B$, that

$$\begin{array}{ll} & R \subseteq p_A{}^{\circ} \cdot p_B \\ \equiv & \quad \{\text{shunting functions}\} \\ & p_A \cdot R \cdot p_B{}^{\circ} \subseteq id_U \\ \equiv & \quad \{\text{definition unit}\} \\ & \mathit{true}. \end{array}$$

In other words, $p_A{}^{\circ} \cdot p_B$ is the largest arrow of type $A \leftarrow B$. From now on, we shall denote this arrow by $\Pi : A \leftarrow B$. In **Rel**, the arrow Π is just the product $A \times B$.

As a special case of the above result we have that $\Pi : U \leftarrow A$ is the arrow p_A, and since inclusion of functions is equality, it follows that p_A is the only function with this type. Thus a unit in an allegory is always a terminal object in its subcategory of functions, a point we illustrated above in the case of **Rel**.

Restricted to tabular allegories, the converse is also true: in a tabular allegory, a terminal object in the subcategory of functions is a unit of the allegory. By the definition of a unit, it suffices to show that id_1 is the largest arrow of type $1 \leftarrow 1$. To this end, let $R : 1 \leftarrow 1$, and let (f, g) be a tabulation of R. Because 1 is terminal and $f, g : 1 \leftarrow A$ we have $f = g = !$. Hence,

$$R = f \cdot g^\circ = ! \cdot !^\circ \subseteq id,$$

and the claim is proved. Based on this equivalence of terminal objects and units, we will write $!_A$ in preference to p_A, and 1 instead of U when discussing units.

Finally, let us consider the tabulation (f, g) of $\Pi : A \leftarrow B$, where $f : A \leftarrow C$ and $g : B \leftarrow C$ for some C. Since $h \cdot k^\circ \subseteq \Pi$ for any two functions $h : A \leftarrow D$ and $k : B \leftarrow D$, we obtain from Proposition 4.2 that $m = (f^\circ \cdot h) \cap (g^\circ \cdot k)$ is the unique arrow such that $h = f \cdot m$ and $k = g \cdot m$. But this just says that C, together with f and g, is a product of A and B in the subcategory of functions. So without loss of generality, we may assume that $C = A \times B$, and $(f, g) = (outl, outr)$. Also, $m = \langle h, k \rangle$ and so

$$\langle h, k \rangle \quad = \quad (outl^\circ \cdot h) \cap (outr^\circ \cdot k).$$

We will use this fact in the following chapter, when we discuss how to obtain a useful notion of products in a relational setting.

Set-theoretic relations

Finally, let us briefly return to the question of pointless and pointwise proofs. There is a meta-theorem about unitary tabular allegories which makes precise our intuition that they are very much like relations in set theory.

A *Horn-sentence* is a predicate of the form

$$E_1 = D_1 \wedge E_2 = D_2 \wedge \ldots \wedge E_n = D_n \quad \Rightarrow \quad E_{n+1} = D_{n+1},$$

where E_i and D_i are expressions that refer only to the operators of an allegory, as well as tabulations and units. The meta-theorem is that a Horn-sentence holds for every unitary tabular allegory if and only if it is true for the specific category **Rel** of sets and relations. In other words, everything we have said so far could have been proved by checking it in set theory. A proof of this remarkable meta-theorem is outside the scope of this book and the interested reader is referred to (Freyd and Ščedrov 1990) for details.

Although set-theoretic proofs are valid for any unitary tabular allegory, direct proofs from the axioms are usually simpler, possess more structure, and are more revealing. Accordingly, we resort to proof by tabulation only when other methods fail.

Exercises

4.19 Prove that a function m is monic if and only if $m^\circ \cdot m = id$.

4.20 Prove that for every function f there exist functions c and m such that

$$f = m \cdot c \text{ and } c \cdot c^\circ = id \text{ and } m^\circ \cdot m = id.$$

Is this factorisation in some sense unique? (In text books on category theory, this factorisation is introduced under the heading 'epi-monic factorisation'.)

4.21 Show that if (f, g) tabulates R and R is simple, then g is monic.

4.22 Show that if $R = f \cdot g^\circ$ and R is entire, $g \cdot g^\circ = id$.

4.23 Using the above two exercises, show that if (f, g) tabulates R and R is a function, then g is an isomorphism.

4.24 Show that $h \cdot k^\circ \subseteq R \cdot S$ iff there exists an entire relation Q such that

$$h \cdot Q^\circ \subseteq R \quad \text{and} \quad Q \cdot k^\circ \subseteq S.$$

4.25 Is the allegory of V-valued relations in Exercise 4.18 tabular?

4.26 Prove that $(X \subseteq Y) \equiv (ran\, X \subseteq ran\, Y)$ for all $X, Y : A \leftarrow 1$.

4.27 Prove that $dom\, S = id \cap \Pi \cdot S$.

4.4 Locally complete allegories

It is now time to study the operator that seems, somewhat mysteriously, to have been left out of the discussion so far: the operator $\cup$ that returns the union of two relations. In fact, we will consider the more general operator $\bigcup$ that returns the union of a set of relations. In particular, we shall see how its distributivity properties give rise to two other operations, implication ($\Rightarrow$) and division ($\backslash$).

Join and zero

An allegory is said to be *locally complete* if for any set $\mathcal{H}$ of arrows $A \leftarrow B$ there is an arrow $\bigcup \mathcal{H} : A \leftarrow B$, called the *join* of $\mathcal{H}$, characterised by the universal property

$$\bigcup \mathcal{H} \subseteq X \quad \equiv \quad (\forall R \in \mathcal{H} : R \subseteq X)$$

for all $X : A \leftarrow B$.

It is assumed that meet and composition distribute over join:

$$
\begin{array}{rcl}
(\bigcup\mathcal{H}) \cap S & = & \bigcup\{R \cap S \mid R \in \mathcal{H}\} \\
(\bigcup\mathcal{H}) \cdot S & = & \bigcup\{R \cdot S \mid R \in \mathcal{H}\}.
\end{array}
$$

Neither of these equations follows from the universal property of join. On the other hand, the universal property does give us that converse distributes over join:

$$(\bigcup\mathcal{H})^\circ \quad = \quad \bigcup\{R^\circ \mid R \in \mathcal{H}\}.$$

This is because converse is a monotonic involution. In the special case where $\mathcal{H}$ is the empty set we write $\emptyset$ for $\bigcup\mathcal{H}$; when $\mathcal{H} = \{R, S\}$ we write $R \cup S$. By taking $\mathcal{H}$ to be the empty set we obtain that $\emptyset$ is a zero both of meet and composition. In **Rel**, the arrow $\emptyset$ is the empty relation.

Like meet, the binary join $\cup$ is associative, commutative and idempotent. It is important to bear in mind, however, that in a locally complete allegory there does not exist the symmetry between meet and join found in the predicate calculus: for meet one only has the modular law, while composition properly distributes over join.

Implication

Given two arrows $R, S : A \leftarrow B$, the *implication* $(R \Rightarrow S) : A \leftarrow B$ can be defined by the universal property

$$X \subseteq (R \Rightarrow S) \quad \equiv \quad (X \cap R) \subseteq S \quad \text{for all } X : A \leftarrow B.$$

The intended interpretation in set theory is that $a(R \Rightarrow S)b \equiv (aRb \Rightarrow aSb)$.

Implication can also be defined directly as a join:

$$R \Rightarrow S \quad = \quad \bigcup\{X \mid (X \cap R) \subseteq S\}.$$

To prove that this definition satisfies the universal property, assume first that $(X \cap R) \subseteq S$. Then we obtain $X \in \{X \mid (X \cap R) \subseteq S\}$ and so $X \subseteq (R \Rightarrow S)$ by the universal property of join. For the other direction we argue:

$$
\begin{array}{ll}
 & X \subseteq (R \Rightarrow S) \\
\equiv & \quad \{\text{definition of } R \Rightarrow S\} \\
 & X \subseteq \bigcup\{Y \mid (Y \cap R) \subseteq S\} \\
\Rightarrow & \quad \{\text{meet distributes over join}\} \\
 & X \cap R \subseteq \bigcup\{Y \cap R \mid (Y \cap R) \subseteq S\} \\
\Rightarrow & \quad \{\text{universal property of join}\} \\
 & X \cap R \subseteq S.
\end{array}
$$

Composing preorders

One use of implication is in composing preorders. Consider, for instance, the lexical ordering on pairs of numbers:

$$(a, b) \leq (c, d) \quad \equiv \quad a < c \ \vee \ (a = c \ \wedge \ b \leq d).$$

We can write this in an alternative way:

$$(a, b) \leq (c, d) \quad \equiv \quad a \leq c \ \wedge \ (c \leq a \Rightarrow b \leq d).$$

This suggests defining $R; S$ (pronounced 'R then S') by

$$R\,;S \quad = \quad R \cap (R^\circ \Rightarrow S).$$

The relation $R\,;S$ first compares two elements by R, and if the two elements are equivalent in R, it then compares them by S. In particular, the lexical ordering on pairs of numbers is rendered as

$$(\mathit{outl}^\circ \cdot \mathit{leq} \cdot \mathit{outl})\,;(\mathit{outr}^\circ \cdot \mathit{leq} \cdot \mathit{outr}),$$

where leq is the prefix name for $\leq$.

If R and S are preorders, then $R\,;S$ is a preorder. The proof makes use of the symmetric modular law (4.8) and is left as an exercise. One can also show that $(;)$ is associative with unit Π. This, too, is left as an exercise.

Division

Given two arrows R and S with a common target, the left-division operation $R\backslash S$ (pronounced 'R under S') is defined by the universal property

$$X \subseteq R\backslash S \quad \equiv \quad R \cdot X \subseteq S.$$

In a diagram, $X = R\backslash S$ is the largest arrow that makes the triangle

$$\begin{tikzcd}
A \arrow[dr, "R"'] & & C \arrow[ll, "X"'] \arrow[dl, "S"] \\
& B \arrow[u, phantom, "\subseteq"] &
\end{tikzcd}$$

semi-commute. The interpretation of $R\backslash S$ as a predicate is

$$a(R\backslash S)c \quad \equiv \quad (\forall b : bRa \Rightarrow bSc),$$

so the operation $(\backslash)$ gives us a way of expressing specifications that involve universal quantification.

Left-division can also be defined explicitly as a join:

$$R\backslash S \quad = \quad \bigcup\{X \mid R \cdot X \subseteq S\}.$$

The proof that this works hinges on the fact that composition distributes over join, and is analogous to the argument for implication spelled out above. Note that $R\backslash S$ is monotonic in S, but anti-monotonic in R. In fact, we have

$$(R \cup S)\backslash T = (R\backslash T) \cap (S\backslash T) \quad \text{and} \quad R\backslash(S \cap T) = (R\backslash S) \cap (R\backslash T).$$

The universal property of left-division also gives the identity

$$(R \cdot S)\backslash T \quad = \quad S\backslash(R\backslash T),$$

but nothing interesting can be said about $R\backslash(S \cdot T)$.

The cancellation law of division is

$$R \cdot (R\backslash S) \quad \subseteq \quad S$$

and its proof is an immediate consequence of the universal property.

Right-division

Since composition is symmetric in both arguments we can define the dual operation of right-division S/R (pronounced 'S over R') for any relations S and R with a common source:

$$X \subseteq S/R \quad \equiv \quad X \cdot R \subseteq S.$$

At the point level we have

$$a(S/R)b \quad = \quad (\forall c : aSc \Leftarrow bRc).$$

Since converse is a monotonic involution the two division operators can be defined in terms of each other:

$$R\backslash S = (S^\circ/R^\circ)^\circ \quad \text{and} \quad S/R = (R^\circ\backslash S^\circ)^\circ.$$

Sometimes it's better to use one version of division rather than the other; the choice is usually dictated by the desire to reduce the number of converses in an expression.

Exercises

4.28 Prove that the meet of a collection $\mathcal{H}$ of arrows can be constructed as a join:

$$\bigcap\mathcal{H} \quad = \quad \bigcup\{S \mid (\forall R \in \mathcal{H} : S \subseteq R)\}.$$

4.29 Prove that $ran\,(\bigcup \mathcal{H}) = \bigcup\{ran\, X \mid X \in \mathcal{H}\}$.

4.30 Show that there exists an operator $(-)$ such that

$$R - S \subseteq X \quad \equiv \quad R \subseteq S \cup X$$

for all X. Using this universal property, show that

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

4.31 Prove that $R = \emptyset$ if and only if $ran\, R = \emptyset$. Show how this may be used to prove that

$$((R \cdot S) \cap T = \emptyset) \equiv (R \cap (T \cdot S^\circ) = \emptyset).$$

4.32 Prove the following properties of implication using its universal property:

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

4.33 Prove the following property of implication

$$f^\circ \cdot (R \Rightarrow S) \cdot g \quad = \quad (f^\circ \cdot R \cdot g) \Rightarrow (f^\circ \cdot S \cdot g).$$

4.34 Prove that $R;S$ is a preorder if R and S are. Also prove that $(;)$ is associative with unit Π.

4.35 Prove the laws

$$\begin{array}{rcl} (R \backslash S) \cdot f & = & R \backslash (S \cdot f) \\ f^\circ \cdot (R \backslash S) & = & (R \cdot f) \backslash S. \end{array}$$

4.36 Let $(\leq, A)$ and $(\sqsubseteq, B)$ be preorders (in the ordinary set-theoretic sense, not as arrows in an allegory). A *Galois connection* is a pair (f, g) of monotonic mappings $f : A \leftarrow B$ and $g : B \leftarrow A$ such that

$$x \leq g\, y \quad \equiv \quad f\, x \sqsubseteq y.$$

The function f is called the *lower adjoint*, and g the *upper adjoint*. For example, defining

$$f\,X = X \cap R \quad \text{and} \quad g\,Y = R \Rightarrow Y,$$

we see that the universal property of $\Rightarrow$ in fact asserts the existence of a Galois connection. Spot the Galois connection in the universal properties of division and subtraction (see Exercise 4.30).

The following few exercises continue the theme of Galois connections.

4.37 What does it mean to say that the mapping $X \mapsto X \cdot R$ is lower adjoint to $Y \mapsto S \cdot Y$?

4.38 In Exercise 3.19, we defined the floor of a rational number using a universal property. This property can be phrased as a Galois connection; identify the relevant preorders and the adjoints.

4.39 Show how the universal property of binary meet can be viewed as a Galois connection.

4.40 Now consider a Galois connection between complete lattices (partially ordered sets where every subset has both a least upper bound (lub) and a greatest lower bound (glb)). Prove that the following two statements are equivalent:

- (f, g) is a Galois connection.
- f preserves least upper bounds and for all x,

$$g\,y \quad = \quad lub\{x \mid f\,x \leq y\}.$$

4.5 Boolean allegories

One operator is still missing, namely the operator $\neg$ of negation. In a locally complete allegory, one can define negation by

$$\neg R \quad = \quad (R \Rightarrow \emptyset).$$

This notion of negation satisfies a number of the properties one would expect. First of all, negation is order-reversing. Furthermore, we have *De Morgan's* law

$$\neg(R \cup S) \quad = \quad (\neg R) \cap (\neg S).$$

In general, however, it is not true that negation is an involution.

If the equation $\neg(\neg R) = R$ is satisfied, then the allegory is called *boolean*; in particular, **Rel** is boolean. Boolean allegories satisfy many properties that are not valid in other allegories. For instance, division can be defined in terms of negation:

$$X/Y \quad = \quad \neg(\neg X \cdot Y^\circ). \tag{4.21}$$

(From now on, we omit the brackets round $\neg X$, giving $\neg$ the highest priority in formulae.) This definition expresses the relationship between universal and existential quantification in classical logic. To prove (4.21), it suffices to show that

$$\neg R/Y \quad = \quad \neg(R \cdot Y^\circ), \tag{4.22}$$

because taking $R = \neg X$ and using $X = \neg(\neg X)$ gives the desired result. It turns out that equation (4.22) is valid in any locally complete allegory:

$$\begin{array}{ll}
 & X \subseteq \neg R/Y \\
\equiv & \quad \{\text{division}\} \\
 & X \cdot Y \subseteq \neg R \\
\equiv & \quad \{\text{negation}\} \\
 & X \cdot Y \subseteq (R \Rightarrow \emptyset) \\
\equiv & \quad \{\text{implication}\} \\
 & X \cdot Y \cap R \subseteq \emptyset \\
\equiv & \quad \{\text{Exercise (4.31)}\} \\
 & X \cap R \cdot Y^\circ \subseteq \emptyset \\
\equiv & \quad \{\text{implication; negation}\} \\
 & X \subseteq \neg(R \cdot Y^\circ).
\end{array}$$

Notice that the above proof uses indirect equational reasoning, proving that $R = S$ by showing that $X \subseteq R \equiv X \subseteq S$ for arbitrary X.

In our own experience, it is best to avoid proofs that involve negation as much as possible, since the number of rules in the relational calculus becomes quite unmanageable when boolean negation is considered.

Exercises

4.41 Without assuming the allegory is boolean, prove that:

$$\begin{array}{rcl}
\neg\Pi & = & 0 \\
\neg R & = & \neg\neg\neg R \\
\neg(R \cup S) & = & \neg R \cap \neg S \\
\neg\neg(R \cup \neg R) & = & \Pi
\end{array}$$

4.42 Prove that a locally complete allegory is boolean iff $R \cup \neg R = \Pi$ for all R.

4.43 In relational calculi that take negation as primitive, the use of division is often replaced by an appeal to *Schröder's rule*, which asserts that

$$(\neg T \cdot S^\circ \subseteq \neg R) \equiv (R \cdot S \subseteq T) \equiv (R^\circ \cdot \neg T \subseteq \neg S).$$

Prove that Schröder's rule is valid in any boolean allegory.

4.6 Power allegories

In set theory, relations are usually defined as subsets of a cartesian product, a fact we have used a number of times already. But it is important to observe that this is a more or less arbitrary decision, since relations could have been introduced as boolean-valued functions of two arguments, or as set-valued functions. In this section, we shall show how the notion of powersets may be defined in an allegory by exploiting the isomorphism between relations and set-valued functions.

Power transpose and epsilon

In order to model set-valued functions in an allegory $\mathbf{A}$ we need three things:

- for each object A in $\mathbf{A}$ an object $\mathsf{P}A$, called the *power-object* of A;
- a function Λ, called *power transpose*, that takes an arrow $R : A \leftarrow B$ and returns a function $\Lambda R : \mathsf{P}A \leftarrow B$;
- an arrow $\in : A \leftarrow \mathsf{P}A$, called the *membership* relation of P.

These three things are defined (up to unique isomorphism) by the following universal property. For all $R : A \leftarrow B$ and $f : \mathsf{P}A \leftarrow B$,

$$f = \Lambda R \quad \equiv \quad \in \cdot f = R.$$

The following diagram summarises the type information:

$$\begin{tikzcd} \mathsf{P}A \arrow[dr, "\in"'] & & B \arrow[ll, "\Lambda R"'] \arrow[dl, "R"] \\ & A & \end{tikzcd}$$

It is immediate from the universal property that $\Lambda R = \Lambda S$ implies $R = S$, so Λ is an isomorphism between relations and (set-valued) functions. In set theory, Λ is defined by the set comprehension

$$(\Lambda R)\, b \quad = \quad \{\, a \mid aRb \,\}.$$

Indeed, one might say that the definition of Λ is merely a restatement of the comprehension principle in axiomatic set theory.

Let us now see how the universal property of Λ can be used to prove some simple identities in set theory. First of all, by taking $f = \Lambda R$, we have the cancellation property

$$\in \cdot \Lambda R \quad = \quad R,$$

so the diagram above commutes.

As a consequence of Λ-cancellation we obtain the fusion law

$$\Lambda(R \cdot f) \quad = \quad \Lambda R \cdot f,$$

which is valid only if f is a function.

Finally, we have the reflection law $\Lambda\!\in\, = id$, which is proved by taking $f = id$ and $R = \,\in$ in the universal property.

For completeness we remark that the definition of $(\mathsf{P}A, \Lambda, \in)$ can also be phrased as asserting the existence of a terminal object in a certain category. Given an allegory $\mathbf{A}$ and an object A, consider the category $\mathbf{A}/A$ whose objects are all arrows R of $\mathbf{A}$ with target A, and whose arrows are those functions $f : R \leftarrow S$ for which the following diagram commutes:

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

Composition in $\mathbf{A}/A$ is the same as that in $\mathbf{A}$. Now, the terminal object of $\mathbf{A}/A$ is the relation $\in_A : A \leftarrow \mathsf{P}A$ because

$$f = \Lambda R \quad \Leftarrow \quad f : \in_A \leftarrow R$$

and so ΛR is just another notation for $!_R$.

Existential image

It is a general principle in category theory that any suitable operator on objects can be extended to a functor. Since we have just introduced the operator P on the objects of an allegory $\mathbf{A}$, it is natural to look for a corresponding functor. In the present case there are several possibilities, one of which is the *power* functor $\mathsf{P} : \mathbf{A} \leftarrow \mathbf{A}$, and another is the *existential image* functor $\mathsf{E} : \mathit{Fun}(\mathbf{A}) \leftarrow \mathbf{A}$. It is not immediately obvious what the action of the power functor should be on the arrows of an allegory, so we will postpone consideration of this possibility to the next chapter.

The existential image functor is defined by

$$\mathsf{E}R \quad = \quad \Lambda(R \cdot \in).$$

In set theory we have

$$(\mathsf{E}R)\, x \quad = \quad \{\, a \mid (\exists b : aRb \wedge b \in x)\,\}.$$

It is easy to see from the reflection law $\Lambda\!\in\; = id$ that E preserves identities. To show that E also preserves composition, it suffices to prove the *absorption* property

$$\mathsf{E}R \cdot \Lambda S \quad = \quad \Lambda(R \cdot S).$$

Taking $S = T \cdot \in$ gives us $\mathsf{E}R \cdot \mathsf{E}T = \mathsf{E}(R \cdot T)$, which is what is required.

Here is a proof of the absorption property:

$$\begin{array}{ll}
& \mathsf{E}R \cdot \Lambda S = \Lambda(R \cdot S) \\
\equiv & \quad \{\text{defining property of } \Lambda\} \\
& \in \cdot\, \mathsf{E}R \cdot \Lambda S = R \cdot S \\
\equiv & \quad \{\text{definition of } \mathsf{E}, \Lambda\text{-cancellation}\} \\
& R \cdot \in \cdot\, \Lambda S = R \cdot S \\
\equiv & \quad \{\Lambda\text{-cancellation}\} \\
& \mathit{true}
\end{array}$$

As an immediate consequence of Λ-cancellation, we obtain that $\in$ is a natural transformation $id \leftarrow \mathsf{JE}$, where $\mathsf{J} : \mathbf{A} \leftarrow \mathit{Fun}(\mathbf{A})$ is the inclusion functor.

The restriction of E to functions is called the *power* functor P; thus $\mathsf{P} = \mathsf{EJ}$. Note that $\mathsf{P} : \mathit{Fun}(\mathbf{A}) \leftarrow \mathit{Fun}(\mathbf{A})$, while $\mathsf{E} : \mathit{Fun}(\mathbf{A}) \leftarrow \mathbf{A}$. In set theory, P is the *map* operation that applies a function to all elements of a set:

$$\mathsf{P}f\, x \quad = \quad \{\, f\, a \mid a \in x\,\}.$$

In the following chapter we shall show how to extend P to a functor $\mathsf{P} : \mathbf{A} \leftarrow \mathbf{A}$.

From now on we will omit J in the composition JE, silently embedding $\mathit{Fun}(\mathbf{A})$ in $\mathbf{A}$; thus we assume that $\mathsf{E} : \mathbf{A} \leftarrow \mathbf{A}$.

Singleton and union

For any object A, the *singleton* function $\tau : \mathsf{P}A \leftarrow A$ is defined by $\tau = \Lambda id$. In set theory, τ takes an element and returns a singleton set. Using the fusion law $\Lambda(R \cdot f) = \Lambda R \cdot f$, we obtain

$$\Lambda f = \Lambda(id \cdot f) = \Lambda id \cdot f = \tau \cdot f$$

and so

$$\mathsf{P}f \cdot \tau = \mathsf{E}f \cdot \Lambda id = \Lambda(f \cdot id) = \Lambda f = \tau \cdot f.$$

Thus τ is a natural transformation $\mathsf{P} \leftarrow id$. These and similar facts illustrate the difference between P and E, for τ is *not* a natural transformation $\mathsf{E} \leftarrow id$.

For each A, the *union* function $\mu : \mathsf{P}A \leftarrow \mathsf{PP}A$ is given by $\mu = \mathsf{E}\!\in$. In words, μ returns the union of a collection of sets. Since $\in\, : id \leftarrow \mathsf{E}$, we have $\mu : \mathsf{E} \leftarrow \mathsf{EE}$. Union satisfies the *one-point* properties

$$\mu \cdot \mathsf{P}\tau \quad = \quad id = \mu \cdot \tau$$

as well as the *distribution* property

$$\mu \cdot \mu \quad = \quad \mu \cdot \mathsf{P}\mu.$$

This last result follows from the definition of μ plus naturality:

$$\mu \cdot \mathsf{P}\mu = \mu \cdot \mathsf{PE}\!\in\, = \mu \cdot \mathsf{EE}\!\in\, = \mathsf{E}\!\in \cdot\, \mu = \mu \cdot \mu.$$

In later chapters we will use *union* as a synonym for μ.

The subset relation

For any A, the *subset* relation $subset : \mathsf{P}A \leftarrow \mathsf{P}A$ is defined by $subset \;=\; \in\backslash\in$. Interpreted in set theory we have

$$x\; subset\; y \quad \equiv \quad (\forall a : a \in x \Rightarrow a \in y).$$

Note the distinction between *subset* and $\subseteq$: the former models the inclusion relation between sets, while the latter is the primitive operation that compares arrows in an allegory.

Based on its set-theoretic interpretation, we would expect that *subset* is a partial order. Reflexivity and transitivity are immediate from the properties of division, but the proof that *subset* is anti-symmetric, that is, $subset \cap subset^{\circ} = id$, requires a little more effort, and in fact holds only for a unitary tabular allegory. Given this assumption, we will prove a more general fact, namely, that

$$\Lambda R \quad = \quad (\in\backslash R) \cap (R\backslash\in)^{\circ}.$$

Anti-symmetry of *subset* then follows from the reflection law $\Lambda\in = id$.

To prove the above equation for Λ, we invoke the rule of indirect equality and establish that

$$X \subseteq \Lambda R \quad \equiv \quad X \subseteq (\in\backslash R) \cap (R\backslash\in)^{\circ}$$

for all X. Assume that (f, g) is a tabulation of X. We reason:

$$\begin{array}{ll}
 & f \cdot g^{\circ} \subseteq \Lambda R \\
\equiv & \quad \{\text{shunting of functions}\} \\
 & f = \Lambda R \cdot g \\
\equiv & \quad \{\text{fusion}\} \\
 & f = \Lambda(R \cdot g) \\
\equiv & \quad \{\text{universal property of } \Lambda\} \\
 & \in \cdot f = R \cdot g \\
\equiv & \quad \{\text{anti-symmetry of } \subseteq\} \\
 & (\in \cdot f \subseteq R \cdot g) \text{ and } (R \cdot g \subseteq \in \cdot f) \\
\equiv & \quad \{\text{shunting of functions}\} \\
 & (\in \cdot f \cdot g^{\circ} \subseteq R) \text{ and } (R \cdot g \cdot f^{\circ} \subseteq \in) \\
\equiv & \quad \{\text{division}\} \\
 & (f \cdot g^{\circ} \subseteq \in\backslash R) \text{ and } (g \cdot f^{\circ} \subseteq R\backslash\in) \\
\equiv & \quad \{\text{converse, meet}\} \\
 & f \cdot g^{\circ} \subseteq (\in\backslash R) \cap (R\backslash\in)^{\circ},
\end{array}$$

and we are done.

Exercises

4.44 Consider the equation $\Lambda(R \cdot f) = \Lambda R \cdot f$. Why is it not possible to replace the function f by an arbitrary arrow? Give a counter-example.

4.45 The notion of *non-empty* power objects (corresponding to non-empty subsets) can be defined by changing the defining property of power transpose slightly. What is the required change?

4.46 Show that τ is monic.

4.47 Prove that $\Lambda R = \mathsf{E}R \cdot \tau$, and $\mathsf{E}R = \mu \cdot \mathsf{P}(\Lambda R)$.

4.48 Prove that $(\Lambda R)^\circ \cdot \Lambda S = (R\backslash S) \cap (S\backslash R)^\circ$.

4.49 Prove that R is a preorder if and only if $R = R\backslash R$. Using this, show that R is a preorder if and only if there exists a partial order S and a function f such that $R = f^\circ \cdot S \cdot f$.

4.50 Prove that $\in/(R\backslash\in) = R$.

4.51 A *predicate transformer* is a function of type $\mathsf{P}A \leftarrow \mathsf{P}B$. One can define a partial order on predicate transformers by

$$f \leq g \quad \equiv \quad \in \cdot f \subseteq \in \cdot g.$$

A predicate transformer h is said to be *monotonic* if $h \cdot \mathit{subset} \subseteq \mathit{subset} \cdot h$. Prove that h is monotonic if and only if

$$f \leq g \quad \text{implies} \quad h \cdot f \leq h \cdot g$$

for all f and g.

4.52 For $R : B \leftarrow A$, consider the predicate transformer $\mathit{wlp}\, R : \mathsf{P}A \leftarrow \mathsf{P}B$ defined by $\mathit{wlp}\, R = \Lambda(R\backslash\in)$. Prove that $\mathit{wlp}\,(R \cdot S) = \mathit{wlp}\, S \cdot \mathit{wlp}\, R$, and

$$R \subseteq S \quad \equiv \quad \mathit{wlp}\, S \leq \mathit{wlp}\, R.$$

(Exercise 4.50 will come in handy here.) Finally, show how to associate with any predicate transformer $p : \mathsf{P}A \leftarrow \mathsf{P}B$ an arrow $S : B \leftarrow A$ so that

$$p \leq \mathit{wlp}\, R \quad \equiv \quad \mathit{wlp}\, S \leq \mathit{wlp}\, R$$

for any $R : B \leftarrow A$. (This Exercise is the topic of (Morgan 1993).)

Bibliographical remarks

The calculus of relations has a rich history, going back to (De Morgan 1860), (Peirce 1870) and (Schröder 1895). The subject as we know it today was mostly shaped by Tarski and his students in a series of articles, starting with (Tarski 1941). An overview of the origins of the relational calculus can be found in (Maddux 1991; Pratt 1992).

During the 1960s several authors started to explore relations in a categorical setting (Brinkmann 1969; Mac Lane 1961; Puppe 1962). This resulted in a consensus that *regular* categories are the appropriate setting for studying relations in general (Grillet 1970; Kawahara 1973b). In fact, a category is regular if and only if it is isomorphic to the subcategory of functions of a tabular unitary allegory. The study of categories of relations is still a buoyant subject, see for instance (Carboni, Kasangian, and Street 1984; Carboni and Street 1986; Carboni and Walters 1987). The definitive introduction to this area of category theory is the text book by Freyd and Scedrov (Freyd and Ščedrov 1990), on which our presentation is based.

Although we have omitted to elaborate on this, the axioms introduced here, namely those of a tabular allegory that has a unit and power objects, precisely constitute the definition of a *topos*. There are many books on topos theory (Barr and Wells 1985; Goldblatt 1986; Mac Lane and Moerdijk 1992; McLarty 1992), and several of the results quoted here are merely special cases of theorems that can be found in these books.

During the 1970s the calculus of relations was applied to various areas of computing science, see e.g. (De Bakker and De Roever 1973; De Roever 1972, 1976). This work culminated in (Sanderson 1980), where it was suggested that relational algebra could be a unifying instrument in theoretical computing. Our own understanding of the applications to program semantics has been much influenced by (Hoare and He 1986a, 1986b, 1987; Hoare, He, and Sanders 1987). It is probably fair to say that until recently, most of this work was based on Tarski's axiomatisation; the standard reference is (Schmidt and Ströhlein 1993). Related references are (Berghammer and Zierer 1986; Berghammer, Kempf, Schmidt, and Ströhlein 1991; Desharnais, Mili, and Mili 1993; Mili 1983; Mili, Desharnais, and Mili 1987). (Mili, Desharnais, and Mili 1994) is particularly similar to this book, in that it focuses on program derivation.

The categorical viewpoint of relations was first exploited in computing science by (Sheeran 1990), and later also by (Backhouse, De Bruin, Malcolm, Voermans, and Van der Woude 1991; Backhouse, De Bruin, Hoogendijk, Malcolm, Voermans, and Van der Woude 1992; Martin 1991; Kawahara 1990; Brown and Hutton 1994). The presentation in this chapter has greatly benefited from discussions with Roland Backhouse, Jaap van der Woude and their colleagues at Eindhoven University. The use of *Galois connections*, made explicit in the exercises, is mainly due to their influence. An especially interesting application of Galois connections in the relational calculus is presented by (Backhouse and Van der Woude 1993).

The calculus of binary relations is of course quite restrictive, and therefore a number of researchers have started to explore generalisations. In (Möller 1991, 1993; Möller and Russling 1994; Russling 1995) relations are taken to be sets of sequences rather than sets of strings: this is very convenient, for instance, when reasoning about graph algorithms. Another generalisation is to drop converse (Von Karger and Hoare 1995; Berghammer and Von Karger 1995): this leads to a calculus of processes. Another attempt at dealing with distributed algorithms in a relational setting is (Rietman 1995). Many of these developments are summarised in a forthcoming book on relational methods in computer science (Brink and Schmidt 1996).

A topic that we shall not address in this book is that of executing relational expressions. Clearly, it would be desirable to do so, but it is as yet unclear what the model of computation should be. One promising proposal, which involves rewriting using the axioms of an allegory, has been put forward by (Broome and Lipton 1994).

Datatypes in Allegories

The idea now is to replace categories by allegories as the mathematical basis of a useful calculus for deriving programs. However, there is a major stumbling block: categorical definitions of datatypes are not suitable when working in an allegory. Each allegory is identical to its opposite so dual categorical constructs coincide. In particular, products coincide with coproducts, which is not what one wants in a sensible theory of datatypes.

The solution proposed in this chapter is to define all relevant datatype constructions in *Fun*($\mathbf{A}$), and then extend them in some canonical way to $\mathbf{A}$. In fact, we show that the base functors of datatypes in *Fun*($\mathbf{A}$), the power functor, and type functors can all be extended to *monotonic* functors of $\mathbf{A}$. In particular this means that a monotonic extension of a categorical product exists in $\mathbf{A}$, and this extension – called *relational* product – can be used in place of a categorical product. Crucial to the success of the whole enterprise is the notion of tabulations introduced in the preceding chapter.

As a result, catamorphisms can be extended to include relational algebras. Relational catamorphisms are powerful tools for problem specification, and we go on to illustrate their use by showing how some standard combinatorial functions can be defined very succinctly using relations. This material is used heavily in later chapters on solving optimisation problems. The chapter ends with a discussion of how natural transformations can be generalised in a relational setting.

5.1 Relators

Let $\mathbf{A}$ and $\mathbf{B}$ be tabular allegories. By definition, a *relator* is a monotonic functor $\mathsf{F} : \mathbf{A} \leftarrow \mathbf{B}$, that is, a functor F satisfying

$$R \subseteq S \quad \Rightarrow \quad \mathsf{F}R \subseteq \mathsf{F}S$$

for all R and S.

As we shall see in Theorem 5.1 below, a relator can also be characterised as a functor on relations that preserves converse, that is,

$$(\mathsf{F}R)^\circ \quad = \quad \mathsf{F}(R^\circ).$$

As a first step towards proving this result, we prove the following lemma.

Lemma 5.1 Let F be a relator and f a function. Then $\mathsf{F}f$ is a function, and $(\mathsf{F}f)^\circ = \mathsf{F}(f^\circ)$.

Proof. Since functions are entire and simple, we have

$$\begin{array}{lllllll}
\mathsf{F}f \cdot \mathsf{F}(f^\circ) & = & \mathsf{F}(f \cdot f^\circ) & \subseteq & \mathsf{F}\,id & = & id \\
\mathsf{F}(f^\circ) \cdot \mathsf{F}f & = & \mathsf{F}(f^\circ \cdot f) & \supseteq & \mathsf{F}\,id & = & id.
\end{array}$$

Now recall Proposition 4.1 of the preceding chapter, which states that R is a function if and only if there exists an S such that $R \cdot S \subseteq id$ and $id \subseteq S \cdot R$. Furthermore, these two inequations imply that $S = R^\circ$. It follows that $\mathsf{F}f$ is a function with converse $\mathsf{F}(f^\circ)$.

□

Theorem 5.1 A functor is a relator if and only if it preserves converse.

Proof. First, assume F is a relator and let (f, g) be a tabulation of R. Using Lemma 5.1 we have:

$$\begin{array}{lllllllll}
\mathsf{F}(R^\circ) & = & \mathsf{F}((f \cdot g^\circ)^\circ) & = & \mathsf{F}(g \cdot f^\circ) & = & \mathsf{F}g \cdot \mathsf{F}(f^\circ) & & \\
(\mathsf{F}R)^\circ & = & (\mathsf{F}f \cdot \mathsf{F}(g^\circ))^\circ & = & (\mathsf{F}f \cdot (\mathsf{F}g)^\circ)^\circ & = & \mathsf{F}g \cdot (\mathsf{F}f)^\circ & = & \mathsf{F}g \cdot \mathsf{F}(f^\circ).
\end{array}$$

Thus $\mathsf{F}(R^\circ) = (\mathsf{F}R)^\circ$.

For the reverse direction we again use tabulations. Suppose $R = h \cdot k^\circ \subseteq f \cdot g^\circ = S$, with (f, g) jointly monic. By Proposition 4.2 of the preceding chapter there exists a function m such that $h = f \cdot m$ and $k = g \cdot m$. Hence, we can reason:

$$\begin{array}{ll}
& \mathsf{F}R \\
= & \quad \{\text{definition of } R \text{ and } \mathsf{F} \text{ a functor}\} \\
& \mathsf{F}h \cdot \mathsf{F}(k^\circ) \\
= & \quad \{\mathsf{F} \text{ preserves converse}\} \\
& \mathsf{F}h \cdot (\mathsf{F}k)^\circ
\end{array}$$

$$\begin{array}{cl} = & \{\text{definition of } m \text{ and } \mathsf{F} \text{ is a functor}\} \\ & \mathsf{F}f \cdot \mathsf{F}m \cdot (\mathsf{F}m)^\circ \cdot (\mathsf{F}g)^\circ \\ \subseteq & \{\text{since } \mathsf{F}m \text{ is a function and so is simple}\} \\ & \mathsf{F}f \cdot \mathsf{F}g^\circ \\ = & \{\mathsf{F} \text{ a functor and definition of } S\} \\ & \mathsf{F}S, \end{array}$$

and so F is monotonic.

□

Corollary 5.1 If two relators F and G agree on functions, that is, if $\mathsf{F}f = \mathsf{G}f$ for all f, then $\mathsf{F} = \mathsf{G}$.

Proof. Let R be an arbitrary relation, and (f, g) a tabulation of R. Then

$$\mathsf{F}R = \mathsf{F}(f \cdot g^\circ) = \mathsf{F}f \cdot (\mathsf{F}g)^\circ = \mathsf{G}f \cdot (\mathsf{G}g)^\circ = \mathsf{G}R.$$

□

One consequence of Theorem 5.1 is that, when F is a relator, it is safe to write $\mathsf{F}R^\circ$ to mean either $(\mathsf{F}R)^\circ$ or $\mathsf{F}(R^\circ)$, a convention we henceforth adopt.

Exercises

5.1 Give an example of a non-monotonic functor $\mathsf{F} : \mathbf{Rel} \leftarrow \mathbf{Rel}$.

5.2 Show that any relator preserves meets of coreflexives, that is,

$$\mathsf{F}(X \cap Y) \quad = \quad \mathsf{F}X \cap \mathsf{F}Y,$$

for all $X, Y \subseteq id$. Is the restriction $X, Y \subseteq id$ necessary?

5.3 Denoting the set of all functions $A \leftarrow X$ by A^X, the exponential functor $(\_)^X$: $\mathbf{Fun} \leftarrow \mathbf{Fun}$ is defined on arrows by $f^X\, h = f \cdot h$. Is the exponential functor a relator? What is its generalisation to relations?

5.4 Consider a functor $\mathsf{F} : \mathbf{Fun} \leftarrow \mathbf{Fun}$ defined on objects by

$$\mathsf{F}A \quad = \quad \begin{cases} \{\}, & \text{if } A = \{\} \\ \{0\}, & \text{otherwise.} \end{cases}$$

This defines the action of F on arrows uniquely; what is it?

Now suppose F can be extended to a relator on **Rel**. Consider the constant functions $one, two : \{1,2\} \leftarrow \{0\}$ returning 1 and 2 respectively. Use the definition of F to show that $\mathsf{F}(one^\circ \cdot two) = id$, where $id : \{0\} \leftarrow \{0\}$.

Now use the assumption that F preserves converse to show that $\mathsf{F}(one^\circ \cdot two) = \emptyset$. (This exercise shows that not every functor of **Fun** can be extended to a monotonic functor of **Rel**.)

5.5 Show that any relator preserves the domain operator

$$\mathsf{F}(dom R) \quad = \quad dom(\mathsf{F}R).$$

5.2 Relational products

Let us now see how we can extend the product functor to a relator. Recall from the discussion of units in the preceding chapter that *Fun*(**A**) has products whenever **A** is a unitary tabular allegory. Recall also that $(outl, outr)$ is the tabulation of Π and the pairing operator satisfies

$$\langle f, g\rangle \quad = \quad (outl^\circ \cdot f) \cap (outr^\circ \cdot g).$$

This immediately suggests how pairing might be generalised to relations: define

$$\langle R, S\rangle \quad = \quad (outl^\circ \cdot R) \cap (outr^\circ \cdot S). \tag{5.1}$$

The interpretation of $\langle R, S\rangle$ in **Rel** is, of course, that $(a, b)\langle R, S\rangle c$ when aRc and bSc. Given (5.1), we can define $\times$ in the normal way by

$$R \times S \quad = \quad \langle R \cdot outl, S \cdot outr\rangle. \tag{5.2}$$

Note that the same sign $\times$ is used for the generalised product construction (hereafter called the *relational* product) in **A** as for the categorical product in the subcategory *Fun*(**A**). However, relational product is *not* a categorical product.

The task now is to show that relational product is a monotonic bifunctor. First, it is clear that $\langle R, S\rangle$ is monotonic both in R and S, and from the definition of $\times$ we obtain by a short proof that $(R \times S)^\circ = R^\circ \times S^\circ$. Thus $\times$ preserves converse. Furthermore, $\times$ preserves identity arrows (because they are functions), so the nub of the matter is to show that it also preserves composition, that is,

$$(R \times S) \cdot (U \times V) \quad = \quad (R \cdot U) \times (S \cdot V).$$

This result follows from the more general absorption property

$$(R \times S) \cdot \langle X, Y\rangle \quad = \quad \langle R \cdot X, S \cdot Y\rangle. \tag{5.3}$$

In one direction ($\subseteq$), the proof of (5.3) is easy: expand the definitions, use monotonicity and the fact that *outl* and *outr* are simple. We leave details as an exercise. The proof in the other direction is a little tricky, so we will break it into stages. Below, we will give a direct proof of the special case

$$\langle R \cdot X, Y \rangle \;\subseteq\; (R \times id) \cdot \langle X, Y \rangle. \tag{5.4}$$

By a symmetrical argument, we also obtain the special case

$$\langle X, S \cdot Y \rangle \;\subseteq\; (id \times S) \cdot \langle X, Y \rangle. \tag{5.5}$$

Now we argue:

$$\begin{array}{ll} & \langle R \cdot X, S \cdot Y \rangle \\ \subseteq & \quad \{(5.4)\} \\ & (R \times id) \cdot \langle X, S \cdot Y \rangle \\ \subseteq & \quad \{(5.5)\} \\ & (R \times id) \cdot (S \times id) \cdot \langle X, Y \rangle \\ \subseteq & \quad \{\text{since } (R \times id) \cdot (id \times S) \subseteq (R \times S) \text{ (exercise)}\} \\ & (R \times S) \cdot \langle X, Y \rangle. \end{array}$$

To prove (5.4) we argue:

$$\begin{array}{ll} & \langle R \cdot X, Y \rangle \\ = & \quad \{(5.1)\} \\ & (outl^\circ \cdot R \cdot X) \cap (outr^\circ \cdot Y) \\ = & \quad \{\text{claim: } outl \cdot (R \times id) = R \cdot outl \text{ for all } R\text{; converse}\} \\ & ((R \times id) \cdot outl^\circ \cdot X) \cap (outr^\circ \cdot Y) \\ \subseteq & \quad \{\text{modular law}\} \\ & (R \times id) \cdot ((outl^\circ \cdot X) \cap ((R^\circ \times id) \cdot outr^\circ \cdot Y)) \\ \subseteq & \quad \{\text{claim: } outr \cdot (R \times S) \subseteq S \cdot outr \text{ for all } R, S\text{; converse}\} \\ & (R \times id) \cdot ((outl^\circ \cdot X) \cap (outr^\circ \cdot Y)) \\ = & \quad \{(5.1)\} \\ & (R \times id) \cdot \langle X, Y \rangle \end{array}$$

In the above calculation we appealed to two claims, both of which follow from the more general facts

$$outl \cdot \langle R, S \rangle \;=\; R \cdot dom\, S \tag{5.6}$$

$$outr \cdot \langle R, S \rangle \;=\; S \cdot dom\, R. \tag{5.7}$$

The proof of (5.6) is

$$
\begin{array}{cl}
 & outl \cdot \langle R, S\rangle \\
= & \quad \{(5.1)\} \\
 & outl \cdot ((outl^\circ \cdot R) \cap (outr^\circ \cdot S)) \\
= & \quad \{\text{modular identity, since } outl \text{ simple; } outl \cdot outl^\circ = id\} \\
 & R \cap (outl \cdot outr^\circ \cdot S) \\
= & \quad \{\text{since } outl \cdot outr^\circ = \Pi\} \\
 & R \cap (\Pi \cdot S) \\
= & \quad \{\text{Exercise (4.27)}\} \\
 & R \cdot dom\, S.
\end{array}
$$

The proof of (5.7) is symmetrical.

Equation (5.6) indicates why $(outl, outr)$ does not form a categorical product in the allegorical setting: for any arrow R, we have $outl \cdot \langle R, \emptyset\rangle = \emptyset$, not R.

Finally, let us prove the following useful cancellation law:

$$\langle R, S\rangle^\circ \cdot \langle X, Y\rangle \quad = \quad (R^\circ \cdot X) \cap (S^\circ \cdot Y). \tag{5.8}$$

The proof is

$$
\begin{array}{cl}
 & \langle R, S\rangle^\circ \cdot \langle X, Y\rangle \\
= & \quad \{\text{converse, absorption (backwards)}\} \\
 & \langle id, id\rangle^\circ \cdot (R^\circ \times S^\circ) \cdot \langle X, Y\rangle \\
= & \quad \{(5.3)\} \\
 & \langle id, id\rangle^\circ \cdot \langle R^\circ \cdot X, S^\circ \cdot Y\rangle \\
= & \quad \{(5.1)\} \\
 & \langle id, id\rangle^\circ \cdot ((outl^\circ \cdot R^\circ \cdot X) \cap (outr^\circ \cdot S^\circ \cdot Y)) \\
= & \quad \{\text{distribution over meet, since } \langle id, id\rangle \text{ is a function}\} \\
 & ((\langle id, id\rangle^\circ \cdot outl^\circ \cdot R^\circ \cdot X) \cap ((\langle id, id\rangle^\circ \cdot outr^\circ \cdot S^\circ \cdot Y) \\
= & \quad \{\text{products}\} \\
 & (R^\circ \cdot X) \cap (S^\circ \cdot Y).
\end{array}
$$

It is worth while observing that *all* the above equations and inclusions can also be proved by an appeal to the meta-theorem of Section 4.3. Such indirect proofs are quite short as compared to the excruciating symbol manipulation found above. On the other hand, practice in the style of calculation given here will be useful later on when the meta-theorem cannot be applied.

Exercises

5.6 Prove that $(R \times id) \cdot (id \times S) \supseteq (R \times S)$ using only (5.4) and (5.2).

5.7 Show that $\langle P, Q\rangle \cdot \langle R, S\rangle^{\circ} \subseteq (P \cdot R^{\circ}) \times (Q \cdot S^{\circ})$.

5.8 Prove that $(R \times S) \cdot \langle X, Y\rangle \subseteq \langle R \cdot X, S \cdot Y\rangle$.

5.9 Prove that $\langle R, S\rangle \cdot f = \langle R \cdot f, S \cdot f\rangle$. Is this equation true when f is replaced by an arbitrary arrow?

5.10 Let $\mathsf{F} : \mathbf{A} \leftarrow \mathbf{A}$ be a relator. Define $\mathit{unzip}\,(\mathsf{F}) = \langle \mathsf{F}\mathit{outl}, \mathsf{F}\mathit{outr}\rangle$. Prove that

$$\mathit{unzip}\,(\mathsf{F}) \cdot \mathsf{F}(R \times S) \quad = \quad (\mathsf{F}R \times \mathsf{F}S) \cdot \mathit{unzip}\,(\mathsf{F}).$$

for all R, S. (Hint: first consider the case $S = id$.)

5.11 Recall the definition of exponentials from Chapter 3 An *exponential* of two objects A and B is an object A^B and an arrow $\mathit{eval} : A \leftarrow A^B \times B$ such that for each $f : A \leftarrow C \times B$ there is a unique arrow $\mathit{curry}\, f : A^B \leftarrow C$ such that

$$g = \mathit{curry}\, f \quad \equiv \quad \mathit{eval} \cdot (g \times id) = f.$$

Reading $(\times)$ as relational product, does **Rel** have exponentials?

5.3 Relational coproducts

Fortunately, coproducts are simpler than products, at least in the setting of power allegories. Let $(\mathit{inl}, \mathit{inr}, A + B)$ be the coproduct of A and B in $\mathit{Fun}(\mathbf{A})$, where $\mathbf{A}$ is a power allegory. Then it is also a coproduct in the whole allegory $\mathbf{A}$:

$$\begin{array}{ll} & T \cdot \mathit{inl} = R \text{ and } T \cdot \mathit{inr} = S \\ \equiv & \quad \{\text{power transpose isomorphism}\} \\ & \Lambda(T \cdot \mathit{inl}) = \Lambda R \text{ and } \Lambda(T \cdot \mathit{inr}) = \Lambda S \\ \equiv & \quad \{\ \Lambda \text{ fusion (backwards)}\} \\ & \Lambda T \cdot \mathit{inl} = \Lambda R \text{ and } \Lambda T \cdot \mathit{inr} = \Lambda S \\ \equiv & \quad \{\text{coproduct of functions}\} \\ & \Lambda T = [\Lambda R, \Lambda S] \\ \equiv & \quad \{\Lambda\text{-cancellation}\} \\ & T = \in \cdot [\Lambda R, \Lambda S]. \end{array}$$

Hence we can define $[R, S] = \in \cdot [\Lambda R, \Lambda S]$. The following diagram illustrates this calculation:

$$\begin{tikzcd}
& A+B \arrow[d, dashed, "{[\Lambda R, \Lambda S]}"] & \\
A \arrow[ur, "inl"] \arrow[r, "\Lambda R"'] \arrow[dr, "R"'] & \mathsf{P}C \arrow[d, dashed, "\in"] & B \arrow[ul, "inr"'] \arrow[l, "\Lambda S"] \arrow[dl, "S"] \\
& C &
\end{tikzcd}$$

The border arrows of the diagram also suggests an explicit formula for $[R, S]$:

$$[R, S] \quad = \quad (R \cdot inl^\circ) \cup (S \cdot inr^\circ). \tag{5.9}$$

The proof of (5.9) is left as Exercise 5.12.

Given the definition of $[R, S]$, we can define the coproduct relator $+$ in the usual way by

$$R + S \quad = \quad [inl \cdot R, inr \cdot S]. \tag{5.10}$$

It is easy to check that $+$ is monotonic in both arguments, so $+$ is a relator. In analogy with products, we obtain the useful cancellation law

$$[R, S] \cdot [U, V]^\circ \quad = \quad (R \cdot U^\circ) \cup (S \cdot V^\circ). \tag{5.11}$$

The proof is easier than the corresponding cancellation law for products, and details are left as an exercise.

Exercises

5.12 Define $X = [id, \emptyset]$. Use the universal property of coproducts to show that $X \cdot inl = id$ and $X \cdot inr = \emptyset$. Give the corresponding equations satisfied by $Y = [\emptyset, id]$. Hence prove that

$$(inl \cdot X) \ \cup \ (inr \cdot Y) \ = \ [inl, inr] \ = \ id.$$

Now use Proposition 4.1 to conclude that $X = inl^\circ$ and $Y = inr^\circ$. Hence prove

$$[R, S] \quad = \quad (R \cdot inl^\circ) \ \cup \ (S \cdot inr^\circ).$$

5.13 Prove (5.11). Why not say simply that this result follows by duality from the corresponding law for products?

5.14 Prove the equation

$$(R + S) \cap ([U, V]^\circ \cdot [P, Q]) \quad = \quad (R \cap (U^\circ \cdot P)) + (S \cap (V^\circ \cdot Q)).$$

5.4 The power relator

Next we show how to extend P to a relator. Recall from the last chapter that, over functions, P was defined as the restriction of existential image to functions: $\mathsf{P} = \mathsf{EJ}$. Furthermore, we know that $\mathsf{P}f = \Lambda(f \cdot \in)$ because $\mathsf{E}R = \Lambda(R \cdot \in)$, and we also know from the final section of the preceding chapter the explicit formula $\Lambda R = (\in\backslash R) \cap (R\backslash\in)^\circ$ for Λ. Putting these bits together we obtain

$$\mathsf{P}f \quad = \quad (\in\backslash(f \cdot \in)) \cap ((f \cdot \in)\backslash\in)^\circ.$$

The second term can be rephrased using the fact that $(f \cdot R)\backslash S = R\backslash(f^\circ \cdot S)$, so

$$\mathsf{P}f \quad = \quad (\in\backslash(f \cdot \in)) \cap ((\ni \cdot f)/\ni),$$

where $\ni$ is a convenient abbreviation for $\in^\circ$.

The last identity suggests the following generalisation of P to relations:

$$\mathsf{P}R \quad = \quad (\in\backslash(R \cdot \in)) \cap ((\ni \cdot R)/\ni).$$

In **Rel**, this reads

$$x(\mathsf{P}R)y \quad \equiv \quad (\forall a \in x : \exists b \in y : aRb) \;\wedge\; (\forall b \in y : \exists a \in x : aRb).$$

In words, if $x(\mathsf{P}R)y$, then every element of x is related by R to some element in y and conversely.

It is immediate from the monotonicity of division that $\mathsf{P}R$ is monotonic in R. We also have that $\mathsf{P}id = id$, since this is just a restatement of the anti-symmetry of *subset*, proved in the preceding chapter. So to show that P is a relator, we are left with the task of proving that P distributes over composition.

The proof is in two parts. To show that

$$\mathsf{P}R \cdot \mathsf{P}S \quad \subseteq \quad \mathsf{P}(R \cdot S),$$

observe that the right-hand side is the meet of two relations. By the universal properties of meet and division, the result follows if we can prove the inclusions

$$\begin{aligned} \in \cdot\, \mathsf{P}R \cdot \mathsf{P}S \quad &\subseteq \quad R \cdot S \cdot \in \\ \mathsf{P}R \cdot \mathsf{P}S \cdot \ni \quad &\subseteq \quad \ni \cdot\, R \cdot S. \end{aligned}$$

Both follow from the definition of P and the cancellation laws of division.

Now for the hard part, which is to show that $\mathsf{P}(R \cdot S) \subseteq \mathsf{P}R \cdot \mathsf{P}S$. The proof involves tabulations. Let (x, z) be a tabulation of $\mathsf{P}(R \cdot S)$ and define

$$y \quad = \quad \Lambda((R^\circ \cdot \in \cdot\, x) \cap (S \cdot \in \cdot\, z)).$$

We aim to justify the following steps:

$$\mathsf{P}(R \cdot S) = x \cdot z^\circ \subseteq x \cdot y^\circ \cdot y \cdot z^\circ \subseteq \mathsf{P}R \cdot \mathsf{P}S.$$

The first inclusion follows because y is a function and functions are entire. For the second inclusion, we prove that $x \cdot y^\circ \subseteq \mathsf{P}R$, and appeal to the symmetrical argument to obtain $y \cdot z^\circ \subseteq \mathsf{P}S$.

By definition of division, $x \cdot y^\circ \subseteq \mathsf{P}R$ is equivalent to

$$\in \cdot\, x \cdot y^\circ \subseteq R \cdot \in \quad \text{and} \quad x \cdot y^\circ \cdot \ni\, \subseteq\, \ni \cdot R.$$

For the first inclusion, we argue

$$
\begin{array}{ll}
 & \in \cdot\, x \cdot y^\circ \subseteq R \cdot \in \\
\equiv & \quad \{\text{shunting of functions}\} \\
 & \in \cdot\, x \subseteq R \cdot \in \cdot\, y \\
\equiv & \quad \{\text{definition of } y,\ \Lambda\text{-cancellation}\} \\
 & \in \cdot\, x \subseteq R \cdot ((R^\circ \cdot \in \cdot\, x) \cap (S \cdot \in \cdot\, z)) \\
\Leftarrow & \quad \{\text{modular law}\} \\
 & \in \cdot\, x \subseteq (\in \cdot\, x) \cap (R \cdot S \cdot \in \cdot\, z) \\
\equiv & \quad \{\text{definition of meet}\} \\
 & \in \cdot\, x \subseteq R \cdot S \cdot \in \cdot\, z \\
\equiv & \quad \{\text{shunting of } z\text{; division}\} \\
 & x \cdot z^\circ \subseteq\, \in \backslash (R \cdot S \cdot \in) \\
\equiv & \quad \{\text{since } x \cdot z^\circ = \mathsf{P}(R \cdot S)\} \\
 & \mathit{true}.
\end{array}
$$

The second inclusion is proved as follows:

$$
\begin{array}{ll}
 & x \cdot y^\circ \cdot \ni \\
= & \quad \{\text{definition of } y,\ \Lambda \text{ cancellation}\} \\
 & x \cdot ((R^\circ \cdot \in \cdot\, x) \cap (S \cdot \in \cdot\, z))^\circ \\
\subseteq & \quad \{\text{monotonicity, converse}\} \\
 & x \cdot x^\circ \cdot \ni \cdot R \\
\subseteq & \quad \{\text{since } x \text{ is simple}\} \\
 & \ni \cdot R.
\end{array}
$$

This completes the proof.

Exercises

5.15 A relator is completely determined by its action on functions. Since P and E coincide on functions, they are equal. What is wrong with this argument?

5.16 Prove that $\mathsf{E}R \cdot \mathsf{P}(dom\, R) \subseteq \mathsf{P}R$.

5.5 Relational catamorphisms

Since the type functors of *Fun*(**A**) are defined as catamorphisms, we first check that catamorphisms can be extended to include relational algebras.

Let F be a relator and suppose that F has initial algebra $\alpha : T \leftarrow \mathsf{F}T$ in the subcategory of functions. By analogy with the above discussion of coproducts, we can show that α is also initial in the whole allegory:

$$(X \cdot \alpha = R \cdot \mathsf{F}X) \quad \equiv \quad (X = \in \cdot \llparenthesis \Lambda(R \cdot \mathsf{F}\in) \rrparenthesis). \tag{5.12}$$

The proof is:

$$\begin{array}{ll} & X \cdot \alpha = R \cdot \mathsf{F}X \\ \equiv & \quad \{\Lambda \text{ is an isomorphism}\} \\ & \Lambda(X \cdot \alpha) = \Lambda(R \cdot \mathsf{F}X) \\ \equiv & \quad \{\Lambda \text{ cancellation (backwards)}\} \\ & \Lambda(X \cdot \alpha) = \Lambda(R \cdot \mathsf{F}(\in \cdot \Lambda X)) \\ \equiv & \quad \{\text{relators; } \Lambda \text{ fusion (backwards, twice)}\} \\ & \Lambda X \cdot \alpha = \Lambda(R \cdot \mathsf{F}\in) \cdot \mathsf{F}\Lambda X \\ \equiv & \quad \{\text{catamorphisms of functions}\} \\ & \Lambda X = \llparenthesis \Lambda(R \cdot \mathsf{F}\in) \rrparenthesis \\ \equiv & \quad \{\Lambda \text{ cancellation}\} \\ & X = \in \cdot \llparenthesis \Lambda(R \cdot \mathsf{F}\in) \rrparenthesis. \end{array}$$

The proof is summarised in the following diagram:

$$\begin{tikzcd}
T \arrow[d, "\llparenthesis \Lambda(R \cdot \mathsf{F}\in) \rrparenthesis"'] & \mathsf{F}T \arrow[l, "\alpha"'] \arrow[d, "\mathsf{F}\llparenthesis \Lambda(R \cdot \mathsf{F}\in) \rrparenthesis"] \\
\mathsf{P}A \arrow[d, "\in"'] & \mathsf{FP}A \arrow[l, "\Lambda(R \cdot \mathsf{F}\in)"] \arrow[d, "\mathsf{F}\in"] \\
A & \mathsf{F}A \arrow[l, "R"]
\end{tikzcd}$$

It follows that we can define $(\![R]\!)$ by the equation

$$(\![R]\!) \quad = \quad \in \cdot (\![\Lambda(R \cdot \mathsf{F}\in)]\!).$$

Equivalently, $\Lambda(\![R]\!) = (\![\Lambda(R \cdot \mathsf{F}\in)]\!)$. This identity was first exploited in (Eilenberg and Wright 1967) to reason algebraically about the equivalence between deterministic and nondeterministic automata. For this reason we will refer to it subsequently as the *Eilenberg–Wright Lemma*.

Type relators

Let F be a binary relator with initial type (α, T), so T is a type functor. To show that T is a relator, it is sufficient to prove that it preserves converse:

$$\begin{array}{ll}
 & (\mathsf{T}R)^\circ = \mathsf{T}(R^\circ) \\
\equiv & \quad \{\text{definition of } \mathsf{T}, \text{ catamorphisms}\} \\
 & (\mathsf{T}R)^\circ \cdot \alpha = \alpha \cdot \mathsf{F}((\mathsf{T}R)^\circ, R^\circ) \\
\equiv & \quad \{\text{since } \alpha \text{ is an isomorphism}\} \\
 & (\mathsf{T}R)^\circ = \alpha \cdot \mathsf{F}((\mathsf{T}R)^\circ, R^\circ) \cdot \alpha^\circ \\
\equiv & \quad \{\text{converse, } \mathsf{F} \text{ relator}\} \\
 & \mathsf{T}R = \alpha \cdot \mathsf{F}(\mathsf{T}R, R) \cdot \alpha^\circ \\
\equiv & \quad \{\text{as before}\} \\
 & \mathit{true}.
\end{array}$$

Exercises

5.17 Provided the allegory is Boolean, every coreflexive C can be associated with another coreflexive $\sim C$ such that

$$C \cap {\sim} C = \emptyset \quad \text{and} \quad C \cup {\sim} C = \mathit{id}.$$

For any coreflexive C define *guard* $C = [C, {\sim}C]^\circ$. Prove that *guard* C is a function. Define the *conditional* $(C \to R, S)$ by

$$(C \to R, S) \quad = \quad (R + S) \cdot \mathit{guard}\ C.$$

Now prove that conditionals are very similar to cases:

$$R \subseteq (C \to S, T) \quad \equiv \quad (R \cdot C \subseteq S) \text{ and } (R \cdot {\sim}C \subseteq T).$$

This useful exercise gives us another way of modelling conditional expressions, and is the one that we will adopt in the future.

5.18 Let F be a binary relator, with initial type (α, T). Suppose that F preserves meets, i.e.,

$$\mathsf{F}(R \cap X, S \cap Y) \quad = \quad \mathsf{F}(R, S) \cap \mathsf{F}(X, Y).$$

Show that T also preserves meets. Note that this proves, for instance, that the list functor preserves meets.

5.19 Prove that $(\![R]\!)$ is entire when R is entire. *Hint:* use reflection to show that $dom\,(\![R]\!) = id$.

5.6 Combinatorial functions

To counter-balance the foregoing, rather technical material, we devote the rest of this chapter to giving some immediate feeling for the increase in descriptive power that one obtains in a relational setting. The functional programmer is familiar with a range of list-theoretic functions that can be used in the specification and solution of many combinatorial problems. In this section, we define some of these functions in terms of relational catamorphisms. All the functions defined below will re-appear in later chapters, so this section is quite important.

We also take the opportunity to fix some conventions. In the second half of the book we will be writing functional programs to solve a number of combinatorial problems and, since cons-lists are given privileged status in functional programming, we will stipulate that, in the future, lists mean cons-lists except where otherwise stated. We will henceforth write *list* rather than *listr* to denote the type functor for lists.

Subsequences

The subsequence relation $subseq : list\ A \leftarrow list\ A$ can be defined by

$$subseq \quad = \quad (\![nil, cons \cup outr]\!).$$

The function $\Lambda subseq$ takes a list x and returns the set of all subsequences of x. This definition is very succinct, but is not available in functional programming, which does not allow either relations or sets. To express $\Lambda subseq$ without using relations, recall the Eilenberg–Wright lemma, which states that

$$\Lambda(\![R]\!) \quad = \quad (\![\Lambda(R \cdot \mathsf{F}(id, \in))]\!).$$

If we can find e and f such that

$$\Lambda([nil, cons \cup outr] \cdot \mathsf{F}(id, \in)) \quad = \quad [e, f],$$

where $\mathsf{F}(A, B) = 1 + (A \times B)$, then we obtain $\Lambda subseq = (\![e, f]\!)$.

To determine e and f we just expand the left-hand side and simplify:

$$\begin{array}{ll} & \Lambda([nil, cons \cup outr] \cdot \mathsf{F}(id, \in)) \\ = & \quad \{\text{definition of } \mathsf{F}\} \\ & \Lambda([nil, cons \cup outr] \cdot (id + id \times \in)) \\ = & \quad \{\text{coproduct}\} \\ & \Lambda[nil, (cons \cup outr) \cdot (id \times \in)] \\ = & \quad \{\text{power transpose of case}\} \\ & [\Lambda nil, \Lambda((cons \cup outr) \cdot (id \times \in))]. \end{array}$$

So we can certainly take $e = \Lambda nil = \tau \cdot nil$, where τ converts its argument into a singleton set.

To find an appropriate expression for f we will need the power transpose of the join of two relations. This is given by

$$\Lambda(R \cup S) \quad = \quad cup \cdot \langle \Lambda R, \Lambda S \rangle,$$

where $cup = \Lambda((\in \cdot outl) \cup (\in \cdot outr))$ is the function that returns the union of two sets. The proof of the above equation is a simple exercise using the universal property of Λ, and we omit details.

Now we continue:

$$\begin{array}{ll} & \Lambda((cons \cup outr) \cdot (id \times \in)) \\ = & \quad \{\text{composition over join, naturality of } outr\} \\ & \Lambda((cons \cdot (id \times \in)) \cup (\in \cdot outr)) \\ = & \quad \{\text{power transpose of join}\} \\ & cup \cdot \langle \Lambda(cons \cdot (id \times \in)), \Lambda(\in \cdot outr) \rangle \\ = & \quad \{\text{power transpose of composition, } cons \text{ and } outr \text{ are functions}\} \\ & cup \cdot \langle \mathsf{P} cons \cdot \Lambda(id \times \in), outr \rangle. \end{array}$$

It follows that we can take

$$f \quad = \quad cup \cdot \langle \mathsf{P} cons \cdot \Lambda(id \times \in), outr \rangle,$$

and so

$$\Lambda subseq \quad = \quad (\![\tau \cdot nil, cup \cdot \langle \mathsf{P} cons \cdot \Lambda(id \times \in), outr \rangle]\!).$$

The final task in implementing $\Lambda subseq$ is to replace the sets by lists. The result is simple enough to see if we write e and f in the form

$$\begin{array}{rcl} e & = & \{[\,]\} \\ f\,(a, xs) & = & \{cons\,(a, x) \mid x \in xs\} \cup xs. \end{array}$$

In the implementation of $\Lambda subseq$ these definitions are replaced by

$$\begin{aligned} e &= [[\,]] \\ f\,(a, xs) &= [cons\,(a, x) \mid x \leftarrow xs] \mathbin{+\!\!+} xs. \end{aligned}$$

In other words, we define the implementation *subseqs* of $\Lambda subseq$ by

$$subseqs \;=\; (\![wrap \cdot nil, cat \cdot \langle list\ cons \cdot cpr, outr\rangle]\!),$$

where $cpr : list\,(A \times B) \leftarrow A \times list\,B$ (short for "cartesian product, right") is defined by

$$cpr\,(a, x) \;=\; [(a, b) \mid b \leftarrow x].$$

To justify the definition of *subseqs* we need the function $setify : \mathsf{P}A \leftarrow list\,A$ that turns a list into the set of its elements:

$$setify \;=\; (\![\omega, cup \cdot (\tau \times id)]\!),$$

where ω returns the empty set. With the help of *setify* we can formalise the relationship between each set-theoretic operation and the list-theoretic function that implements it. For instance,

$$\begin{aligned} setify \cdot nil &= \omega \\ setify \cdot wrap &= \tau \\ setify \cdot cat &= cup \cdot (setify \times setify) \\ setify \cdot concat &= union \cdot setify \cdot list\,setify \\ setify \cdot list\,f &= \mathsf{P}f \cdot setify \\ setify \cdot cpr &= \Lambda(id \times \in) \cdot (id \times setify). \end{aligned}$$

Using these identities, it is easy to show by an appeal to fusion that

$$setify \cdot subseqs \;=\; \Lambda subseq.$$

We leave the details as an exercise.

Cartesian product

The function *cpr* used above implements one member of an important class of combinators associated with cartesian product. Two other functions, *cpp* and *cpl*, defined by

$$\begin{aligned} cpp\,(x, y) &= [(a, b) \mid a \leftarrow x,\ b \leftarrow y] \\ cpl\,(x, b) &= [(a, b) \mid a \leftarrow x], \end{aligned}$$

implement $\Lambda(\in \times \in)$ and $\Lambda(\in \times id)$, respectively. Thus,

$$\begin{aligned} setify \cdot cpp &= \Lambda(\in \times \in) \\ setify \cdot cpl &= \Lambda(\in \times id). \end{aligned}$$

All three functions are among the combinators catalogued in the Appendix for useful point-free programming.

The three functions are examples of a more general pattern. For a relator F, the function

$$cp\,(\mathsf{F}) : \mathsf{PF}A \leftarrow \mathsf{FP}A$$

is defined by $cp\,(\mathsf{F}) = \Lambda\mathsf{F}(\in)$. In particular, cpp is an implementation of $cp\,(\mathsf{F})$ for $\mathsf{F}A = A \times A$, and cpl is a similar implementation when $\mathsf{F}A = A \times B$.

As another example, consider $cp\,(list)$, which is described informally by

$$cp\,(list)\,[x_1, x_2, \ldots, x_n] \quad = \quad \{[a_1, a_2, \ldots, a_n] \mid a_j \in x_j\}.$$

Since $list\ R = (\![nil, cons \cdot (R \times id)]\!)$, appeal to the Eilenberg–Wright lemma gives

$$cp\,(list) \quad = \quad (\![\Lambda[nil, cons \cdot (\in \times \in)]]\!).$$

Expanding this definition, we obtain

$$cp\,(list) = (\![\Lambda nil, \Lambda(cons \cdot (\in \times \in))]\!) = (\![\tau \cdot nil, \mathsf{P}cons \cdot \Lambda(\in \times \in)]\!).$$

The function $cp\,(list)$ is implemented by a function $cplist : list\,(list\ A) \leftarrow list\,(list\ A)$ obtained by representing sets by lists in the way we have seen above. The result is:

$$cplist \quad = \quad (\![wrap \cdot nil, list\ cons \cdot cpp]\!).$$

The function $cplist$ is another example of a useful combinator for point-free programming. We will meet the cp-family again in Section 8.3.

Prefix and suffix

The relation $prefix$ describes the prefix relation on lists, so $x\ prefix\ y$ when $x +\!\!\!+ z = y$ for some z. Thus, $prefix = outl \cdot cat^{\circ}$. Alternatively, we can define $prefix = init^*$, where $init^*$ is the reflexive transitive closure of the relation $init = outl \cdot snoc^{\circ}$ that removes the last element of a list. The reflexive transitive closure R^* of a relation R will be defined formally in the following chapter.

We can also describe $prefix$ by a relational catamorphism:

$$prefix \quad = \quad (\![nil, nil \cup cons]\!).$$

The first *nil* in the catamorphism has type $list\ A \leftarrow 1$, while the second has type $list\ A \leftarrow A \times list\ A$. Strictly speaking, we should write the second one in the form $nil \cdot !$, where $! : 1 \leftarrow A \times list\ A$.

Applying the Eilenberg–Wright lemma to the given definition of *prefix*, we find

$$\Lambda prefix \quad = \quad (\![\tau \cdot nil, cup \cdot \langle \tau \cdot nil, \mathsf{P}\, cons \cdot \Lambda(id \times \in)\rangle]\!).$$

Replacing sets by lists in the usual way, we obtain an implementation of $\Lambda prefix$ by a function *inits* defined by

$$inits \quad = \quad (\![wrap \cdot nil, cat \cdot \langle wrap \cdot nil, list\ cons \cdot cpr\rangle]\!).$$

This translates to two familiar equations:

$$\begin{aligned} inits\,[\,] &= [[\,]] \\ inits\,([a] +\!\!\!+\, x) &= [[\,]] +\!\!\!+\, [[a] +\!\!\!+\, y \mid y \leftarrow inits\,x]. \end{aligned}$$

Note that *inits* returns a list of initial segments in increasing order of length.

The relation *suffix* is dual to *prefix*, but we have to use snoc-lists to describe it as a relational catamorphism. Alternatively, $suffix = tail^*$, where $tail = outr \cdot cons^\circ$ removes the first element from a list. The implementation of $\Lambda suffix$ is by a function *tails* that returns a list of tail segments in decreasing order of length. The two equations defining *tails* are

$$\begin{aligned} tails\,[\,] &= [[\,]] \\ tails\,(x +\!\!\!+\, [a])) &= [y +\!\!\!+\, [a] \mid y \leftarrow tails\,x] +\!\!\!+\, [[\,]]. \end{aligned}$$

This is not a legitimate implementation in most functional languages. Instead, one can use the two equations

$$\begin{aligned} tails\,[\,] &= [[\,]] \\ tails\,([a] +\!\!\!+\, x)) &= [[a] +\!\!\!+\, x] +\!\!\!+\, tails\,x. \end{aligned}$$

We will see in Section 6.7 how these equations are obtained. Alternatively, *tails* can be implemented as a catamorphism on cons-lists:

$$tails \quad = \quad (\![wrap \cdot nil, extend]\!),$$

where

$$extend\,(a, [x] +\!\!\!+\, xs) \quad = \quad [[a] +\!\!\!+\, x] +\!\!\!+\, [x] +\!\!\!+\, xs.$$

We can put *inits* and *tails* together to give an implementation of the function Λcat° that splits a list in all possible ways:

$$splits \quad = \quad zip \cdot \langle inits, tails\rangle.$$

For this implementation to work it is essential that *inits* and *tails* order their segments in opposite ways.

Partitions

A *partition* of a list is a decomposition of the list into a list of non-empty contiguous segments. For instance, the set of partitions of $[1, 2, 3]$ is

$$\{[[1],[2],[3]],\ \ [[1,2],[3]],\ \ [[1],[2,3]],\ \ [[1,2,3]]\}.$$

A surprising number of combinatorial problems can be phrased as problems about partitions, and we will see examples in Chapters 7 and 8. The relation

$$partition : list\,(list^{+}\,A) \leftarrow list\,A$$

is defined by $partition = concat^{\circ}$, where $concat = (\![nil, cat]\!)$ and cat is restricted to the type $list\,A \leftarrow list\,A^{+} \times list\,A$. One can also express *partition* as a catamorphism

$$partition \;=\; (\![nil, new \cup glue]\!),$$

where

$$\begin{aligned} new &= cons \cdot (wrap \times id) \\ glue &= cons \cdot (cons \times id) \cdot assocl \cdot (id \times cons^{\circ}). \end{aligned}$$

The pointwise definitions are

$$\begin{aligned} new\,(a, xs) &= [[a]] +\!\!+\, xs \\ glue\,(a, [x] +\!\!+\, xs) &= [[a] +\!\!+\, x] +\!\!+\, xs. \end{aligned}$$

The definition of *partition* as a catamorphism thus describes a step-by-step procedure, where at each step either a new singleton segment is created, or the next element is 'glued' to the front of the first segment.

The definition of *partition* by a catamorphism is not as perspicuous as its definition by the converse of a catamorphism. The definitions can be shown to be equivalent using a theorem that we will prove in the following chapter. This theorem states that if $R : A \leftarrow \mathsf{F}A$ is a surjective relation, and if $f : \mathsf{T}\,A \leftarrow A$ satisfies $f \cdot R \subseteq \alpha \cdot \mathsf{F}(id, f)$, where T is the type functor induced by T, then $f^{\circ} = (\![R]\!)$.

We will apply the theorem with $f = concat$ and $R = [nil, new \cup glue]$. We have to show that

$$\begin{aligned} concat \cdot nil &\subseteq nil \\ concat \cdot new &\subseteq cons \cdot (id \times concat) \\ concat \cdot glue &\subseteq cons \cdot (id \times concat). \end{aligned}$$

We prove only the third inclusion:

$$
\begin{array}{cl}
& \mathit{concat} \cdot \mathit{glue} \\
= & \quad \{\text{definition of } \mathit{glue}\} \\
& \mathit{concat} \cdot \mathit{cons} \cdot (\mathit{cons} \times \mathit{id}) \cdot \mathit{assocl} \cdot (\mathit{id} \times \mathit{cons}^{\circ}) \\
= & \quad \{\text{since } \mathit{concat} = (\![\mathit{nil}, \mathit{cat}]\!)\} \\
& \mathit{cat} \cdot (\mathit{cons} \times \mathit{concat}) \cdot \mathit{assocl} \cdot (\mathit{id} \times \mathit{cons}^{\circ}) \\
= & \quad \{\text{naturality of } \mathit{assocl}\} \\
& \mathit{cat} \cdot (\mathit{cons} \times \mathit{id}) \cdot \mathit{assocl} \cdot (\mathit{id} \times (\mathit{id} \times \mathit{concat}) \cdot \mathit{cons}^{\circ}) \\
\subseteq & \quad \{\text{since } \mathit{concat} = (\![\mathit{nil}, \mathit{cat}]\!)\} \\
& \mathit{cat} \cdot (\mathit{cons} \times \mathit{id}) \cdot \mathit{assocl} \cdot (\mathit{id} \times \mathit{cat}^{\circ}) \cdot (\mathit{id} \times \mathit{concat}) \\
= & \quad \{\text{since } \mathit{cat} \cdot (\mathit{cons} \times \mathit{id}) = \mathit{cons} \cdot (\mathit{id} \times \mathit{cat}) \cdot \mathit{assocr}\} \\
& \mathit{cons} \cdot (\mathit{id} \times \mathit{cat}) \cdot \mathit{assocr} \cdot \mathit{assocl} \cdot (\mathit{id} \times \mathit{cat}^{\circ}) \cdot (\mathit{id} \times \mathit{concat}) \\
\subseteq & \quad \{\text{since } \mathit{assocr} \cdot \mathit{assocl} = \mathit{id} \text{ and } \mathit{cat} \cdot \mathit{cat}^{\circ} \subseteq \mathit{id}\} \\
& \mathit{cons} \cdot (\mathit{id} \times \mathit{concat}).
\end{array}
$$

We also have to show that $[\mathit{nil}, \mathit{new} \cup \mathit{glue}]$ is surjective, that is,

$$\mathit{id} \;\; \subseteq \;\; (\mathit{nil} \cdot \mathit{nil}^{\circ}) \;\cup\; ((\mathit{new} \cup \mathit{glue}) \cdot (\mathit{new} \cup \mathit{glue})^{\circ}).$$

We leave it as an exercise to show that

$$
\begin{array}{rcl}
\mathit{new} \cdot \mathit{new}^{\circ} & = & \mathit{cons} \cdot (\mathit{wrap} \cdot \mathit{wrap}^{\circ} \times \mathit{id}) \cdot \mathit{cons}^{\circ} \\
\mathit{glue} \cdot \mathit{glue}^{\circ} & = & \mathit{cons} \cdot (\mathit{cons} \cdot \mathit{cons}^{\circ} \times \mathit{id}) \cdot \mathit{cons}^{\circ}.
\end{array}
$$

Using these equalities, we can now conclude that

$$
\begin{array}{cl}
& (\mathit{nil} \cdot \mathit{nil}^{\circ}) \;\cup\; (\mathit{new} \cdot \mathit{new}^{\circ}) \;\cup\; (\mathit{glue} \cdot \mathit{glue}^{\circ}) \\
= & \quad \{\text{above}\} \\
& (\mathit{nil} \cdot \mathit{nil}^{\circ}) \;\cup\; (\mathit{cons} \cdot (((\mathit{wrap} \cdot \mathit{wrap}^{\circ}) \cup (\mathit{cons} \cdot \mathit{cons}^{\circ})) \times \mathit{id}) \cdot \mathit{cons}^{\circ}) \\
= & \quad \{\text{since } (\mathit{wrap} \cdot \mathit{wrap}^{\circ}) \cup (\mathit{cons} \cdot \mathit{cons}^{\circ}) = \mathit{id} \text{ (on non-empty lists)}\} \\
& (\mathit{nil} \cdot \mathit{nil}^{\circ}) \;\cup\; (\mathit{cons} \cdot \mathit{cons}^{\circ}) \\
= & \quad \{\text{since } \alpha \cdot \alpha^{\circ} = \mathit{id} \text{ for all initial algebras } \alpha\} \\
& \mathit{id},
\end{array}
$$

which gives the result.

By the Eilenberg–Wright lemma, we obtain

$$\Lambda \mathit{partition} \;\; = \;\; (\![\Lambda \mathit{nil}, \Lambda((\mathit{new} \cup \mathit{glue}) \cdot (\mathit{id} \times \in))]\!).$$

We can simplify the second term, arguing:

$$
\begin{array}{ll}
 & \Lambda((new \cup glue) \cdot (id \times \in)) \\
= & \quad \{\text{power transpose of composition}\} \\
 & union \cdot \mathsf{P}\Lambda(new \cup glue) \cdot \Lambda(id \times \in) \\
= & \quad \{\text{power transpose of join}\} \\
 & union \cdot \mathsf{P}(cup \cdot \langle \Lambda new, \Lambda glue \rangle) \cdot \Lambda(id \times \in) \\
= & \quad \{\text{since } new \text{ is a function}\} \\
 & union \cdot \mathsf{P}(cup \cdot \langle \tau \cdot new, \Lambda glue \rangle) \cdot \Lambda(id \times \in).
\end{array}
$$

Finally, we can implement $\Lambda partition$ by a function *partitions* defined by

$$partitions \quad = \quad (\![wrap \cdot nil, concat \cdot list\,(cons \cdot \langle new, glues \rangle) \cdot cpr]\!),$$

where *glues* implements $\Lambda glue$:

$$
\begin{array}{rcl}
glues\,(a, [\,]) & = & [\,] \\
glues\,(a, [x] +\!\!+ xs) & = & [[[a] +\!\!+ x] +\!\!+ xs].
\end{array}
$$

The proof of $setify \cdot partitions = \Lambda partition$ is left as an exercise.

Permutations

Finally, consider the relation *perm* that holds between two lists if one is a permutation of the other. There are a number of ways to specify *perm*; perhaps the simplest is to use the type *bag A* of finite bags over A as an intermediate datatype. This type can be described as a functional F-algebra $[bnil, bcons]$ of the functor $\mathsf{F}(A, B) = 1 + A \times B$. The function *bcons* satisfies the property that

$$bcons\,(a, bcons\,(b, x)) \quad = \quad bcons\,(b, bcons\,(a, x)),$$

which in point-free style reads

$$bcons \cdot (id \times bcons) \quad = \quad bcons \cdot (id \times bcons) \cdot exch,$$

where $exch : B \times (A \times C) \leftarrow A \times (B \times C)$ is the natural isomorphism

$$exch \quad = \quad assocr \cdot (swap \times id) \cdot assocl.$$

The property captures the fact that the order of the elements in a bag is irrelevant but duplicates do have to be taken into account. The function $bagify : bag\,A \leftarrow list\,A$ turns a list into the bag of its elements, and is defined by the catamorphism

$$bagify \quad = \quad (\![bnil, bcons]\!).$$

Since every finite bag is the bag of elements of some list, *bagify* is a surjective function, so $bagify \cdot bagify^{\circ} = id$.

We can now define *perm* by

$$perm \quad = \quad bagify^{\circ} \cdot bagify.$$

In words, x is a permutation of y if the bag of values in x is equal to the bag of values in y. In particular, it follows at once from the definition that $perm = perm^{\circ}$.

The above specification of *perms* does not lead directly to a functional program for computing $\Lambda perm$. One possibility is to express *perm* as a catamorphism $perm = (\![nil, add]\!)$ and then follow the path taken with all the examples given above. It is easy to show (exercise) that

$$perm \cdot cons \quad = \quad perm \cdot cons \cdot (id \times perm),$$

so we can take $add = perm \cdot cons$, although, of course, the result is not useful for computing *perm*. An alternative choice for *add* is the relation

$$add \quad = \quad cat \cdot (id \times cons) \cdot exch \cdot (id \times cat^{\circ}).$$

In words, $add\,(a, x) = y +\!\!+ [a] +\!\!+ z$ where $y +\!\!+ z = x$, so $add\,(a, x)$ adds a somewhere to the list x. Although this definition of *add* is intuitively straightforward, the proof that $perm = (\![nil, add]\!)$ depends on the fact that bags can be viewed as an initial algebra, and we won't go into it. The function Λadd can be implemented as a function *adds* defined by

$$adds\,(a, x) \quad = \quad [y +\!\!+ [a] +\!\!+ z \mid (y, z) \leftarrow splits\; x],$$

where *splits* is the implementation of Λcat° described above. The function *perms* that implements $\Lambda perm$ is given by

$$perms \quad = \quad (\![wrap \cdot nil, list\; add \cdot cpr]\!).$$

We will meet *perm* again in the following chapter when we derive some sorting algorithms.

Exercises

5.20 Construct functions *cup*, *cap* and *cross* so that

$$\begin{aligned} \Lambda(R \cup S) &= cup \cdot \langle \Lambda R, \Lambda S \rangle \\ \Lambda(R \cap S) &= cap \cdot \langle \Lambda R, \Lambda S \rangle \\ \Lambda(R \times S) &= cross \cdot (\Lambda R \times \Lambda S). \end{aligned}$$

5.21 Prove that

$$\begin{aligned} setify \cdot nil &= \omega \\ setify \cdot wrap &= \tau \\ setify \cdot cons &= cup \cdot (\tau \times setify) \\ setify \cdot list\, f &= \mathsf{P}f \cdot setify. \end{aligned}$$

5.22 Prove that $setify \cdot subseqs = \Lambda subseq$.

5.23 Express *subseq* as the converse of a catamorphism. (*Hint:* think about *supersequences.*)

5.24 As a function of type $list\, A \leftarrow list^{+}\, A$, the relation *init* can be defined as a catamorphism. How?

5.25 Prove that

$$\begin{aligned} new \cdot new^{\circ} &= cons \cdot (wrap \cdot wrap^{\circ} \times id) \cdot cons^{\circ} \\ glue \cdot glue^{\circ} &= cons \cdot (cons \cdot cons^{\circ} \times id) \cdot cons^{\circ}. \end{aligned}$$

5.26 Prove that $setify \cdot partitions = \Lambda partition$.

5.27 Show that

$$\Lambda partition \;=\; ([\Lambda nil, cup \cdot \langle \mathsf{P}new, union \cdot \mathsf{P}\Lambda glue \rangle \cdot \Lambda(id \times \in)]),$$

and hence find another implementation of $\Lambda partition$.

5.28 Using $bagify \cdot bagify^{\circ} = id$, show that $perm \cdot cons = perm \cdot cons \cdot (id \times perm)$.

5.29 A list x is an *interleaving* of two sequences y and z if it can be split into a series of subsequences, with alternate subsequences extracted from y and z. For example, $[1, 10, 2, 3, 11, 12, 4]$ is an interleaving of $[1, 2, 3, 4]$ and $[10, 11, 12]$. The relation *interleave* interleaves two lists nondeterministically. Define *interleave* as the converse of a catamorphism.

5.7 Lax natural transformations

As we have seen, reasoning about datatypes in a relational setting makes it possible to explore properties that are difficult or impossible to express in a functional setting. On the other hand, some properties that are simple equalities in a functional setting become inequalities in a relational one. A good example is provided by natural transformations and lax natural transformations. A lax natural transformation is like a natural transformation but the naturality condition becomes an inequation. Formally, a collection of arrows $\phi_A : \mathsf{F}A \leftarrow \mathsf{G}A$ is called a *lax natural* transformation, and is denoted by $\phi : \mathsf{F} \hookleftarrow \mathsf{G}$, if

$$\mathsf{F}R \cdot \phi \ \supseteq \ \phi \cdot \mathsf{G}R \tag{5.13}$$

for all R. Notice the direction $\supseteq$ of the inclusion, which can be remembered by relating it to the shape of the hook in $\hookleftarrow$. The inclusion can be pictured as

$$\begin{tikzcd}
\mathsf{F}A \arrow[d, "\mathsf{F}R"'] \arrow[dr, phantom, "\supseteq"] & \mathsf{G}A \arrow[l, "\phi"'] \arrow[d, "\mathsf{G}R"] \\
\mathsf{F}B & \mathsf{G}B \arrow[l, "\phi"]
\end{tikzcd}$$

As one example of a lax natural transformation, we have $\in \, : id \hookleftarrow \mathsf{P}$; in other words,

$$R \cdot \in \ \supseteq \ \in \cdot \mathsf{P}R,$$

for all R. This follows at once from the definition of $\mathsf{P}R$.

The main result concerning lax natural transformations is the following theorem.

Theorem 5.2 Let $\mathsf{F}, \mathsf{G} : \mathbf{A} \leftarrow \mathbf{B}$ be relators and $\mathsf{J} : \mathbf{B} \leftarrow \mathit{Fun}(\mathbf{B})$ the inclusion of functions into relations. Then $\phi : \mathsf{F} \hookleftarrow \mathsf{G} \equiv \phi : \mathsf{FJ} \leftarrow \mathsf{GJ}$.

Proof. First, assume that $\phi : \mathsf{F} \hookleftarrow \mathsf{G}$, so in particular we have $\mathsf{F}f \cdot \phi \supseteq \phi \cdot \mathsf{G}f$. But also

$$\begin{array}{ll}
 & \mathsf{F}f \cdot \phi \subseteq \phi \cdot \mathsf{G}f \\
\equiv & \quad \{\text{shunting of functions}\} \\
 & \phi \cdot \mathsf{G}f^\circ \subseteq \mathsf{F}f^\circ \cdot \phi \\
\equiv & \quad \{\text{inequation (5.13) with } R = f^\circ\} \\
 & \mathit{true},
\end{array}$$

and so $\mathsf{F}f \cdot \phi = \phi \cdot \mathsf{G}f$ for all f.

Conversely, assume that $\phi : \mathsf{FJ} \leftarrow \mathsf{GJ}$, so $\mathsf{F}g \cdot \phi = \phi \cdot \mathsf{G}g$ for all functions g. By shunting of functions, this gives $\mathsf{F}g^\circ \cdot \phi \supseteq \phi \cdot \mathsf{G}g^\circ$ since F and G are relators and thus preserve converse.

Now, we complete the proof by arguing:

$$
\begin{array}{cl}
 & \mathsf{F}R \cdot \phi \\
= & \quad \{\text{let } (f, g) \text{ be a tabulation of } R\} \\
 & \mathsf{F}(f \cdot g^{\circ}) \cdot \phi \\
= & \quad \{\text{relators}\} \\
 & \mathsf{F}f \cdot \mathsf{F}g^{\circ} \cdot \phi \\
\supseteq & \quad \{\text{above}\} \\
 & \mathsf{F}f \cdot \phi \cdot \mathsf{G}g^{\circ} \\
= & \quad \{\text{since } \phi : \mathsf{FJ} \leftarrow \mathsf{GJ}\} \\
 & \phi \cdot \mathsf{G}f \cdot \mathsf{G}g^{\circ} \\
= & \quad \{\text{relators}\} \\
 & \phi \cdot \mathsf{G}(f \cdot g^{\circ}) \\
= & \quad \{\text{since } f \cdot g^{\circ} \text{ tabulates } R\} \\
 & \phi \cdot \mathsf{F}R.
\end{array}
$$

□

Exercises

5.30 Each of the following pairs is related by $\subseteq$ or $\supseteq$. State in each case what the relationship is:

$$
\begin{array}{rcl}
\mathsf{P}R \cdot \tau & \text{and} & \tau \cdot R \\
(R \times R) \cdot \langle id, id \rangle & \text{and} & \langle id, id \rangle \cdot R \\
cup \cdot (\mathsf{P}R \times \mathsf{P}R) & \text{and} & \mathsf{P}R \cdot cup.
\end{array}
$$

Bibliographical Remarks

The notion of a relator was first introduced in (Kawahara 1973a); the concept then went unnoticed for a long time, until it was reinvented in (Carboni, Kelly, and Wood 1991). Almost simultaneously, Backhouse and his colleagues started to write a series of papers that demonstrated the relevance of relators to computing (Backhouse et al. 1991, 1992; Backhouse and Hoogendijk 1993). The discussion in this chapter owes much to that work. Our definition of relators is more restrictive than that of (Mitchell and Ščedrov 1993).

Several authors have considered the use of relational products in the context of program derivation, e.g. (De Roever 1976). The (sometimes excruciating) symbol manipulations that result from their introduction can be made more attractive by adapting a graphical notation (Brown and Hutton 1994; Curtis and Lowe 1995).

As already mentioned in the text, relational algebras and catamorphisms were first employed in (Eilenberg and Wright 1967) to reason about the equivalence of deterministic and nondeterministic automata. This work was further amplified in (Goguen and Meseguer 1983). Numerous examples of relational catamorphisms can be found in (Meertens 1987), and these examples awakened our own interest in the topic (Bird and De Moor 1993c; De Moor 1992a, 1994). The circuit design language *Ruby* is essentially a relational programming language based on catamorphisms (Jones and Sheeran 1990).

In the context of imperative program derivation, it has been convincingly demonstrated that predicate transformer semantics are preferable to a relational framework (Dijkstra 1976; Dijkstra and Scholten 1990). Just as it is possible to lift the type structure of **Fun** to **Rel**, one can also lift the type structure of **Rel** to an appropriate category of predicate transformers. The key property that makes this possible is that every monotonic predicate transformer can be factored as a pair of relations, in the same way as every relation can be factored in terms of functions (De Moor 1992b; Gardiner, Martin, and De Moor 1994; Martin 1991; Naumann 1994). Although initial results are promising, there is as yet no definitive evidence that this will lead to a useful calculus for program derivation.

Chapter 6

Recursive Programs

The recursive programs we have seen so far have all been based, one way or the other, on catamorphisms. But not all the recursions that arise in programming are homomorphisms of a datatype. For example, one may want to implement the converse of a catamorphism, or a divide and conquer scheme.

To set the scene, we begin this chapter with a simple programming problem whose solution is given by a non-structural recursion. The solutions of a recursion equation are called its *fixed points* and we continue with a discussion of some general properties of fixed points. We then go on to discuss an important class of computations, called *hylomorphisms*, that captures most of the recursive programs one is likely to meet in practice. To illustrate the material, we give applications to the problem of deriving fast exponentiation, one or two sorting algorithms, and an algorithm for computing the closure of a relation.

6.1 Digits of a number

The problem in this section is simply to convert a natural number to its decimal representation. The decimal representation of a nonzero natural number is a list of digits starting with a nonzero digit. The representation of zero is exceptional in this respect, in that it is a list with one element, namely zero itself. Having observed this anomaly, we shall concentrate on deriving an algorithm for converting positive natural numbers.

The first step is to specify the problem formally. The types involved are four in number: the type Nat^+ of positive natural numbers; the type $Digit = \{0, 1, \ldots, 9\}$ of digits; the type $Digit^+ = \{1, 2, \ldots, 9\}$ of nonzero digits; and finally the type of decimal representations, which are non-empty sequences of digits beginning with a nonzero digit. This last type can be declared as

$$Decimal \quad ::= wrap\; Digit^+ \mid snoc\,(Decimal, Digit).$$

Thus, $([wrap, snoc], Decimal)$ is the initial algebra of the functor

$$\mathsf{F}A \quad = \quad Digit^+ + (A \times Digit).$$

The function $val : Nat^+ \leftarrow Decimal$ is a catamorphism

$$val \quad = \quad (\![embed, op]\!),$$

where $embed : Nat^+ \leftarrow Digit^+$ is the inclusion of digits into natural numbers, and $op(n, d) = 10n + d$. To check this, let x be the decimal

$$x \quad = \quad snoc\,(snoc\,(wrap\; d_2, d_1), d_0).$$

Then we have

$$val\,x \quad = \quad 10(10d_2 + d_1) + d_0 \quad = \quad 10^2 d_2 + 10^1 d_1 + 10^0 d_0.$$

We can now specify the function *digits*, which takes a number and returns its decimal representation, by

$$digits \quad \subseteq \quad val^\circ. \tag{6.1}$$

The use of $\subseteq$ rather than $=$ is necessary because we do not know (at least not yet) that val° is a function. One should read (6.1) as requiring a functional refinement of val°. The goal is to synthesise an algorithm from this specification of *digits*.

As a first step, we expand the definition of val°:

$$\begin{array}{ll}
& val^\circ \\
= & \quad \{\text{definition}\} \\
& (\![embed, op]\!)^\circ \\
= & \quad \{\text{catamorphisms}\} \\
& (\,[embed, op] \cdot \mathsf{F}val \cdot [wrap, snoc]^\circ\,)^\circ \\
= & \quad \{\text{converse}\} \\
& [wrap, snoc] \cdot \mathsf{F}val^\circ \cdot [embed, op]^\circ \\
= & \quad \{\text{definition of } \mathsf{F}\} \\
& [wrap, snoc] \cdot (id + val^\circ \times id) \cdot [embed, op]^\circ \\
= & \quad \{\text{coproduct}\} \\
& [wrap, snoc \cdot (val^\circ \times id)] \cdot [embed, op]^\circ \\
= & \quad \{\text{coproduct}\} \\
& (wrap \cdot embed^\circ) \cup (snoc \cdot (val^\circ \times id) \cdot op^\circ).
\end{array}$$

Hence val° satisfies the recursive equation

$$val^\circ \quad = \quad (wrap \cdot embed^\circ) \;\cup\; (snoc \cdot (val^\circ \times id) \cdot op^\circ).$$

In order to see what relation is given by $op^\circ : Nat^+ \times Digit \leftarrow Nat^+$, we reason:

$$
\begin{array}{cl}
 & op(n, d) = m \\
\equiv & \quad \{\text{definition of } op\} \\
 & 10n + d = m \\
\equiv & \quad \{\text{arithmetic and } 0 \leq d < 10\} \\
 & n = m \operatorname{div} 10 \ \wedge \ d = m \bmod 10.
\end{array}
$$

To obtain the right type for op° we need to ensure that $0 < n$ in the above calculation, and this means that we require $10 \leq m$ as a precondition. So op° is a partial function, defined if and only if its argument is at least 10. On the other hand, $embed^\circ$ is also a partial function, defined if and only if its argument is less than 10. The join in the recursive equation for val° can therefore be replaced by a conditional expression, and we obtain

$$
val^\circ\, m \quad = \quad \begin{cases} wrap\, m, & \text{if } m < 10 \\ snoc\,(val^\circ\,(m \operatorname{div} 10), m \bmod 10), & \text{otherwise.} \end{cases}
$$

As a recursive program, this equation determines val° uniquely. The recursion terminates on all arguments because $m \geq 10$ and $n = m \operatorname{div} 10$ together imply $n < m$, and so val° is applied to successively smaller natural numbers. It therefore follows that val° is a (total) function and we can take $digits = val^\circ$.

Writing the result in functional programming style, we obtain the program

$$
digits\, m \quad = \quad \begin{cases} [m], & \text{if } m < 10 \\ digits\,(m \operatorname{div} 10) +\!\!+ [m \bmod 10], & \text{otherwise.} \end{cases}
$$

The program runs in quadratic time because the implementation of *snoc* on lists takes linear time. To obtain a linear-time program we can introduce an accumulation parameter and write $digits\, m = f\,(m, [\,])$, where

$$
f\,(m, x) \quad = \quad \begin{cases} [m] +\!\!+ x, & \text{if } m < 10 \\ f\,(m \operatorname{div} 10, [n \bmod 10] +\!\!+ x), & \text{otherwise.} \end{cases}
$$

Notice that the anomalous case of 0 is treated correctly in the above algorithm.

Simple as it is, the digits of a number example illustrates a basic strategy for program derivation using a relational calculus. First, a function of interest is specified as a refinement of some relation R. Then, after due process of manipulation, R is discovered to be a solution of a certain recursion equation. Finally, the recursion is used to implement the function. As we shall see, the due process of manipulation can often be replaced by an appeal to a single theorem.

Exercises

6.1 Justify the final program above.

6.2 Least fixed points

Catamorphisms are defined as the unique fixed points of certain recursion equations (as are the converses of catamorphisms). Here we are interested in the fact that, when working in a relational context, one may also consider *least* fixed points.

The key result for reasoning about least fixed points is the celebrated Knaster–Tarski theorem (Knaster 1928; Tarski 1955), which in our terminology is as follows:

Theorem 6.1 (Knaster–Tarski) Suppose ϕ is a monotonic mapping (not necessarily a functor) on the arrows of a locally complete allegory, taking a relation $X : A \leftarrow B$ to $\phi X : A \leftarrow B$. Then each of the equations $\phi X \subseteq X$ and $\phi X = X$ has a least solution and these least solutions coincide. Dually, each of the equations $X \subseteq \phi X$ and $X = \phi X$ has a greatest solution and these greatest solutions coincide.

Proof. Let $\mathcal{X} = \{X \mid \phi X \subseteq X\}$ and define $R = \bigcap \mathcal{X}$. We first show that $\phi R \subseteq R$, or, equivalently, that $X \in \mathcal{X}$ implies $\phi R \subseteq X$. We reason:

$$
\begin{array}{ll}
& X \in \mathcal{X} \\
\Rightarrow & \quad \{\text{definition of } R\} \\
& R \subseteq X \\
\Rightarrow & \quad \{\phi \text{ monotonic}\} \\
& \phi R \subseteq \phi X \\
\Rightarrow & \quad \{\text{since } \phi X \subseteq X\} \\
& \phi R \subseteq X.
\end{array}
$$

But now, since $R \in \mathcal{X}$, it follows that $X = R$ is the least solution of $\phi X \subseteq X$. It remains to prove that $R \subseteq \phi R$:

$$
\begin{array}{ll}
& R \subseteq \phi R \\
\Leftarrow & \quad \{\text{definition of } R\} \\
& \phi(\phi R) \subseteq \phi R \\
\Leftarrow & \quad \{\text{since } \phi \text{ monotonic}\} \\
& \phi R \subseteq R \\
\equiv & \quad \{\text{above}\} \\
& \mathit{true}.
\end{array}
$$

□

For brevity we henceforth write $(\mu X : \phi X)$ for the least solution of the equation $X = \phi X$.

Let us now consider what the Knaster–Tarski theorem says about datatypes and catamorphisms. Recall that $([R])$ was defined by the universal property

$$X = ([R]) \quad \equiv \quad X \cdot \alpha = R \cdot \mathsf{F}X,$$

where F is the base relator of the catamorphism. Because α is an isomorphism, the equation on the right can also be written as $X = R \cdot \mathsf{F}X \cdot \alpha^\circ$, so $X = ([R])$ is the unique (and hence both greatest and least) solution of the equation. Since F is a relator, the mapping ϕ defined by $\phi X = R \cdot \mathsf{F}X \cdot \alpha^\circ$ is monotonic. Hence by the Knaster–Tarski theorem we obtain

$$([R]) \subseteq X \quad \Leftarrow \quad R \cdot \mathsf{F}X \cdot \alpha^\circ \subseteq X \tag{6.2}$$

$$X \subseteq ([R]) \quad \Leftarrow \quad X \subseteq R \cdot \mathsf{F}X \cdot \alpha^\circ. \tag{6.3}$$

The fusion law for catamorphisms therefore has two variants in which equality is replaced by inclusion:

$$([T]) \subseteq S \cdot ([R]) \quad \Leftarrow \quad T \cdot \mathsf{F}S \subseteq S \cdot R \tag{6.4}$$

$$S \cdot ([R]) \subseteq ([T]) \quad \Leftarrow \quad S \cdot R \subseteq T \cdot \mathsf{F}S. \tag{6.5}$$

The proofs of these results are easy exercises.

Exercises

6.2 Where in the proof of the Knaster–Tarski theorem did we use the locally complete property of the allegory?

6.3 Say that ϕ is *continuous* if it preserves joins of ascending chains of relations. That is, if $X_0 \subseteq X_1 \subseteq X_2 \ldots$, then $\phi(\bigcup\{X_n \mid 0 \leq n\}) = \bigcup\{\phi X_n \mid 0 \leq n\}$. Prove Kleene's theorem (Kleene 1952) which states that, under the conditions of the Knaster–Tarski theorem and the assumption that ϕ is continuous,

$$(\mu X : \phi X) \quad = \quad \bigcup\{\phi^n \emptyset \mid 0 \leq n\},$$

where $\phi^n X = \phi X \cdot \phi X \cdot \cdots \cdot \phi X$ (n times).

6.4 Use the Knaster–Tarski theorem to justify the following method for showing $(\mu X : \phi X) \subseteq A$: show that $\phi A \subseteq A$.

Use Kleene's theorem to justify the alternative method: show that $X \subseteq A$ implies $\phi X \subseteq A$. This method is called *fixed point induction.*

6.5 If ϕ is a monotonic mapping, then the least solution of $\phi X \subseteq X$ satisfies $\phi X = X$. Show that this is a special case of Lambek's lemma when the partial order of arrows $A \leftarrow B$ is regarded as a category, and ϕ is regarded as a functor on this category.

6.6 Prove (6.4) and (6.5).

6.7 Prove that $(\![R]\!) \subseteq (\![S]\!)$ follows from $R \cdot \mathsf{F}(\![S]\!) \subseteq S \cdot \mathsf{F}(\![S]\!)$. Give an example where it is not true that $R \subseteq S$, but that, nevertheless, $(\![R]\!) \subseteq (\![S]\!)$.

6.8 An arrow is said to be *difunctional* if $R = R \cdot R^\circ \cdot R$. The *difunctional closure* of an arbitrary arrow R is the least difunctional relation that contains R. Construct the difunctional closure of R as a least fixed point.

6.3 Hylomorphisms

The composition of a catamorphism with the converse of a catamorphism is called a *hylomorphism*. Thus hylomorphisms are expressions of the form $(\![R]\!) \cdot (\![S]\!)^\circ$. Hylomorphisms are important because they capture the idea of using an intermediate data structure in the solution of a problem.

More precisely, suppose that $R : A \leftarrow \mathsf{F}A$ and also that $S : B \leftarrow \mathsf{F}B$. Then we have $(\![R]\!) \cdot (\![S]\!)^\circ : A \leftarrow B$, where $(\![R]\!) : A \leftarrow T$ and $(\![S]\!)^\circ : T \leftarrow B$, and where T is the initial type of F. The type T is the intermediate data structure.

Practically every relation of interest can be expressed as a hylomorphism. Since $(\![\alpha]\!) = id$, all catamorphisms and converses of catamorphisms are themselves examples of hylomorphisms. We will see many other examples in due course.

Hylomorphisms can be characterised as least fixed points. More precisely, the following theorem holds:

Theorem 6.2 Suppose that $R : A \leftarrow \mathsf{F}A$ and $S : B \leftarrow \mathsf{F}B$ are two F-algebras. Then $(\![R]\!) \cdot (\![S]\!)^\circ : A \leftarrow B$ is given by

$$(\![R]\!) \cdot (\![S]\!)^\circ \quad = \quad (\mu X : R \cdot \mathsf{F}X \cdot S^\circ).$$

Proof. First we show that $(\![R]\!) \cdot (\![S]\!)^\circ$ is a fixed point:

$$\begin{array}{ll} & R \cdot \mathsf{F}((\![R]\!) \cdot (\![S]\!)^\circ) \cdot S^\circ \\ = & \quad \{\text{functors}\} \\ & R \cdot \mathsf{F}(\![R]\!) \cdot \mathsf{F}(\![S]\!)^\circ \cdot S^\circ \end{array}$$

$$
\begin{aligned}
&= \quad \{\text{catamorphisms}\} \\
&\quad (\![R]\!) \cdot \alpha \cdot \mathsf{F}(\![S]\!)^\circ \cdot S^\circ \\
&= \quad \{\text{converse; catamorphisms}\} \\
&\quad (\![R]\!) \cdot (\![S]\!)^\circ.
\end{aligned}
$$

Second, we show that

$$(\![R]\!) \cdot (\![S]\!)^\circ \subseteq X \quad \Leftarrow \quad R \cdot \mathsf{F}X \cdot S^\circ \subseteq X$$

and appeal to Knaster–Tarski. The proof makes use of the division operation, a typical strategy in reasoning about least fixed points:

$$
\begin{aligned}
&\quad (\![R]\!) \cdot (\![S]\!)^\circ \subseteq X \\
&\equiv \quad \{\text{division}\} \\
&\quad (\![R]\!) \subseteq X/(\![S]\!)^\circ \\
&\Leftarrow \quad \{\text{Knaster–Tarski, and equation (6.2)}\} \\
&\quad R \cdot \mathsf{F}(X/(\![S]\!)^\circ) \cdot \alpha^\circ \subseteq X/(\![S]\!)^\circ \\
&\equiv \quad \{\text{division}\} \\
&\quad R \cdot \mathsf{F}(X/(\![S]\!)^\circ) \cdot \alpha^\circ \cdot (\![S]\!)^\circ \subseteq X \\
&\equiv \quad \{\text{catamorphims}\} \\
&\quad R \cdot \mathsf{F}(X/(\![S]\!)^\circ) \cdot \mathsf{F}(\![S]\!)^\circ \cdot S^\circ \subseteq X \\
&\Leftarrow \quad \{\text{functors and division cancellation}\} \\
&\quad R \cdot \mathsf{F}X \cdot S^\circ \subseteq X.
\end{aligned}
$$

□

When $\mathsf{F}X = \mathsf{G}X + \mathsf{H}X$, so F-algebras are coproducts, we can appeal to the following corollary of Theorem 6.2:

Corollary 6.1

$$(\![R_1, R_2]\!) \cdot (\![S_1, S_2]\!)^\circ \quad = \quad (\mu X : (R_1 \cdot \mathsf{G}X \cdot S_1{}^\circ) \cup (R_2 \cdot \mathsf{H}X \cdot S_2{}^\circ)).$$

Proof. We reason:

$$
\begin{aligned}
&\quad [R_1, R_2] \cdot (\mathsf{G}X + \mathsf{H}X) \cdot [S_1, S_2]^\circ \\
&= \quad \{\text{coproduct}\} \\
&\quad [R_1 \cdot \mathsf{G}X, R_2 \cdot \mathsf{H}X] \cdot [S_1, S_2]^\circ \\
&= \quad \{\text{coproduct}\} \\
&\quad (R_1 \cdot \mathsf{G}X \cdot S_1{}^\circ) \cup (R_2 \cdot \mathsf{H}X \cdot S_2{}^\circ)).
\end{aligned}
$$

□

Theorem 6.2, henceforth called the hylomorphism theorem, can be read as representing a prototypical 'divide and conquer' scheme. The term S° represents the decomposition stage, $\mathsf{F}X$ represents the stage of solving the subproblems recursively, and R represents the recombination stage. We will see applications in the next section and in Section 6.6.

Exercises

6.9 Specify the function that converts the binary representation of a number to its octal representation as a hylomorphism.

6.10 Show that hylomorphisms preserve simplicity: if R is simple and S is simple, then $(\![R]\!) \cdot (\![S^\circ]\!)^\circ$ is simple.

6.4 Fast exponentiation and modulus computation

Consider the problem of computing a^b for natural numbers a and b. The curried function $exp : (Nat \leftarrow Nat) \leftarrow Nat$ is defined by the catamorphism

$$exp\ a \quad = \quad (\![one, mult\ a]\!).$$

This definition encapsulates the two equations $a^0 = 1$ and $a^{b+1} = a \times a^b$. The computation of $exp\ a\ b$ by the catamorphism takes $O(b)$ steps, but by using a divide and conquer scheme we can improve the running time to $O(\log b)$ steps.

To derive the fast exponentiation algorithm consider the type Bin of binary numbers, defined by $Bin = listl\ Bit$, where $Bit = \{0, 1\}$. For example, as an element of Bin the number 6 is given as $[1, 1, 0]$. The partial function $convert : Nat \leftarrow Bin$ converts a well-formed binary number, that is, a sequence of bits that is either empty or begins with a 1, into natural numbers and is given by a snoc-list catamorphism

$$convert \quad = \quad (\![zero, shift]\!),$$

where $shift : Nat^+ \leftarrow Nat \times Bit$ is given by $shift\,(n, d) = 2 \times n + d$.

Now we can argue:

$$\begin{array}{ll}
& exp\ a \\
\supseteq & \quad \{\text{since } convert \text{ is simple}\} \\
& exp\ a \cdot convert \cdot convert^\circ \\
= & \quad \{\text{fusion, see below, with } op\ a\,(n, d) = (d = 0 \to n^2, a \times n^2)\} \\
& (\![one, op\ a]\!) \cdot convert^\circ
\end{array}$$

$$
\begin{array}{ll}
= & \{\text{corollary to hylomorphism theorem}\} \\
 & (\mu X : (one \cdot zero^{\circ}) \cup (op\ a \cdot (X \times id) \cdot shift^{\circ}))
\end{array}
$$

The fusion step is justified by the equations

$$
\begin{array}{lcl}
exp\ a \cdot zero & = & one \\
exp\ a \cdot shift & = & op\ a \cdot (exp\ a \times id).
\end{array}
$$

By construction, *zero* and *shift* have disjoint ranges, so we can proceed as in the digits of a number example and replace the join by a conditional expression. The result is the following program for computing $exp\ a\ b$:

$$
exp\ a\ b \quad = \quad \begin{cases} 1, & \text{if } b = 0 \\ op\ a\,(exp\ a\,(b \text{ div } 2), b \text{ mod } 2), & \text{otherwise.} \end{cases}
$$

Modulus computation

Exactly the same idea can be used to compute a mod b for natural a and positive natural b. The curried function $mod : (Nat \leftarrow Nat) \leftarrow Nat^{+}$ is defined by the catamorphism

$$
mod\ b \quad = \quad (\![zero, succ\ b]\!),
$$

where $succ\ b\ a = (a = b - 1 \rightarrow 0, a + 1)$. The computation of a mod b by this method takes $O(a)$ steps. But, as before, we can argue:

$$
\begin{array}{ll}
 & mod\ b \\
\supseteq & \{\text{since } convert \text{ is simple}\} \\
 & mod\ b \cdot convert \cdot convert^{\circ} \\
= & \{\text{fusion, see below}\} \\
 & (\![zero, op\ b]\!) \cdot convert^{\circ} \\
= & \{\text{hylomorphisms}\} \\
 & (\mu X : (zero \cdot zero^{\circ}) \cup (op\ b \cdot (X \times id) \cdot shift^{\circ})).
\end{array}
$$

The fusion step is justified by the equations

$$
\begin{array}{lcl}
mod\ b \cdot zero & = & zero \\
mod\ b \cdot shift & = & op\ b \cdot (mod\ b \times id),
\end{array}
$$

where $op\ b\,(r, d) = (n \geq b \rightarrow n - b, n)$ and $n = 2 \times r + d$.

The result is the program

$$mod\; b\; a \quad = \quad \begin{cases} 0, & \text{if } a = 0 \\ op\; b\,(mod\; b\,(a \operatorname{div} 2), 0), & \text{if } even\; a \\ op\; b\,(mod\; b\,(a \operatorname{div} 2), 1), & \text{if } odd\; a. \end{cases}$$

The running time is $O(\log a)$ steps.

These simple exercises demonstrate how divide and conquer schemes can be introduced by invoking a suitable intermediate datatype.

Exercises

6.11 Why do the programs for exponentiation and modulus terminate and deliver functions?

6.5 Unique fixed points

The hylomorphism theorem states that a hylomorphism is the least fixed point of a certain recursion equation. However, it is not necessarily the only fixed point. To illustrate, consider the hylomorphism

$$X \quad = \quad (\![zero, id]\!) \cdot (\![zero, positive]\!)^{\circ}$$

on natural numbers, where *positive* is the coreflexive that holds only on positive integers (so $positive = succ \cdot succ^{\circ}$). The catamorphism $(\![zero, id]\!)$ describes the constant function that always returns zero, and $(\![zero, positive]\!)$ describes the coreflexive that holds only on zero. Hence X is again the coreflexive that holds only on zero. However, the recursion equation corresponding to the hylomorphism is

$$X \quad = \quad [zero, id] \cdot (id + X) \cdot [zero, positive]^{\circ},$$

which simplifies to $X = (zero \cdot zero^{\circ}) \cup (X \cdot positive)$. This equation has other solutions, including $X = id$.

Note also that $[zero, positive]^{\circ} : 1 + Nat \leftarrow Nat$ is a function, as is $[zero, id]$, but the least solution of the recursion is not even entire. However, Exercise 6.10 shows that if R and S are simple relations, then so is $(\mu X : R \cdot \mathsf{F}X \cdot S)$.

It is important to know when a recursion equation $X = R \cdot \mathsf{F}X \cdot S^{\circ}$ has a unique solution, and when the solution is a function. It is not sufficient for R and S° to be functions, as we saw above. The condition is simple to state: we need the fact that $member\,(\mathsf{F}) \cdot S^{\circ}$ is an *inductive* relation, where $member\,(\mathsf{F})_A : A \leftarrow \mathsf{F}A$ is the *membership* relation for the datatype $\mathsf{F}A$. The two sections that follow explain the essential ideas without going into full details.

Inductive relations

Basically, a relation is inductive if one can use it to perform induction. Formally, $R : A \leftarrow A$ is inductive if

$$R \backslash X \subseteq X \quad \Rightarrow \quad \Pi \subseteq X$$

for all $X : A \leftarrow B$. At the point level, this definition says that if

$$(\forall c : cRa \Rightarrow cXb) \quad \Rightarrow \quad aXb$$

holds for all a and b, then X holds everywhere.

To take a simple example, let $<$ be the usual ordering on natural numbers. For fixed b, the implication

$$(\forall a : (\forall c : c < a \Rightarrow cXb) \Rightarrow aXb) \quad \Rightarrow \quad (\forall a : aXb)$$

asserts the general principle of mathematical induction for natural numbers, in which the role of an arbitrary proposition involving a is played by the expression aXb. As another example, take the relation $tail = outr \cdot cons^{\circ}$. The induction principle here is that if a relation holds for a list x whenever it holds for $tail\, x$, then it holds for every list.

A key result is that if S is inductive and $R \cdot R \subseteq R \cdot S$, then R is also inductive. This result is left as an instructive exercise in division. It follows that if S is inductive and $R \subseteq S$, then R is inductive. It also follows that S is inductive if and only if S^+ is, where S^+ is the transitive closure of S. This relation can be defined by $S^+ = (\mu X : S \cup (X \cdot S))$. The *reflexive* transitive closure S^* is the subject of a separate section given below.

There is another way to define the notion of an inductive relation, but it requires the allegory to be Boolean. A relation $R : A \leftarrow A$ is *well-founded* if

$$X \subseteq X \cdot R \quad \Rightarrow \quad X \subseteq \emptyset$$

for all $X : B \leftarrow A$. This corresponds to the set-theoretic notion that there are no infinite chains $a_0, a_1, \ldots$ such that $a_{i+1} R a_i$ for all $i \geq 0$. If a relation is inductive, then it is also well-founded, but the converse holds only in a Boolean allegory.

Membership

The other key idea is membership. Data types record the presence of elements, so one would expect a relator F to come equipped with a *membership* arrow $member_A : A \leftarrow \mathsf{F}A$ for each A, such that $a\; member\; x$ precisely when a is an element of x. In fact, not all relators do possess a membership relation, though fortunately those relators that arise in programming (the polynomial relators, the power relator, and the type relators) do. Here are the membership relations for the polynomial relators, in which we write *member* (F) to emphasise the dependence on F:

$$\begin{array}{rcl} \mathit{member}\,(\mathsf{id}) & = & \mathit{id} \\ \mathit{member}\,(\mathsf{K}_A) & = & \emptyset \\ \mathit{member}\,(\mathsf{F}+\mathsf{G}) & = & [\mathit{member}\,(\mathsf{F}), \mathit{member}\,(\mathsf{G})] \\ \mathit{member}\,(\mathsf{F}\times\mathsf{G}) & = & (\mathit{member}\,(\mathsf{F})\cdot \mathit{outl}) \cup (\mathit{member}\,(\mathsf{G})\cdot \mathit{outr}) \\ \mathit{member}\,(\mathsf{F}\cdot\mathsf{G}) & = & \mathit{member}\,(\mathsf{G})\cdot \mathit{member}\,(\mathsf{F}). \end{array}$$

Most of these are intuitively obvious, given the informal idea of membership. For example, in **Rel** the relator $\mathsf{F}A = A \times A$ returns pairs of elements and x is a member of a pair (y, z) if $x = y$ or $x = z$. On the other hand the constant relator $\mathsf{K}_A(B) = A$ records no elements from B, so its membership relation is the empty relation.

The membership relation for the power relator is $\in$, as one would expect. That leaves the membership relation for type relators. In a power allegory, the problem of defining the membership relation for a type functor T is the same problem as defining *setify* for the type. We have

$$\begin{array}{rcl} \mathit{member}\,(\mathsf{T}) & = & \in \cdot\, \mathit{setify}\,(\mathsf{T}) \\ \mathit{setify}\,(\mathsf{T}) & = & \Lambda \mathit{member}\,(\mathsf{T}). \end{array}$$

There is an alternative method (see Exercise 6.17) for defining the membership relation of type functors that does not depend on sets.

So far we have not said what it means for a relation to be a membership relation. One might expect that the formal definition would be straightforward, but in fact it is not and we will not discuss it in the text (but see Exercise 6.18). If F does have a membership relation *member*, then

$$R \cdot \mathit{member} \;\supseteq\; \mathit{member} \cdot \mathsf{F}R$$

for all R, so *member* is a lax natural transformation $\mathit{member} : \mathit{id} \hookleftarrow \mathsf{F}$. In fact, *member* – provided it exists – is the largest lax natural transformation with this type. It follows that membership relations, if they exist, are unique.

Consequences

The central result about the existence of inductive relations is that $\mathit{member}\,(\mathsf{F}) \cdot \alpha^\circ$ is inductive, where α is the initial F-algebra. For example, consider the initial type $([\mathit{zero}, \mathit{succ}], \mathit{Nat})$ of the functor $\mathsf{F}X = 1 + X$. The membership relation here is $[\emptyset, \mathit{id}]$, so we now know that

$$[\emptyset, \mathit{id}] \cdot [\mathit{zero}, \mathit{succ}]^\circ \;=\; \mathit{succ}^\circ$$

is inductive. Furthermore, $<$ is the relation $pred^+$, where $pred = succ^\circ$, so this relation is also inductive. This remark justifies the termination of the recursion in the digits of a number example.

As a second example, take lists. The membership relation is $[\emptyset, outr]$, so

$$[\emptyset, outr] \cdot [nil, cons]^\circ \quad = \quad outr \cdot cons^\circ$$

is inductive. Since $tail = outr \cdot cons^\circ$ we obtain that $tail^+$, the proper suffix relation is inductive. With snoc-lists, *init* and the proper prefix relation are both inductive.

The theorem referred to earlier about unique solutions is the following one.

Theorem 6.3 If $member\,(\mathsf{F}) \cdot S$ is inductive, then the equation $X = R \cdot \mathsf{F}X \cdot S$ has a unique solution $X = \phi(R, S)$. Moreover, $\phi(R, S)$ is entire if both R and S are entire.

Proof. For a full proof see (Doornbos and Backhouse 1995).

□

Corollary 6.2 Suppose $member\,(\mathsf{F}) \cdot g$ is inductive. Then the unique solution of $X = f \cdot \mathsf{F}X \cdot g$ is a function.

Proof. The unique solution is $X = (\![f]\!) \cdot (\![g^\circ]\!)^\circ$, which is entire by the theorem, since f and g are. But Exercise 6.10 shows that the solution is also simple, since f and g are.

□

For the next result, recall that R is surjective if $id \subseteq R \cdot R^\circ$. Thus, R is surjective if and only if R° is entire.

Corollary 6.3 If $member\,(\mathsf{F}) \cdot R^\circ$ is inductive, then $(\![R]\!)$ is surjective if R is.

Proof. $X = \alpha \cdot \mathsf{F}X \cdot R^\circ$ has the unique solution $X = (\![R]\!)^\circ$.

□

Using these results, we can now prove the theorem used in Section 5.6 to justify the definition of *partition* as a catamorphism.

Theorem 6.4 If R is surjective and $f \cdot R \subseteq \alpha \cdot \mathsf{F}f$, then $f^\circ = (\![R]\!)$.

Proof. In one direction we argue:

$$
\begin{array}{ll}
 & (\!|R|\!) \subseteq f^{\circ} \\
\equiv & \quad \{\text{shunting and } (\!|\alpha|\!) = id\} \\
 & f \cdot (\!|R|\!) \subseteq (\!|\alpha|\!) \\
\Leftarrow & \quad \{\text{fusion}\} \\
 & f \cdot R \subseteq \alpha \cdot \mathsf{F}f \\
\Leftarrow & \quad \{\text{assumption}\} \\
 & \mathit{true}.
\end{array}
$$

In the other direction we argue:

$$
\begin{array}{ll}
 & f^{\circ} \subseteq (\!|R|\!) \\
\Leftarrow & \quad \{\text{claim: } (\!|R|\!) \text{ is surjective}\} \\
 & (\!|R|\!) \cdot (\!|R|\!)^{\circ} \cdot f^{\circ} \subseteq (\!|R|\!) \\
\Leftarrow & \quad \{\text{since } f \cdot (\!|R|\!) \subseteq id \text{ from above; converse}\} \\
 & \mathit{true}.
\end{array}
$$

By Corollary 6.3 the claim follows by showing that $\mathit{member} \cdot R^{\circ}$ is inductive. But

$$
\begin{array}{ll}
 & \mathit{member} \cdot R^{\circ} \\
\subseteq & \quad \{\text{since } f \cdot R \subseteq \alpha \cdot \mathsf{F}f, \text{ shunting}\} \\
 & \mathit{member} \cdot \mathsf{F}f^{\circ} \cdot \alpha^{\circ} \cdot f \\
\subseteq & \quad \{\text{since } \mathit{member} : id \hookleftarrow \mathsf{F}\} \\
 & f^{\circ} \cdot \mathit{member} \cdot \alpha^{\circ} \cdot f.
\end{array}
$$

Now, by Exercise 6.16, $f^{\circ} \cdot \mathit{member} \cdot \alpha^{\circ} \cdot f$ is inductive because $\mathit{member} \cdot \alpha^{\circ}$ is. Finally, any relation included in an inductive relation is inductive, so $\mathit{member} \cdot R^{\circ}$ is inductive.

□

Exercises

6.12 Prove that R is inductive if and only if the equation $X = R\backslash X$ has a unique solution.

6.13 Prove that if S is inductive and $R \cdot R \subseteq R \cdot S$, then R is inductive.

6.14 Is the empty relation $\emptyset$ inductive? What about Π?

6.15 Show that the meet of two inductive relations is again inductive. Give a counter-example to show that the join of two inductive relations need not be inductive.

6.16 Show that if R is well-founded, then so is $f^{\circ} \cdot R \cdot f$ for any function f.

6.17 Define $inlist : A \leftarrow list^{+} A$ as a catamorphism. Why can't $inlist : A \leftarrow list\ A$ also be defined as a catamorphism? An arbitrary element of a list can be found by taking the first element of an arbitrary suffix, thus we can define $inlist = head \cdot tail^{*}$. Show how this definition can be generalised to define *intree*, where

$$tree\ A \quad ::= \quad tip\ A \mid bin\,(tree\ A, tree\ A).$$

How about

$$tree\ A \quad ::= \quad null \mid fork\,(tree\ A, A, tree\ A) \ \ ?$$

6.18 The formal definition of membership is this: a collection of arrows *member* is a membership relation of F if

$$\mathsf{F}R \cdot (member \backslash id) \quad = \quad member \backslash R$$

for all R. Show that F has a membership relation *member* if and only if $\mathsf{F}R \cdot (member \backslash S) = member \backslash (R \cdot S)$ for all R and S.

6.19 Assume that id is the largest lax natural transformation of type $id \hookleftarrow id$, and that relator F has a membership relation *member*. Show that *member* is the largest lax natural transformation of type $id \hookleftarrow \mathsf{F}$.

6.20 Prove that for any relators F and G, the relation $member\,(\mathsf{F}) \backslash member\,(\mathsf{G})$ is the largest lax natural transformation of type $\mathsf{F} \hookleftarrow \mathsf{G}$.

6.21 Prove that in a Boolean allegory $member\,(\mathsf{F})$ is entire if and only if $\mathsf{F}\emptyset = \emptyset$.

6.6 Sorting by selection

The problem of sorting is an interesting one because of the variety of approaches one can take. One can head for a catamorphism, the converse of a catamorphism, or various hylomorphisms using different intermediate datatypes. We will concentrate on just two sorting algorithms that depend on selection for their operation.

The function $sort : list\ A \leftarrow list\ A$ sorts a list under a given connected preorder $R : A \leftarrow A$. A relation R is said to be *connected* if $R \cup R^{\circ} = \Pi$; the normal terminology is *total* but this invites confusion with the quite different idea of an entire relation. The function *sort* is specified by

$$sort \ \subseteq \ ordered \cdot perm, \tag{6.6}$$

where *perm* was defined in the preceding chapter, and *ordered* is the coreflexive that tests whether a list is ordered under R.

If relation R is a *linear* order (that is, a connected anti-symmetric preorder), then $ordered \cdot perm$ is a function and (6.6) determines *sort* uniquely, but we assume only that R is a connected preorder, so the use of refinement is necessary. Strictly speaking, we should parameterise both *sort* and *ordered* with the relation R, but for this section it is simplest to assume that R is fixed.

We can define *ordered* as a relational catamorphism

$$ordered \quad = \quad (\![nil, cons \cdot ok]\!),$$

where the coreflexive *ok* is defined by the corresponding predicate

$$ok\,(a, x) \quad = \quad (\forall b : b\; inlist\; x : aRb).$$

The relation $inlist : A \leftarrow list\, A$ is the membership relation for lists. Thus *ordered* rebuilds its argument list, ensuring at each step that only smaller elements are added to the front. There is an alternative definition of *ok*, namely,

$$ok\,(a, x) \quad = \quad (x = [\,] \;\vee\; a\, R\,(head\; x)),$$

but this definition turns out not to be so useful for our purposes.

Selection sort

In outline, the derivation of selection sort is as follows:

$$\begin{array}{cl}
& ordered \cdot perm \\
= & \quad \{\text{since } perm = perm^\circ \text{ and } ordered = ordered^\circ\} \\
& (perm \cdot ordered)^\circ \\
= & \quad \{\text{since } ordered = (\![nil, cons \cdot ok]\!)\} \\
& (perm \cdot (\![nil, cons \cdot ok]\!))^\circ \\
\supseteq & \quad \{\text{fusion, for an appropriate relation } select\} \\
& (\![nil, select^\circ]\!)^\circ.
\end{array}$$

In selection sort we head for an algorithm expressed as the converse of a catamorphism. The proviso for the fusion step is

$$perm \cdot cons \cdot ok \quad \supseteq \quad select^\circ \cdot (id \times perm)$$

and the following calculation shows how *select* may be constructed:

$$
\begin{array}{cl}
 & perm \cdot cons \cdot ok \\
= & \quad \{\text{since } perm = ([nil, perm \cdot cons]) \text{ (Section 5.6)}\} \\
 & perm \cdot cons \cdot (id \times perm) \cdot ok \\
= & \quad \{\text{claim: } (id \times perm) \cdot ok = ok \cdot (id \times perm) \text{ (Exercise 6.22)}\} \\
 & perm \cdot cons \cdot ok \cdot (id \times perm) \\
\supseteq & \quad \{\text{specifying } select \subseteq ok \cdot cons^{\circ} \cdot perm\} \\
 & select^{\circ} \cdot (id \times perm).
\end{array}
$$

In words, *select* is defined by the rule that if $(a, y) = select\, x$, then $[a] +\!\!+\, y$ is a permutation of x with aRb for all elements b of y. The relation *select* is not a function because it is undefined on the empty list. But we do want it to be a function on non-empty lists. Suppose we can find *base* and *step* so that

$$([base, step]) \cdot embed \quad \subseteq \quad ok \cdot cons^{\circ} \cdot ([nil, perm \cdot cons]),$$

where $embed : list^{+} A \leftarrow list\, A$ converts a non-empty element of *list* A to an element of $list^{+} A$. Then we can take $select = ([base, step]) \cdot embed$.

The functions *base* and *step* are specified by the fusion conditions:

$$
\begin{array}{rcl}
base & \subseteq & ok \cdot cons^{\circ} \cdot perm \cdot wrap \\
step \cdot (id \times ok \cdot cons^{\circ}) & \subseteq & ok \cdot cons^{\circ} \cdot perm \cdot cons.
\end{array}
$$

These conditions are satisfied by taking

$$
\begin{array}{rcl}
base\, a & = & (a, [\,]) \\
step\,(a, (b, x)) & = & \begin{cases} (a, [b] +\!\!+\, x), & \text{if } aRb \\ (b, [a] +\!\!+\, x), & \text{otherwise.} \end{cases}
\end{array}
$$

We leave details as an exercise. Finally, appeal to the hylomorphism theorem gives that $X = ([nil, select^{\circ}])^{\circ}$ is the unique solution of the equation

$$X \quad = \quad (nil \cdot nil^{\circ}) \cup (cons \cdot (id \times X) \cdot select),$$

so we can implement *sort* by

$$
sort\, x \quad = \quad \begin{cases} [\,], & \text{if } x = [\,] \\ [a] +\!\!+\, sort\, y, & \text{otherwise} \\ \text{where } (a, y) = select\, x. & \end{cases}
$$

Quicksort

The so-called 'advanced' sorting algorithms (quicksort, mergesort, heapsort, and so on) all use some form of tree as an intermediate datatype. Here we sketch the development of Hoare's quicksort (Hoare 1962), which follows the path of selection sort quite closely.

Consider the type *tree A* defined by

$$tree\ A \quad ::= \quad null \mid fork\,(tree\ A, A, tree\ A).$$

The function *flatten* : *list A* $\leftarrow$ *tree A* is defined by

$$flatten \quad = \quad (\![nil, join]\!),$$

where $join\,(x, a, y) = x +\!\!+ [a] +\!\!+ y$. Thus *flatten* produces a list of the elements in a tree in left to right order.

In outline, the derivation of quicksort is

$$\begin{array}{cl}
 & ordered \cdot perm \\
\supseteq & \quad \{\text{since } flatten \text{ is a function}\} \\
 & ordered \cdot flatten \cdot flatten^\circ \cdot perm \\
= & \quad \{\text{claim: } ordered \cdot flatten = flatten \cdot inordered \text{ (see below)}\} \\
 & flatten \cdot inordered \cdot flatten^\circ \cdot perm \\
= & \quad \{\text{converses}\} \\
 & flatten \cdot (perm \cdot flatten \cdot inordered)^\circ \\
\supseteq & \quad \{\text{fusion, for an appropriate definition of } split\} \\
 & flatten \cdot (\![nil, split^\circ]\!)^\circ .
\end{array}$$

In quicksort we head for an algorithm expressed as a hylomorphism using trees as an intermediate datatype.

The coreflexive *inordered* on trees is defined by

$$inordered \quad = \quad (\![null, fork \cdot check]\!)$$

where the coreflexive *check* holds for (x, a, y) if

$$(\forall b : b\ intree\ x \Rightarrow bRa) \quad \wedge \quad (\forall b : b\ intree\ y \Rightarrow aRb).$$

The relation *intree* is the membership test for trees. Introducing $\mathsf{F}f = f \times id \times f$ for brevity, the proviso for the fusion step in the above calculation is

$$split^\circ \cdot \mathsf{F}(perm \cdot flatten) \quad \subseteq \quad perm \cdot flatten \cdot fork \cdot check.$$

To establish this condition we need the coreflexive *check'* that holds for (x, a, y) if

$$(\forall b : b \; inlist \; x \Rightarrow bRa) \quad \wedge \quad (\forall b : b \; inlist \; y \Rightarrow aRb).$$

Thus *check'* is similar to *check* except for the switch to lists.

We now reason:

$$\begin{array}{ll} & perm \cdot flatten \cdot fork \cdot check \\ = & \quad \{\text{catamorphisms, since } flatten = (\!|nil, join|\!)\} \\ & perm \cdot join \cdot \mathsf{F}\, flatten \cdot check \\ = & \quad \{\text{claim: } \mathsf{F}\, flatten \cdot check = check' \cdot \mathsf{F}\, flatten\} \\ & perm \cdot join \cdot check' \cdot \mathsf{F}\, flatten \\ = & \quad \{\text{claim: } perm \cdot join = perm \cdot join \cdot \mathsf{F}\, perm\} \\ & perm \cdot join \cdot \mathsf{F}\, perm \cdot check' \cdot \mathsf{F}\, flatten \\ = & \quad \{\text{claim: } \mathsf{F}\, perm \cdot check' = check' \cdot \mathsf{F}\, perm\text{; functors}\} \\ & perm \cdot join \cdot check' \cdot \mathsf{F}(perm \cdot flatten) \\ \supseteq & \quad \{\text{taking } split \subseteq check' \cdot join^{\circ} \cdot perm\} \\ & split^{\circ} \cdot \mathsf{F}(perm \cdot flatten). \end{array}$$

Formal proofs of the three claims are left as exercises. In words, *split* is defined by the rule that if $(y, a, z) = split\, x$, then $y +\!\!+ [a] +\!\!+ z$ is a permutation of x with bRa for all b in y and aRb for all b in z. As in the case of selection sort, we can implement *split* with a catamorphism on non-empty lists:

$$split \quad = \quad (\!|base, step|\!) \cdot embed.$$

The fusion conditions are:

$$\begin{array}{rcl} base & \subseteq & check' \cdot join^{\circ} \cdot perm \cdot wrap \\ split \cdot (id \times check' \cdot join) & \subseteq & check' \cdot join^{\circ} \cdot perm \cdot cons. \end{array}$$

These conditions are satisfied by taking

$$\begin{array}{rcl} base\; a & = & ([\,], a, [\,]) \\ step\,(a, (x, b, y)) & = & \begin{cases} ([a] +\!\!+ x, b, y), & \text{if } aRb \\ (x, b, [a] +\!\!+ y), & \text{otherwise.} \end{cases} \end{array}$$

Finally, appeal to the hylomorphism theorem gives that $X = flatten \cdot (\!|nil, split^{\circ}|\!)^{\circ}$ is the least solution of the equation

$$X \quad = \quad (nil \cdot nil^{\circ}) \cup (join \cdot (X \times id \times X) \cdot split).$$

Hence *sort* can be implemented by

$$sort\, x \quad = \quad \begin{cases} [\,], & \text{if } x = [\,] \\ sort\, y \mathbin{+\!\!+} [a] \mathbin{+\!\!+} sort\, z, & \text{otherwise} \\ \text{where } (y, a, z) = split\, x. & \end{cases}$$

The derivation of quicksort is thus very similar to that of selection sort except for the introduction of trees as an intermediate datatype.

Exercises

6.22 Using Exercise 6.19, show that $inlist = inlist \cdot perm$. Give a point-free definition of ok. Using the preceding exercise, prove that $(id \times perm) \cdot ok = ok \cdot (id \times perm)$.

6.23 Why is the recursion in the programs for selection sort and quicksort guaranteed to terminate?

6.24 Writing $ordered\, R$ to show explicitly the dependence on the preorder R, prove that $ordered\, R \cdot ordered\, S = ordered\, (R \cap S)$, stating any assumption you use.

6.25 Consider the problem of sorting a (finite) set. Why is the second of the specifications

$$\begin{aligned} sort &\subseteq ordered\, R \cdot setify^{\circ} \\ sort &\subseteq ordered\, (R \cap neq) \cdot setify^{\circ}, \end{aligned}$$

more sensible? The relation neq satisfies $a\, neq\, b$ if $a \neq b$. Develop the second specification to a version of selection sort, assuming that the input is presented as a list possibly with duplicates.

6.26 Sort using the type $tree\, A$ as in quicksort, but changing the definition of $flatten = (\![nil, join]\!)$ by taking $join\, (x, a, y) = [a] \mathbin{+\!\!+} x \mathbin{+\!\!+} y$.

6.27 Repeat the preceding exercise but with $join\, (x, a, y) = x \mathbin{+\!\!+} y \mathbin{+\!\!+} [a]$.

6.28 Repeat the preceding exercise but with $join\, (x, a, y) = [a] \mathbin{+\!\!+} merge\, (x, y)$, where $merge$ merges two ordered lists into one:

$$\begin{aligned} merge\, (x, [\,]) &= x \\ merge\, ([\,], y) &= y \\ merge\, ([a] \mathbin{+\!\!+} x, [b] \mathbin{+\!\!+} y) &= \begin{cases} [a] \mathbin{+\!\!+} merge\, (x, [b] \mathbin{+\!\!+} y), & \text{if } aRb \\ [b] \mathbin{+\!\!+} merge\, ([a] \mathbin{+\!\!+} x, y), & \text{otherwise.} \end{cases} \end{aligned}$$

6.29 What goes wrong if one attempts to sort using the intermediate datatype

$$tree\, A \quad ::= \quad null \mid tip\, A \mid fork\, (tree\, A, tree\, A) \quad ?$$

6.30 Recall from Section 5.6 that $perm = (\![nil, add]\!)$, where

$$add \quad = \quad cat \cdot (id \times cons) \cdot exch \cdot (id \times cat^{\circ}),$$

and $exch = assocr \cdot (swap \times id) \cdot assocl$. Using this characterisation of $perm$, we can reason:

$$\begin{aligned} & ordered \cdot perm \\ = \quad & \quad \{\text{using } perm = (\![nil, add]\!)\} \\ & ordered \cdot (\![nil, add]\!) \\ = \quad & \quad \{\text{fusion}\} \\ & (\![nil, ordered \cdot add]\!). \\ \supseteq \quad & \quad \{\text{for a suitable function } insert\} \\ & (\![nil, insert]\!). \end{aligned}$$

Verify the fusion condition $ordered \cdot add = ordered \cdot add \cdot (id \times ordered)$. Describe a function $insert$ satisfying $insert \cdot (id \times ordered) \subseteq ordered \cdot add$, and hence justify the last step. The resulting algorithm is known as 'insertion sort'.

6.7 Closure

A good illustration of the problem of how to compute the least fixed point of a recursion equation, when other fixed points may exist, is provided by relational closure. For every relation $R : A \leftarrow A$, there exists a smallest preorder R^* containing R, called the *reflexive transitive closure* of R. Our primary aim in this section is to show how to compute $\mathsf{E}(R^*) : \mathsf{P}A \leftarrow \mathsf{P}A$ whenever the result is known to be a finite set (so the computation will terminate). Many graph algorithms make use of such a computation, for instance in determining the set of vertices reachable from a given vertex.

The closure of R is characterised by the universal property

$$R \subseteq X \quad \equiv \quad R^* \subseteq X \quad \text{for all preorders } X.$$

It can also be defined explicitly by either of the equations

$$R^* \quad = \quad (\mu X : id \cup (X \cdot R)) \tag{6.7}$$

$$R^* \quad = \quad (\mu X : id \cup (R \cdot X)). \tag{6.8}$$

The proof that these definitions coincide is left as an exercise.

To justify (6.7) we have to prove that $S = (\mu X : id \cup (X \cdot R))$ is the smallest preorder containing R. Since

$$id \subseteq id \cup (S \cdot R) \subseteq S,$$

we have that S is reflexive. Using this, we obtain

$$R \subseteq id \cup R \subseteq id \cup (S \cdot R) \subseteq S,$$

and so S contains R. For transitivity, we argue:

$$\begin{array}{ll}
 & S \cdot S \subseteq S \\
\equiv & \quad \{\text{left-division}\} \\
 & S \subseteq S \backslash S \\
\Leftarrow & \quad \{\text{definition of } S\} \\
 & id \cup (S \backslash S) \cdot R \subseteq S \backslash S \\
\equiv & \quad \{\text{division}\} \\
 & S \cdot (id \cup (S \backslash S) \cdot R) \subseteq S \\
\equiv & \quad \{\text{composition over join}\} \\
 & S \cup (S \cdot (S \backslash S) \cdot R) \subseteq S \\
\Leftarrow & \quad \{\text{cancellation of division}\} \\
 & S \cup (S \cdot R) \subseteq S \\
\equiv & \quad \{\text{definition of } S\} \\
 & \mathit{true}.
\end{array}$$

Note the similarity of the proof to that of Theorem 6.2 with a switch of division operator. Finally, suppose X is a preorder that contains R. Then we have

$$id \cup (X \cdot R) \subseteq id \cup (X \cdot X) \subseteq X,$$

and so $S \subseteq X$.

Computing closure

It is a fact that the equation $X = id \cup (X \cdot R)$ has a unique solution, necessarily $X = R^*$, if and only if R is an inductive relation. In particular, *tail* is inductive, so *suffix* is characterised by the equation

$$\mathit{suffix} = id \cup (\mathit{suffix} \cdot \mathit{tail}).$$

Simple calculation leads to the following recursion equation for $\Lambda \mathit{suffix}$:

$$\Lambda \mathit{suffix} = \mathit{cup} \cdot \langle \tau, \Lambda(\mathit{suffix} \cdot \mathit{tail}) \rangle.$$

Using the fact that *tail* is not defined on the empty list, we can introduce a case analysis, obtaining

$$\begin{aligned}(\Lambda suffix)\,[\,] &= \{[\,]\}\\ (\Lambda suffix)\,([a] \mathbin{+\!\!+} x) &= \{[a] \mathbin{+\!\!+} x\} \,\cup\, (\Lambda suffix)\,x.\end{aligned}$$

Representing sets by lists in the usual way, we now obtain the following recursive program for computing the function *tails* of Section 5.6:

$$\begin{aligned}tails\,[\,] &= [[\,]]\\ tails\,([a] \mathbin{+\!\!+} x) &= [[a] \mathbin{+\!\!+} x] \mathbin{+\!\!+} tails\,x.\end{aligned}$$

All this is very straightforward, but the method only works because the basic recursion equation has a unique solution. In this section we show how to compute $\Lambda(R^*)$ when R is not an inductive relation.

Rather than attempt to derive a method for computing $\Lambda(R^*)$ directly, we concentrate first on giving an alternative recursive formulation for R^*. This recursion will be designed for efficient computation once we bring in the sets. The reason for this strategy is that it will enable us to employ relational reasoning for as long as possible.

In the following development use is made of relational subtraction. Recall from Exercise 4.30 that $R - S$ is defined by the universal property

$$R - S \;\subseteq\; T \;\equiv\; R \;\subseteq\; S \cup T.$$

From this we obtain a number of expected properties, including

$$\begin{aligned}R - \emptyset &= R\\ R \cup S &= R \cup (S - R)\\ R - (S \cup T) &= R - S - T\\ (R \cup S) - T &= (R - T) \cup (S - T).\end{aligned}$$

In the third identity the subtraction operator is assumed to associate to the left, so $R - S - T = (R - S) - T$. Use of these rules will be signalled just with the hint *subtraction.*

We will also make use of the following property of least fixed points, called the *rolling rule*:

$$(\mu X : \phi(\psi X)) \;=\; \phi(\mu X : \psi(\phi X)).$$

The proof of the rolling rule is left as an exercise, as are two other identities for manipulating least fixed points.

Finally, we make use of the fact that

$$R^* \cdot S \quad = \quad (\mu X : S \cup (R \cdot X))$$

This too is left as an exercise.

Now for the main calculation. The idea is to define θ by

$$\theta(P, Q) \quad = \quad P \cup (\mu X : Q \cup (R \cdot X - P)), \tag{6.9}$$

and use θ to obtain a recursive method for computing $R^* \cdot S$. From above we have $\theta(\emptyset, S) = R^* \cdot S$ so the aim is to show how to compute $\theta(\emptyset, S)$.

Since

$$\theta(P, \emptyset) = P \cup (\mu X : R \cdot X - P) = P \cup \emptyset = P,$$

it follows that $\theta(P, \emptyset) = P$. We can also obtain an expression for $\theta(P, Q)$, arguing:

$$\begin{array}{ll}
& \theta(P, Q) \\
= & \quad \{\text{definition}\} \\
& P \cup (\mu X : Q \cup (R \cdot X - P)) \\
= & \quad \{\text{subtraction}\} \\
& P \cup (\mu X : Q \cup (R \cdot X - P - Q)) \\
= & \quad \{\text{rolling with } \phi X = Q \cup X \text{ and } \psi X = (R \cdot X - P - Q)\} \\
& P \cup Q \cup (\mu X : R \cdot (Q \cup X) - P - Q) \\
= & \quad \{\text{subtraction}\} \\
& P \cup Q \cup (\mu X : (R \cdot Q - P - Q) \cup (R \cdot X - P - Q)) \\
= & \quad \{\text{definition of } \theta\} \\
& \theta(P \cup Q, Q \cup (R \cdot Q - P - Q)).
\end{array}$$

Summarising, we have shown that

$$\begin{array}{rcl}
\theta(\emptyset, S) & = & R^* \cdot S \\
\theta(P, \emptyset) & = & P \\
\theta(P, Q) & = & \theta(P \cup Q, R \cdot Q - P - Q)).
\end{array}$$

These three equations can be used in a recursive method for computing $R^* \cdot S$: compute $\theta(\emptyset, S)$, where

$$\theta(P, Q) \quad = \quad \begin{cases} P, & \text{if } Q = \emptyset \\ \theta(P \cup Q, R \cdot Q - P - Q), & \text{otherwise.} \end{cases}$$

The algorithm will not terminate unless, regarded as a set of pairs, $R^* \cdot S$ is a finite relation; under this restriction the algorithm will terminate because the size of P

increases at each recursive step. If it does terminate, then the three properties of θ given above guarantee that the result is $R^* \cdot S$.

The above algorithm is non-standard in that relations appear as data objects. But we can rewrite the algorithm to use sets as data rather than relations. Think of the relations P, Q and S as being elements (that is, having type $A \leftarrow 1$, where $R : A \leftarrow A$ is given) and let $p = \Lambda P$, $q = \Lambda Q$ and $s = \Lambda S$. Then p, q and s are the corresponding elements of type $\mathsf{P}A \leftarrow 1$. By applying Λ to everything in sight, and recalling that $\Lambda(R^* \cdot S) = \mathsf{E}(R^*) \cdot \Lambda S$, we obtain the following method for computing $\mathsf{E}(R^*)(s)$: compute *close* $(\emptyset, s)$, where

$$close\,(p,q) \quad = \quad \begin{cases} p, & \text{if } q = \emptyset \\ close\,(p \cup q, (\mathsf{E}R)\, q - p - q), & \text{otherwise.} \end{cases}$$

In this algorithm the operations are set-theoretic rather than relational; thus $\cup$ is set union and $(-)$ is set difference. As before, the algorithm is not guaranteed to terminate unless the closure of s under R is a finite set.

Exercises

6.31 Justify the alternative definition (6.8) of closure.

6.32 Show that $R \cdot S^* = (\mu X : R \cup (X \cdot S))$ and $S^* \cdot R = (\mu X : R \cup (S \cdot X))$.

6.33 Show that $R^* = (\![id, R]\!) \cdot (\![id, id]\!)^\circ$, where the intermediate datatype is

$$iterate\,A \quad ::= \quad once\,A \mid again\,(iterate\,A).$$

6.34 Give a catamorphism *chainR* on non-empty lists so that $R^* = head \cdot chainR^\circ$.

6.35 *The μ-calculus.* There are just two defining properties of $(\mu X : \phi X)$:

$$\begin{aligned} \phi(\mu X : \phi X) \quad &= \quad (\mu X : \phi X) \\ \phi Y \subseteq Y \quad &\Rightarrow \quad (\mu X : \phi X) \subseteq Y. \end{aligned}$$

The first one states that $(\mu X : \phi X)$ is a fixed point of ϕ, and the second one states that $(\mu X : \phi X)$ is a lower bound on all fixed points. Use these two rules to give a proof of the rolling rule

$$(\mu X : \phi(\psi X)) \quad = \quad \phi(\mu X : \psi(\phi X)).$$

The *diagonal rule* of the μ-calculus states that

$$(\mu X : \mu Y : \phi(X, Y)) \quad = \quad (\mu X : \phi(X, X)).$$

Prove the diagonal rule.

Finally, the *substitution rule* states that

$$(\mu X : \phi(\mu Y : \psi(X, Y))) \quad = \quad \phi(\mu X : \psi(\phi X, X))$$

The proof of the substitution rule is a simple combination of the preceding two rules. What is it?

6.36 Using the diagonal rule of the μ-calculus, show that

$$(R \cup S)^* \quad = \quad R^* \cdot (S \cdot R^*)^*$$

6.37 Using the preceding exercise, show that for any coreflexive C

$$R \cdot C = C \quad \Rightarrow \quad R^* = (R \cdot {\sim}C)^*,$$

where ${\sim}C$ is defined in Exercise 5.17.

Bibliographical remarks

The first account of the Knaster–Tarski fixed point theorem occurs in (Knaster 1928). That version only applied to powersets, and the generalisation to arbitrary complete lattices was published in (Tarski 1955). The use of this theorem (and of Kleene's celebrated result) are all-pervading in computing science. A much more in-depth account of the relevant theory can be found in (Davey and Priestley 1990).

The term *hylomorphism* was coined in (Meijer 1992). According to Meijer, the terminology is inspired by the Aristotelian philosophy that form and matter are one, $\upsilon\lambda o\sigma$ meaning 'dust' or 'matter'. The original proof that hylomorphisms can be characterised as least solutions of certain equations in the relational calculus is due to Backhouse and his colleagues (Aarts, Backhouse, Hoogendijk, Voermans, and Van der Woude 1992). In (Takano and Meijer 1995), hylomorphisms are used to explain some important optimisations in functional programming.

Results about unique solutions to recursion equations can be found in most introductory text books on set theory, e.g. (Enderton 1977). These expositions do not parameterise the recursion scheme by datatype constructors, because that requires a categorical view of datatypes. The systematic exploration in a categorical setting was initiated by (Mikkelsen 1976), and further elaborated in (Brook 1977). The account given here is based on (Doornbos and Backhouse 1995), which contains proofs of the quoted results, as well as various techniques for establishing that a relation is inductive. The calculus of least fixed points, partly developed in the exercises, has its roots in (De Bakker and De Roever 1973) in the context of relations. A more modern account can be found in (Mathematics of Program Construction Group 1995). The concept of membership appears to be original work by De Moor, in collaboration with Hoogendijk. An application appears in (Bird, Hoogendijk, and De Moor 1996), and a full account can be found in (Hoogendijk 1996).

The idea of using function converses in program specification and synthesis was originally suggested by (Dijkstra 1979), and has since been elaborated by various authors (Chen and Udding 1990; Harrison and Khoshnevisan 1992; Gries 1981). Our own interest in the topic was revived by reading (Augusteijn 1992; Schoenmakers 1992; Knapen 1993), and this led to the statement of Theorem 6.4. Indeed, this theorem seems to be at the heart of several of the references cited above.

The idea that algorithms can be classified through their synthesis is fundamental to this book, and it is a recurring theme in the literature on formal program development. Clark and Darlington first illustrated the idea by a classification of sorting algorithms (Darlington 1978; Clark and Darlington 1980), and the exposition given here is inspired by that pioneering work. An even more impressive classification of parsing algorithms was undertaken by (Partsch 1986); in (Bird and De Moor 1994) we have attempted to improve over a tiny portion of Partsch's results using the framework of this book.

Optimisation Problems

In the remaining four chapters we concentrate on a single class of problems; the aim is to develop a useful body of results for solving such problems efficiently. The problems are those that can be specified in the form

$$min\, R \cdot \Lambda((\![S]\!) \cdot (\![T]\!)^{\circ}).$$

This asks for a minimum element under the relation R in the set of results returned by a hylomorphism. Problems of this form will be referred to as *optimisation* problems.

Formalising a programming problem as one of optimisation is attractive because the specification is short, the idiom is widely applicable, and there are a number of well-known strategies for arriving at efficient solutions. We will study two such strategies in some depth: the *greedy* method, and *dynamic programming.*

The present chapter and Chapter 10 deal with greedy algorithms, while Chapters 8 and 9 are concerned with dynamic programming. This chapter and Chapter 8 consider a restricted class of optimisation problem in which T is the initial algebra of the intermediate datatype of the hylomorphism, so the problems take the form $min\, R \cdot \Lambda(\![S]\!)$. Chapters 9 and 10 deal with the general case.

The central result of this chapter is Theorem 7.2, which gives a simple condition under which an optimum result can be computed by computing an optimum partial result at each stage. The theoretical material is followed by three applications; each application ends with a functional program, written in Gofer, that solves the problem. The same format (specification, derivation, program) is followed for each optimisation problem that we solve in the remainder of the book.

We begin by defining the relation $min\, R$ formally and establishing its properties. Some proofs illustrate the interaction of division, membership and power-transpose, while others show the occasional need to bring in tabulations; many are left as instructive exercises in the relational calculus.

7.1 Minimum and maximum

For any relation $R : A \leftarrow A$ the relation $min\, R : A \leftarrow \mathsf{P}A$ is defined by

$$min\, R \;=\; \in \cap\, (R/\ni).$$

In words, a is a minimum element of x under R if a is both an element of x and a lower bound of x under R. The definition of $min\, R$ does not require that R be a preorder, but it is only really useful when such is the case. The definition of $min\, R$ can be phrased as the universal property

$$X \subseteq min\, R \;\equiv\; X \subseteq \in \text{ and } X \cdot \ni \subseteq R$$

for all $X : A \leftarrow A$. We can also define

$$max\, R \;=\; min\, R^{\circ},$$

so a maximum element under R is a minimum element under R°.

The following three properties of lower bounds are easy consequences of the fact that $(R/S) \cdot f = R/(f^{\circ} \cdot S)$:

$$(R/\ni) \cdot \tau \;=\; R \tag{7.1}$$
$$(R/\ni) \cdot \Lambda S \;=\; R/S^{\circ} \tag{7.2}$$
$$(R/\ni) \cdot union \;=\; (R/\ni)/\ni. \tag{7.3}$$

From (7.1) and (7.2) we obtain

$$min\, R \cdot \tau \;=\; id \cap R \tag{7.4}$$
$$min\, R \cdot \Lambda S \;=\; S \cap (R/S^{\circ}). \tag{7.5}$$

Equation (7.4) gives that R is reflexive if and only if the minimum element under R of a singleton set is its sole inhabitant. Equation (7.5) can be rephrased as the universal property

$$X \subseteq min\, R \cdot \Lambda S \;\equiv\; X \subseteq S \text{ and } X \cdot S^{\circ} \subseteq R.$$

This rule is used frequently and is indicated in calculations by the hint *universal property of min*.

Another useful rule is the following one:

$$min\, R \cdot \Lambda S \;=\; min\, (R \cap (S \cdot S^{\circ})) \cdot \Lambda S. \tag{7.6}$$

For the proof we argue as follows:

$$
\begin{array}{ll}
& min\,(R \cap (S \cdot S^\circ)) \cdot \Lambda S \\
= & \quad \{(7.5)\} \\
& S \;\cap\; ((R \cap (S \cdot S^\circ))/S^\circ) \\
= & \quad \{\text{division}\} \\
& S \;\cap\; (R/S^\circ) \;\cap\; ((S \cdot S^\circ)/S^\circ) \\
= & \quad \{\text{commutativity of meet, and } S \subseteq (S \cdot S^\circ)/S^\circ\} \\
& S \cap (R/S^\circ) \\
= & \quad \{(7.5)\} \\
& min\,R \cdot \Lambda S.
\end{array}
$$

Equation (7.6) allows us to bring in context into an optimisation problem. It states that for the purpose of taking a minimum under R on sets returned by ΛS, it is sufficient to constrain R to those values that are related to one and the same element by S. This context condition can be helpful in the task of checking the conditions we need to hold in order to solve an optimisation problem in a particular way. Below, we will refer to uses of (7.6) by the hint *context*.

Fusion with the power functor

Since $\mathsf{E}S = \Lambda(S \cdot \in)$, equation (7.5) leads to:

$$min\,R \cdot \mathsf{E}S \quad = \quad (S \cdot \in) \;\cap\; (R/(S \cdot \in)^\circ). \tag{7.7}$$

One application of (7.7) is the following result, which shows how to shunt a function through a minimum:

$$min\,R \cdot \mathsf{P}f \quad = \quad f \cdot min\,(f^\circ \cdot R \cdot f). \tag{7.8}$$

We reason:

$$
\begin{array}{ll}
& min\,R \cdot \mathsf{P}f \\
= & \quad \{(7.7) \text{ and } \mathsf{E} = \mathsf{P} \text{ on functions}\} \\
& (f \cdot \in) \;\cap\; (R/(f \cdot \in)^\circ) \\
= & \quad \{\text{converse; division; } f \text{ a function}\} \\
& (f \cdot \in) \;\cap\; ((R \cdot f)/\ni) \\
= & \quad \{\text{modular law, } f \text{ simple}\} \\
& f \cdot (\in \;\cap\; (f^\circ \cdot (R \cdot f)/\ni)) \\
= & \quad \{\text{division}\} \\
& f \cdot (\in \;\cap\; ((f^\circ \cdot R \cdot f)/\ni)
\end{array}
$$

$$\begin{array}{ll} = & \{\text{definition of } \mathit{min}\} \\ & f \cdot \mathit{min}\,(f^\circ \cdot R \cdot f). \end{array}$$

As an intermediate step in the above proof, we showed that

$$\mathit{min}\,R \cdot \mathsf{P}f \quad = \quad (f \cdot \in) \cap ((R \cdot f)/\ni).$$

This suggests the truth of

$$\mathit{min}\,R \cdot \mathsf{P}S \quad = \quad (S \cdot \in) \cap ((R \cdot S)/\ni). \tag{7.9}$$

Equation (7.9) does in fact hold provided that R is reflexive. In one direction the proof involves tabulations. The other half, namely,

$$\mathit{min}\,R \cdot \mathsf{P}S \quad \subseteq \quad (S \cdot \in) \cap ((R \cdot S)/\ni), \tag{7.10}$$

is easier and is all we will need later on. For the proof, observe that by the universal property of meet, (7.10) is equivalent to

$$\mathit{min}\,R \cdot \mathsf{P}S \subseteq S \cdot \in \quad \text{and} \quad \mathit{min}\,R \cdot \mathsf{P}S \cdot \ni \subseteq R \cdot S.$$

We argue in two lines, using the naturality of $\in$:

$$\begin{array}{lclcl} \mathit{min}\,R \cdot \mathsf{P}S & \subseteq & \in \cdot \mathsf{P}S & \subseteq & S \cdot \in \\ \mathit{min}\,R \cdot \mathsf{P}S \cdot \ni & \subseteq & \mathit{min}\,R \cdot \ni \cdot S & \subseteq & R \cdot S. \end{array}$$

Inclusion (7.10) is referred to subsequently as *fusion with the power functor.*

Distribution over union

Given a collection of non-empty sets, one can select a minimum of the union by selecting a minimum element in each collection and then taking a minimum of the set of minimums. Since a minimum of the empty set is not defined, the procedure breaks down if any set in the collection is empty, which is why we have inclusion rather than equality in:

$$\mathit{min}\,R \cdot \mathsf{P}(\mathit{min}\,R) \quad \subseteq \quad \mathit{min}\,R \cdot \mathit{union}. \tag{7.11}$$

Inclusion (7.11) only holds if R is a preorder. Under the same assumption we can strengthen (7.11) to read

$$\mathit{min}\,R \cdot \mathsf{P}(\mathit{min}\,R) \quad = \quad \mathit{min}\,R \cdot \mathit{union} \cdot \mathsf{P}(\mathit{dom}\,(\mathit{min}\,R)). \tag{7.12}$$

The proof of (7.11) is straightforward using (7.5) and the fact that $\in : \mathit{id} \hookleftarrow \mathsf{P}$. We leave the details as an exercise. The direction $\subseteq$ in (7.12) is also easy, using

$$\mathit{min}\,R \quad = \quad \mathit{min}\,R \cdot \mathit{dom}\,(\mathit{min}\,R).$$

Using fusion with the power functor, the other half, namely

$$min\,R \cdot union \cdot \mathsf{P}(dom\,(min\,R)) \;\subseteq\; min\,R \cdot \mathsf{P}(min\,R),$$

follows from the two inclusions

$$\begin{aligned} min\,R \cdot union \cdot \mathsf{P}(dom\,(min\,R)) &\subseteq min\,R \cdot \in \\ min\,R \cdot union \cdot \mathsf{P}(dom\,(min\,R)) \cdot \ni &\subseteq R \cdot min\,R. \end{aligned}$$

The proofs are left as exercises.

Implementing min

We cannot refine $min\,R$ to an implementable function except on non-empty finite sets; even then we require R to be a connected preorder. Given $setify : \mathsf{P}A \leftarrow list^+\,A$, the specification of $minlist\,R : A \leftarrow list^+\,A$ reads:

$$minlist\,R \;\subseteq\; min\,R \cdot setify.$$

Assuming R is connected, we can take $minlist\,R = (\!|id, bmin\,R|\!)$, where $bmin\,R$ (short for 'binary minimum') is defined by

$$bmin\,R\,(a, b) \;=\; (aRb \rightarrow a, b).$$

The function $minlist\,R$ chooses the leftmost minimum element in the case of ties. In the Appendix, *minlist* is defined as a function that takes as argument a Boolean function of type $Bool \leftarrow A \times A$.

Exercises

7.1 Prove that $(R/\ni) \cdot subset^\circ = R/\ni$, where $subset = \in\backslash\in$.

7.2 Prove that $subset \cdot \mathsf{E}R = \in\backslash(R \cdot \in)$.

7.3 Prove that $(R{\cdot}S)/T = (R/\ni){\cdot}((\ni{\cdot}S)/T)$ by rewriting R in the form $(\in \cdot \Lambda R^\circ)^\circ$.

7.4 Prove that $(R/\ni) \cdot \mathsf{P}S = (R \cdot S)/\ni$. (*Hint*: Exercises 7.1, 7.2, and 7.3 will be useful, as well as the fact that $Inc : \mathsf{P} \leftarrow \mathsf{E}$.)

7.5 Show that if R is a preorder, then $R \cdot (R/\ni) = R/\ni$.

7.6 Prove that $min\,(R \cap S) = (min\,R) \cap (min\,S)$.

7.7 Prove that $\in \cdot \ni \;=\; \Pi$. What well-known principle of set theory does this equation express? Using the result, prove that $min\,R = \in$ if and only if $R = \Pi$.

7.8 Prove that if R and S are reflexive, then $R \cap S^\circ = \mathit{min}\, R \cdot (\mathit{min}\, S)^\circ$. (*Hint*: for the direction $\subseteq$ use tabulations, letting (f, g) tabulate $R \cap S^\circ$ and $h = \Lambda(f \cup g)$.)

7.9 Using the preceding exercise prove that if R is reflexive, then $R = \mathit{min}\, R \cdot \ni$.

7.10 Suppose that R and S are reflexive. Prove that $\mathit{min}\, R \subseteq \mathit{min}\, S$ if and only if $R \subseteq S$.

7.11 Prove that $\mathit{min}\, R$ is a simple relation if and only if R is anti-symmetric.

7.12 Suppose that R is a preorder. Using Exercise 7.5, show that $\mathit{min}\, R = \in \cap (R \cdot \mathit{min}\, R)$.

7.13 Show that if R is a preorder and S is a function, then $R \cap (S^\circ \cdot S)$ is a preorder.

7.14 Prove that if R is a preorder, then $\mathit{max}\, R \cdot \Lambda R = R \cap R^\circ$.

7.15 Prove that $\langle \mathit{min}\, R \cdot S, \mathit{min}\, R \cdot T \rangle \subseteq \mathit{min}\,(R \times R) \cdot \Lambda \langle S, T \rangle$.

7.16 Prove that if R is reflexive and S is a preorder, then $\mathit{min}\, R \cdot \Lambda(\mathit{min}\, S) = \mathit{min}\,(S; R)$, where $S; R = S \cap (S^\circ \Rightarrow R)$.

7.17 The *supremum* operator can be defined in two ways:

$$\begin{aligned} \mathit{sup}\, R &= \mathit{min}\, R \cdot \Lambda(R^\circ / \in) \\ \mathit{sup}\, R &= ((\in \backslash R)/R)^\circ \cap (R/(\in \backslash R)). \end{aligned}$$

Prove that these two definitions are equivalent if R is a preorder.

7.18 One proof of the other half of (7.9) makes use of (7.4). Given (7.4) it suffices to show

$$(S \cdot \in) \cap ((R/\ni) \cdot \mathsf{P}S) \quad \subseteq \quad \mathit{min}\, R \cdot \mathsf{P}S.$$

The proof is a difficult exercise in tabulation.

7.19 The following few exercises, many of which originate from (Bleeker 1994), deal with *minimal* elements. Informally, a minimal element of a set x under a relation R is an element $a \in x$ such that for all $b \in x$ with bRa we have aRb. The formal definition is

$$\mathit{mnl}\, R \quad = \quad \mathit{min}\,(R^\circ \Rightarrow R).$$

Prove that $(R^\circ \Rightarrow R)$ is reflexive for any R, but that $(R^\circ \Rightarrow R)$ is not necessarily a preorder even when R is.

7.20 Prove that $\mathit{min}\, R \subseteq \mathit{mnl}\, R$ with equality only if R is a connected preorder.

7.21 Is it the case that $\mathit{mnl}\, R \subseteq \mathit{mnl}\, S$ if $R \subseteq S$?

7.22 Prove that $mnl\, R \cdot \mathsf{P}f = f \cdot mnl\,(f^\circ \cdot R \cdot f)$.

7.23 Prove that $mnl\, R = \in$ if and only if R is a symmetric relation.

7.24 Express $mnl\,(R + S)$ in terms of $mnl\, R$ and $mnl\, S$.

7.25 For an equivalence relation Q define *class* Q by

$$class\; Q \quad = \quad cap \cdot \langle id, \Lambda Q \cdot \in \rangle,$$

where *cap* returns the intersection of two sets. Informally, *class* Q takes a set and returns some equivalence class under Q. Prove that if R is a preorder, then

$$mnl\,(R; S) \quad = \quad mnl\, S \cdot class\,(R \cap R^\circ) \cdot \Lambda(mnl\, R).$$

7.26 The remaining exercises deal with the notion of a *well-bounded* preorder. In set-theoretic terms, a preorder R is well bounded if every non-empty set has a minimum under R; this translates to

$$dom\,(\in) \quad = \quad dom\,(min\, R).$$

Why is a well-bounded preorder necessarily a connected preorder?

As a difficult exercise in tabulations, show that R is well bounded if and only if $R \cap \neg R^\circ$, the *strict* part of R is a well-founded (equivalently, inductive) relation.

7.27 Prove that if R is well bounded, then so is $f^\circ \cdot R \cdot f$ for all functions f.

7.28 Show that R is well bounded if and only if $\in \subseteq R^\circ \cdot min\, R$.

7.29 Using the preceding exercise, show that if R is a well-bounded preorder, then

$$min\, R \cdot union \quad = \quad min\, R \cdot \mathsf{E}(min\, R).$$

This result strengthens (7.11). Using this fact, show how $min\, R \cdot setify$ can be expressed as a catamorphism.

7.30 A relation R is said to be *well supported* if

$$dom\,(\in) \quad = \quad dom\,(mnl\, R).$$

Show that well-supportedness is a weaker notion than well-boundedness.

7.31 Prove that if R is a well-supported preorder, then $\in \subseteq R^\circ \cdot mnl\, R$.

7.32 Prove that if R is a well-supported preorder, then $mnl\, R \cdot union = mnl\, R \cdot \mathsf{E}(mnl\, R)$.

7.2 Monotonic algebras

We come now to an important idea that will dominate the remaining chapters. By definition, an F-algebra $S : A \leftarrow \mathsf{F}A$ is *monotonic on* a relation $R : A \leftarrow A$ if

$$S \cdot \mathsf{F}R \quad \subseteq \quad R \cdot S.$$

To illustrate, consider the function $plus : Nat \leftarrow Nat \times Nat$. Addition of natural numbers is monotonic on leq, the normal linear ordering on numbers, a fact we can express as

$$plus \cdot (leq \times leq) \quad \subseteq \quad leq \cdot plus.$$

At the point level this reads

$$c = a + b \ \wedge \ a \leq a' \ \wedge \ b \leq b' \quad \Rightarrow \quad c \leq a' + b'.$$

When $S = f$, a function, monotonicity can be expressed in either of the following equivalent forms:

$$f \cdot \mathsf{F}R \cdot f^{\circ} \subseteq R \quad \text{and} \quad \mathsf{F}R \subseteq f^{\circ} \cdot R \cdot f.$$

By shunting we also obtain that f is monotonic on R if and only if it is monotonic on R°. However, none of these equivalences hold for general relations; in particular, it does not follow that if S is monotonic on R, then S is also monotonic on R°.

For functions, monotonicity is equivalent to distributivity. We say that $f : A \leftarrow \mathsf{F}A$ *distributes* over R if

$$f \cdot \mathsf{F}(min\, R) \quad \subseteq \quad min\, R \cdot \Lambda(f \cdot \mathsf{F}\in).$$

For example, the pointwise version of the fact that $+$ distributes over $\leq$ is

$$min\, x + min\, y \quad = \quad min\{a + b \mid a \in x \wedge b \in y\},$$

provided that x and y are non-empty. Here $min = min\, leq$.

Theorem 7.1 Function f is monotonic over R if and only if it distributes over R.

Proof. We argue:

$$\begin{aligned}
& f \cdot \mathsf{F}(min\, R) \subseteq min\, R \cdot \Lambda(f \cdot \mathsf{F}\in) \\
\equiv \quad & \{\text{universal property of } min\} \\
& f \cdot \mathsf{F}(min\, R) \subseteq f \cdot \mathsf{F}\in \ \text{ and } \ f \cdot \mathsf{F}(min\, R) \cdot (f \cdot \mathsf{F}\in)^{\circ} \subseteq R \\
\equiv \quad & \{\text{since } min\, R \subseteq \in\} \\
& f \cdot \mathsf{F}(min\, R) \cdot (f \cdot \mathsf{F}\in)^{\circ} \subseteq R
\end{aligned}$$

$$\begin{array}{ll} \equiv & \{\text{converse; relators}\} \\ & f \cdot \mathsf{F}(\mathit{min}\, R \cdot \ni) \cdot f^\circ \subseteq R \\ \equiv & \{\text{since } \mathit{min}\, R \cdot \ni = R \text{ if } R \text{ is reflexive}\} \\ & f \cdot \mathsf{F}R \cdot f^\circ \subseteq R. \end{array}$$

□

In this chapter the main result about monotonicity is the following, which we will refer to subsequently as the *greedy* theorem.

Theorem 7.2 If S is monotonic on a preorder R°, then

$$(\!|\mathit{min}\, R \cdot \Lambda S|\!) \quad \subseteq \quad \mathit{min}\, R \cdot \Lambda(\!|S|\!).$$

Proof. We reason:

$$\begin{array}{ll} & (\!|\mathit{min}\, R \cdot \Lambda S|\!) \subseteq \mathit{min}\, R \cdot \Lambda(\!|S|\!) \\ \equiv & \{\text{universal property of } \mathit{min}\} \\ & (\!|\mathit{min}\, R \cdot \Lambda S|\!) \subseteq (\!|S|\!) \quad \text{and} \quad (\!|\mathit{min}\, R \cdot \Lambda S|\!) \cdot (\!|S|\!)^\circ \subseteq R \\ \equiv & \{\text{since } \mathit{min}\, R \cdot \Lambda S \subseteq S\} \\ & (\!|\mathit{min}\, R \cdot \Lambda S|\!) \cdot (\!|S|\!)^\circ \subseteq R \\ \Leftarrow & \{\text{hylomorphism theorem (see below)}\} \\ & \mathit{min}\, R \cdot \Lambda S \cdot \mathsf{F}R \cdot S^\circ \subseteq R \\ \Leftarrow & \{\text{monotonicity: } \mathsf{F}R \cdot S^\circ \subseteq S^\circ \cdot R\} \\ & \mathit{min}\, R \cdot \Lambda S \cdot S^\circ \cdot R \subseteq R \\ \Leftarrow & \{\text{since } \mathit{min}\, R \cdot \Lambda S \subseteq R/S^\circ\text{; division}\} \\ & R \cdot R \subseteq R \\ \equiv & \{\text{transitivity of } R\} \\ & \mathit{true}. \end{array}$$

Recall that the hylomorphism theorem (Theorem 6.2) expressed a hylomorphism as a least fixed point of a certain recursion equation; thus by Knaster-Tarski, the hylomorphism $(\!|\mathit{min}\, R \cdot \Lambda S|\!) \cdot (\!|S|\!)^\circ$ is included in R if R satisfies the associated recursion inequation.

□

For an alternative formulation of the greedy theorem see Exercise 7.37. For problems involving *max* rather than *min*, the relevant condition of the greedy theorem is that S should be monotonic on R, not R°. Note also that we can always bring in context if we need to, and show that S is monotonic on $R^\circ \cap ((\!|S|\!) \cdot (\!|S|\!)^\circ)$.

The exercises given below explore some simple consequences of the greedy theorem. In the remainder of this chapter we will look at three other problems, each chosen to bring out a different aspect of the theory.

Exercises

7.33 Express the fact that $a+b \leq a+b'$ implies that $b \leq b'$ in a point-free manner.

7.34 Let α be the initial F-algebra. Prove that if α is monotonic on R, then R is reflexive.

7.35 Sometimes we want monotonicity and distributivity to hold only on the set of values returned by a relation. Find a suitably weakened definition of monotonicity that implies

$$f \cdot \mathsf{F}(\mathit{min}\, R \cdot \Lambda S) \quad \subseteq \quad \mathit{min}\, R \cdot \Lambda(f \cdot \mathsf{F}S).$$

7.36 Use the preceding exercise to give a *necessary*, as well as a sufficient, condition for establishing the conclusion of the greedy theorem.

7.37 Prove the following variation of the greedy theorem: if f is monotonic on R and $f \subseteq \mathit{min}\, R \cdot \Lambda S$, then $(\!|f|\!) \subseteq \mathit{min}\, R \cdot \Lambda(\!|S|\!)$.

7.38 Prove that if S is monotonic on R°, then $\mathit{min}\, R \cdot \Lambda S \cdot \mathit{min}\,(\mathsf{F}R) \subseteq \mathit{min}\, R \cdot \mathsf{E}S$.

7.39 The function *takewhile* of functional programming can be specified by

$$\mathit{takewhile}\, p \quad = \quad \mathit{max}\, R \cdot \Lambda(\mathit{list}\, p \cdot \mathit{prefix}),$$

where $R = \mathit{length}^\circ \cdot \mathit{leq} \cdot \mathit{length}$. In words, $\mathit{takewhile}\, p\, x$ returns the longest prefix of x with the property that all its elements satisfy p. (Question: why *the* longest here rather than a longest?) Using $\mathit{prefix} = (\!|\mathit{nil}, \mathit{cons} \cup \mathit{nil}|\!)$ and the greedy theorem, derive the standard implementation of *takewhile*.

7.40 The maximum segment sum problem (Gries 1984, 1990b) is specified by

$$\mathit{mss} \quad = \quad \mathit{max} \cdot \Lambda(\mathit{sum} \cdot \mathit{segment}),$$

where *max* is an abbreviation for *max leq*. Using $\mathit{segment} = \mathit{prefix} \cdot \mathit{suffix}$, express this problem in the form

$$\mathit{mss} \quad = \quad \mathit{max} \cdot \mathsf{P}(\mathit{max} \cdot \Lambda(\mathit{sum} \cdot \mathit{prefix})) \cdot \Lambda \mathit{suffix}.$$

Express *prefix* as a catamorphism on cons-lists, and use fusion to express $\mathit{sum} \cdot \mathit{prefix}$ as a catamorphism. Hence use the greedy theorem to show that

$$(\!|\mathit{zero}, \mathit{oplus}|\!) \quad \subseteq \quad \mathit{max} \cdot \Lambda(\mathit{sum} \cdot \mathit{prefix}),$$

where $oplus = max \cdot \Lambda(zero \cup plus)$. Finally, express $list\ (\![c, f]\!) \cdot tails$ as a catamorphism and hence show how to implement mss by a linear-time algorithm.

7.41 The function *filter* can be specified by $filter\ p = max\ R \cdot \Lambda(list\ p \cdot subseq)$. In words, $filter\ p\ x$ returns the longest subsequence of x with the property that all its elements satisfy p. (Question: again, why *the* rather than *a* longest subsequence?) Using $subseq = (\![nil, cons \cup outr]\!)$ and the greedy theorem, derive the standard program for *filter*.

7.42 Let L denote the normal lexical (i.e. dictionary) ordering on sequences. Justify the monotonicity condition

$$cons \cdot (id \times L) \quad \subseteq L \cdot cons.$$

Hence show that $(\![nil, max\ L \cdot \Lambda(cons \cup outr)]\!) \subseteq max\ L \cdot \Lambda subseq$.

Now justify the facts that: (i) a lexically largest subsequence of a given sequence is necessarily a *descending* sequence; and (ii) if x is descending and $a \geq head\ x$, then $[a] +\!\!+\ x$ is lexically larger than x. Use point-free versions of these facts to prove (formally!) that

$$(\![nil, (ok \rightarrow cons, outr)]\!) \quad = \quad (\![nil, max\ L \cdot \Lambda(cons \cup outr)]\!),$$

where ok holds for (a, x) if $x = [\,]$ or $a \geq head\ x$. Give an example to show $(ok \rightarrow cons, outr) \neq max\ L \cdot \Lambda(cons \cup outr)$.

7.3 Planning a company party

The following problem appears as an exercise in (Cormen, Leiserson, and Rivest 1990) in their chapter on dynamic programming:

> Professor McKenzie is consulting for the president of the A.-B. Corporation, which is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.
>
> **a.** Describe an algorithm to make up the guest list. The goal should be to maximise the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.
>
> **b.** How can the professor ensure that the president gets invited to his or her own party?

We will solve this problem with a greedy algorithm. The moral of the exercise is that our classification of what is a greedy algorithm can include problems that others might view as applications of dynamic programming.

The company structure is given by a tree of type *tree Employee*, where

$$\mathit{tree}\, A \quad ::= \quad \mathit{node}\,(A, \mathit{list}\,(\mathit{tree}\, A)).$$

The base functor is $\mathsf{F}(A, B) = A \times \mathit{list}\, B$. Given $\mathit{party} : \mathit{list}\, A \leftarrow \mathit{tree}\, A$, our problem is to compute $\mathit{max}\, R \cdot \Lambda \mathit{party}$, where

$$R \quad = \quad (\mathit{sum} \cdot \mathit{list}\, \mathit{rating})^{\circ} \cdot \mathit{leq} \cdot (\mathit{sum} \cdot \mathit{list}\, \mathit{rating}),$$

and $\mathit{rating} : \mathit{Real} \leftarrow \mathit{Employee}$ is the conviviality function for individual employees.

We can define $\mathit{party} : \mathit{list}\, A \leftarrow \mathit{tree}\, A$ in terms of a catamorphism that produces two parties, one that includes the root and one that excludes it:

$$\mathit{party} \quad = \quad \mathit{choose} \cdot (\!|\langle \mathit{include}, \mathit{exclude} \rangle|\!).$$

The relation *choose* is defined by $\mathit{choose} = \mathit{outl} \cup \mathit{outr}$. The relation *include* includes the root of the tree, so by the president's ruling the roots of the immediate subtrees have to be excluded. The relation *exclude* excludes the root, so we have an arbitrary choice between including or excluding the roots of the immediate subtrees. The formal definitions are:

$$\begin{array}{lcl} \mathit{include} & = & \mathit{cons} \cdot (\mathit{id} \times (\mathit{concat} \cdot \mathit{list}\, \mathit{outr})) \\ \mathit{exclude} & = & \mathit{outr} \cdot (\mathit{id} \times (\mathit{concat} \cdot \mathit{list}\, \mathit{choose})). \end{array}$$

Note that *include* is a function but *exclude* is not.

Derivation

The derivation involves two appeals to monotonicity, both of which we will justify afterwards.

We argue:

$$\begin{array}{cl} & \mathit{max}\, R \cdot \Lambda \mathit{party} \\ = & \quad \{\text{definition of } \mathit{party}\} \\ & \mathit{max}\, R \cdot \Lambda(\mathit{choose} \cdot (\!|\langle \mathit{include}, \mathit{exclude} \rangle|\!)) \\ = & \quad \{\text{since } \Lambda(X \cdot Y) = \mathsf{E}X \cdot \Lambda Y\} \\ & \mathit{max}\, R \cdot \mathsf{E}\mathit{choose} \cdot \Lambda(\!|\langle \mathit{include}, \mathit{exclude} \rangle|\!) \end{array}$$

$$
\begin{array}{cl}
\supseteq & \{\text{claim: Exercise 7.38 is applicable}\} \\
& \mathit{max}\ R \cdot \Lambda \mathit{choose} \cdot \mathit{max}\ (R \times R) \cdot \Lambda (\![\langle \mathit{include}, \mathit{exclude} \rangle]\!) \\
\supseteq & \{\text{claim: the greedy theorem is applicable}\} \\
& \mathit{max}\ R \cdot \Lambda \mathit{choose} \cdot (\![\mathit{max}\ (R \times R) \cdot \Lambda \langle \mathit{include}, \mathit{exclude} \rangle]\!).
\end{array}
$$

The first claim requires us to show that *choose* is monotonic on R, that is,

$$\mathit{choose} \cdot (R \times R) \quad \subseteq \quad R \cdot \mathit{choose}.$$

The proof is left as a simple exercise. The second claim requires us to show that $\langle \mathit{include}, \mathit{exclude} \rangle$ is monotonic on $R \times R$, that is,

$$\langle \mathit{include}, \mathit{exclude} \rangle \cdot (\mathit{id} \times \mathit{list}\ (R \times R)) \quad \subseteq \quad (R \times R) \cdot \langle \mathit{include}, \mathit{exclude} \rangle.$$

To justify this we argue:

$$
\begin{array}{cl}
& \langle \mathit{include}, \mathit{exclude} \rangle \cdot (\mathit{id} \times \mathit{list}\ (R \times R)) \\
= & \{\text{products}\} \\
& \langle \mathit{include} \cdot (\mathit{id} \times \mathit{list}\ (R \times R)), \mathit{exclude} \cdot (\mathit{id} \times \mathit{list}\ (R \times R)) \rangle \\
\subseteq & \{\text{claims}\} \\
& \langle R \cdot \mathit{include}, R \cdot \mathit{exclude} \rangle \\
= & \{\text{products}\} \\
& (R \times R) \cdot \langle \mathit{include}, \mathit{exclude} \rangle.
\end{array}
$$

We outline the proof of one subclaim and leave the other as an exercise. We argue:

$$
\begin{array}{cl}
& \mathit{include} \cdot (\mathit{id} \times \mathit{list}\ (R \times R)) \\
= & \{\text{definition of } \mathit{include}\text{; functors}\} \\
& \mathit{cons} \cdot (\mathit{id} \times \mathit{concat} \cdot \mathit{list}\ (\mathit{outr} \cdot (R \times R))) \\
\subseteq & \{\text{products; functors}\} \\
& \mathit{cons} \cdot (\mathit{id} \times \mathit{concat} \cdot \mathit{list}\ R \cdot \mathit{list}\ \mathit{outr}) \\
\subseteq & \{\text{claim: } \mathit{concat} \cdot \mathit{list}\ R \subseteq R \cdot \mathit{concat} \text{ (exercise)}\} \\
& \mathit{cons} \cdot (\mathit{id} \times R \cdot \mathit{concat} \cdot \mathit{list}\ \mathit{outr}) \\
\subseteq & \{\text{since } \mathit{cons} \text{ is monotonic on } R \text{ (exercise)}\} \\
& R \cdot \mathit{cons} \cdot (\mathit{id} \times \mathit{concat} \cdot \mathit{list}\ \mathit{outr}) \\
= & \{\text{definition of } \mathit{include}\} \\
& R \cdot \mathit{include}.
\end{array}
$$

It remains to refine $max\ (R \times R) \cdot \Lambda\langle include, exclude\rangle$ to a function. By Exercise 7.15 this expression is refined by

$$\langle max\ R \cdot \Lambda include, max\ R \cdot \Lambda exclude\rangle.$$

Since *include* is a function, the first term simplifies to *include*. We will leave it as a simple exercise to show that the second term refines to

$$concat \cdot list\,(max\ R \cdot \Lambda choose) \cdot outr.$$

In summary, we have derived (renaming *exclude*)

$$\begin{array}{rcl} party & = & max\ R \cdot \Lambda choose \cdot (\!|\langle include, exclude\rangle|\!) \\ include & = & cons \cdot (id \times (concat \cdot list\ outr)) \\ exclude & = & concat \cdot list\,(max\ R \cdot \Lambda choose) \cdot outr. \end{array}$$

The program

For efficiency, a list x is represented by the pair $(x, sum\,(list\ rating\ x))$. The relation $max\ R \cdot choose$ is refined to the standard function $bmax\ R$ that chooses the left-hand argument in the case of ties. All functions not defined in the following Gofer program appear in the list of standard functions given in the Appendix. (Actually, `bmax r` is a standard function, but is given here for clarity.) Employees are identified by their conviviality rating:

```
> party   = bmax r . treecata (pair (include, exclude))
> include = cons' . cross (id, concat' . list outr)
> exclude = concat' . list (bmax r) . outr

> cons'   = cross (cons, plus) . dupl
> concat' = cross (concat, sum) . unzip

> r      = leq . cross (outr, outr)
> bmax r = cond (r . swap) (outl, outr)

> data Tree = Node (Int, [Tree])
> treecata f (Node (a,ts)) = f (a, list (treecata f) ts)
```

Exercises

7.43 Supply the missing proofs in the derivation.

7.44 Answer the remaining questions in the problem, namely (i) what is the running time of the algorithm; and (ii) how can the professor ensure that the president gets invited to his or her own party?

7.4 Shortest paths on a cylinder

The following problem is taken from (Reingold, Nievergelt, and Deo 1977), but is rephrased slightly to avoid drawing a cylinder in LaTeX:

> Consider an $n \times m$ array of positive integers, rolled into a cylinder around a horizontal axis. For instance, the array
>
> $$\begin{array}{rrrrrrrrrrrr} & 11 & 53 & 34 & 73 & 18 & 53 & 99 & 52 & 31 & 54 & \\ & \mathbf{4} & 72 & 24 & 6 & 46 & 17 & 63 & 82 & \mathbf{89} & \mathbf{25} & \\ \longrightarrow & 67 & \mathbf{22} & 10 & 97 & 99 & \mathbf{64} & 33 & \mathbf{45} & 81 & 76 & \longrightarrow \\ & 24 & 71 & \mathbf{46} & \mathbf{62} & \mathbf{18} & 11 & \mathbf{54} & 40 & 17 & 51 & \\ & 99 & 8 & 57 & 76 & 7 & 51 & 90 & 92 & 51 & 21 & \end{array}$$
>
> is rolled into a cylinder by taking the top and bottom rows to be adjacent. A path is to be threaded from the entry side of the cylinder to the exit side, subject to the restriction that from a given square it is possible to go to only one of the three positions in the next column adjacent to the current position. The path may begin at any position on the entry side and may end at any position on the exit side. The cost of such a path is the sum of the integers in the squares through which it passes. Thus the cost of the sample path shown above (in boldface) is 429. Show how the dynamic programming approach to exhaustive search allows a path of least cost to be found in $O(n \times m)$ time.

Once again this exercise in dynamic programming is solvable by the methods given in this chapter, although it is Theorem 7.1 rather than the greedy theorem that is the crux. The other feature of interest is that the specification is motivated by paying due attention to types.

We will suppose that the input is represented as a non-empty cons-list of n-tuples, one tuple for each column of the array. Let F denote the base functor of non-empty cons-lists, so $\mathsf{F}(A, X) = A + (A \times X)$, and let L be a convenient abbreviation for the type functor $list^{+}$. Finally, let N denote the functor that sends A to the set of n-tuples over A. In the final program, n-tuples are represented by lists of length n.

Our problem is to compute $min\ R \cdot paths$, where $R = sum^{\circ} \cdot leq \cdot sum$ and $paths$ is a relation with type $paths : \mathsf{PL}\, Nat \leftarrow \mathsf{LN}\, Nat$. Because of the restriction on moves it is not possible to define $paths$ by the power transpose of a relational catamorphism, so that, strictly speaking, the problem does not fall within the class described at the outset of the chapter. Instead, we will define it in terms of a relation

$$generate : \mathsf{NPL}A \leftarrow \mathsf{F}(\mathsf{N}A, \mathsf{NPL}A).$$

In words, *generate* takes a new tuple and a tuple of sets of paths, and produces a tuple of sets of extended paths. Thus, the catamorphism $([generate])$ returns an n-tuple of sets of paths; the set associated with the kth component of the tuple is the set of valid paths that can start in component k of the first column. We can now define

$$paths \quad = \quad union \cdot setify \cdot ([generate]),$$

where $setify : \mathsf{P}A \leftarrow \mathsf{N}A$ converts an n-tuple into a set of its components.

Note that the type assigned to *generate* is parameterised by A; the restriction $A = Nat$ is required only for comparing paths under the sum ordering. Accordingly, *generate* will be a lax natural transformation. Recall from Section 5.7 that this means

$$\mathsf{NPL}R \cdot generate \quad \supseteq \quad generate \cdot \mathsf{F}(\mathsf{N}R, \mathsf{NPL}R)$$

for any relation R. To define *generate* we will need a number of other lax natural transformations of different types; what follows is an attempt to motivate their introduction.

First of all, it is clear that we have to take into account the restriction on moves in generating legal paths. The relation $moves : \mathsf{PN}A \leftarrow \mathsf{N}A$ is defined by

$$moves\, x \quad = \quad \{up\, x, x, down\, x\},$$

where *up* and *down* rotate columns:

$$\begin{aligned} up\,(a_1, a_2, \ldots, a_n) &\quad = \quad (a_n, a_1, \ldots, a_{n-1}) \\ down\,(a_1, a_2, \ldots, a_n) &\quad = \quad (a_2, a_3, \ldots, a_n, a_1). \end{aligned}$$

These functions are easily implemented when tuples are represented by lists. The relation $\mathsf{F}(id, moves)$ has type

$$\mathsf{F}(\mathsf{N}A, \mathsf{PNPL}A) \quad \leftarrow \quad \mathsf{F}(\mathsf{N}A, \mathsf{NPL}A),$$

and we will define $generate = S \cdot \mathsf{F}(id, moves)$ for an appropriate relation S.

The next step is to make use of a function $trans : \mathsf{NP}A \leftarrow \mathsf{PN}A$ that transposes a set of n-tuples. For example,

$$trans\{(a, b, c), (x, y, z)\} \quad = \quad (\{a, x\}, \{b, y\}, \{c, z\}).$$

In the final program, when sets and n-tuples are both represented by lists, *trans* will be implemented by a catamorphism of type $\mathsf{LL}A \leftarrow \mathsf{LL}A$.

The relation $\mathsf{F}(id, trans \cdot moves)$ has type

$$\mathsf{F}(\mathsf{N}A, \mathsf{NPPL}A) \quad \leftarrow \quad \mathsf{F}(\mathsf{N}A, \mathsf{NPL}A),$$

and so $\mathsf{F}(id, \mathsf{N}union \cdot trans \cdot moves)$ has type

$$\mathsf{F}(\mathsf{N}A, \mathsf{NPL}A) \quad \leftarrow \quad \mathsf{F}(\mathsf{N}A, \mathsf{NPL}A).$$

We now have $generate = S \cdot \mathsf{F}(id, \mathsf{N}union \cdot trans \cdot moves)$ for an appropriately chosen relation S.

The next step is to make use of a function $zip : \mathsf{NF}(A, B) \leftarrow \mathsf{F}(\mathsf{N}A, \mathsf{N}B)$ that commutes N with F. In the final program *zip* is replaced by the standard function on lists. The relation $zip \cdot \mathsf{F}(id, \mathsf{N}union \cdot trans \cdot moves)$ has type

$$\mathsf{NF}(A, \mathsf{PL}A) \quad \leftarrow \quad \mathsf{F}(\mathsf{N}A, \mathsf{NPL}A),$$

so now we have $generate = S \cdot zip \cdot \mathsf{F}(id, \mathsf{N}union \cdot trans \cdot moves)$ for an appropriate relation S.

The next step is to make use of the function $cp : \mathsf{PF}(A, B) \leftarrow \mathsf{F}(A, \mathsf{P}B)$, defined by $cp = \Lambda\mathsf{F}(id, \in)$. The relation $\mathsf{N}cp \cdot zip \cdot \mathsf{F}(id, \mathsf{N}union \cdot trans \cdot moves)$ has type

$$\mathsf{NPF}(A, \mathsf{L}A) \quad \leftarrow \quad \mathsf{F}(\mathsf{N}A, \mathsf{NPL}A),$$

so $generate = S \cdot \mathsf{N}cp \cdot zip \cdot \mathsf{F}(id, \mathsf{N}union \cdot trans \cdot moves)$ for some relation S.

Finally, we bring in $\alpha : \mathsf{L}A \leftarrow \mathsf{F}(A, \mathsf{L}A)$, the initial algebra of non-empty cons-lists. The relation $\mathsf{N}(\mathsf{P}\alpha \cdot cp) \cdot zip \cdot \mathsf{F}(id, \mathsf{N}union \cdot trans \cdot moves)$ has type

$$\mathsf{NPL}A \quad \leftarrow \quad \mathsf{F}(\mathsf{N}A, \mathsf{NPL}A),$$

and is the definition of *generate*.

The above typing information is summarised in the diagram

$$\begin{tikzcd}
\mathsf{NPL}A & \mathsf{F}(\mathsf{N}A, \mathsf{NPL}A) \arrow[l, "generate"'] \arrow[d, "{\mathsf{F}(id, \mathsf{N}union \cdot trans \cdot moves)}"] \\
\mathsf{NF}(A, \mathsf{PL}A) \arrow[u, "\mathsf{N}(\mathsf{P}\alpha \cdot cp)"] & \mathsf{F}(\mathsf{N}A, \mathsf{NPL}A) \arrow[l, "zip"]
\end{tikzcd}$$

We have motivated the definition of *generate* by following the arrows, but one can also work backwards.

Derivation

The derivation that follows relies heavily on the fact that all the above functions and relations are lax natural transformations of appropriate type. The monotonicity condition is that α is monotonic on R and is easy to verify. Since α is a function, Theorem 7.1 gives us that α distributes over R. Since $\Lambda(\alpha \cdot \mathsf{F}(id, \in)) = \mathsf{P}\alpha \cdot cp$, we therefore obtain the inclusion

$$\alpha \cdot \mathsf{F}(id, min\, R) \quad \subseteq \quad min\, R \cdot \mathsf{P}\alpha \cdot cp. \tag{7.13}$$

Armed with this fact, we calculate:

$$\begin{array}{cl}
& min\, R \cdot paths \\
= & \quad \{\text{definition of } paths\} \\
& min\, R \cdot union \cdot setify \cdot (\![generate]\!) \\
\supseteq & \quad \{\text{distribution over union (7.11), since } R \text{ is a preorder}\} \\
& min\, R \cdot \mathsf{P}(min\, R) \cdot setify \cdot (\![generate]\!) \\
\supseteq & \quad \{\text{naturality of } setify\} \\
& min\, R \cdot setify \cdot \mathsf{N}(min\, R) \cdot (\![generate]\!) \\
\supseteq & \quad \{\text{fusion (see below for definition of } Q)\} \\
& min\, R \cdot setify \cdot (\![Q]\!).
\end{array}$$

The condition for fusion is

$$\mathsf{N}(min\, R) \cdot generate \quad \supseteq \quad Q \cdot \mathsf{F}(id, \mathsf{N}(min\, R)),$$

and we can use this to derive a definition of Q:

$$\begin{array}{cl}
& \mathsf{N}(min\, R) \cdot generate \\
= & \quad \{\text{definition of } generate\} \\
& \mathsf{N}(min\, R \cdot \mathsf{P}\,\alpha \cdot cp) \cdot zip \cdot \mathsf{F}(id, \mathsf{N}\, union \cdot trans \cdot moves) \\
\supseteq & \quad \{(7.13);\ \text{functors}\} \\
& \mathsf{N}\alpha \cdot \mathsf{NF}(id, min\, R)) \cdot zip \cdot \mathsf{F}(id, \mathsf{N}\, union \cdot trans \cdot moves) \\
\supseteq & \quad \{\text{naturality of } zip\} \\
& \mathsf{N}\alpha \cdot zip \cdot \mathsf{F}(\mathsf{N}id, \mathsf{N}(min\, R)) \cdot \mathsf{F}(id, \mathsf{N}\, union \cdot trans \cdot moves) \\
= & \quad \{\text{functors}\} \\
& \mathsf{N}\alpha \cdot zip \cdot \mathsf{F}(id, \mathsf{N}(min\, R \cdot union) \cdot trans \cdot moves) \\
\supseteq & \quad \{\text{distribution over union (7.11)}\} \\
& \mathsf{N}\alpha \cdot zip \cdot \mathsf{F}(id, \mathsf{N}(min\, R \cdot \mathsf{P}(min\, R)) \cdot trans \cdot moves) \\
= & \quad \{\text{functors}\} \\
& \mathsf{N}\alpha \cdot zip \cdot \mathsf{F}(id, \mathsf{N}(min\, R) \cdot \mathsf{NP}(min\, R) \cdot trans \cdot moves)
\end{array}$$

$$
\begin{aligned}
&\supseteq \quad \{\text{naturality of } \mathit{trans}\}\\
&\quad \mathsf{N}\alpha \cdot \mathit{zip} \cdot \mathsf{F}(\mathit{id}, \mathsf{N}(\mathit{min}\, R) \cdot \mathit{trans} \cdot \mathsf{PN}(\mathit{min}\, R) \cdot \mathit{moves})\\
&\supseteq \quad \{\text{naturality of } \mathit{moves}\}\\
&\quad \mathsf{N}\alpha \cdot \mathit{zip} \cdot \mathsf{F}(\mathit{id}, \mathsf{N}(\mathit{min}\, R) \cdot \mathit{trans} \cdot \mathit{moves} \cdot \mathsf{N}(\mathit{min}\, R))\\
&= \quad \{\text{functors, introducing } Q\}\\
&\quad Q \cdot \mathsf{F}(\mathit{id}, \mathsf{N}(\mathit{min}\, R)),
\end{aligned}
$$

where $Q = \mathsf{N}\alpha \cdot \mathit{zip} \cdot \mathsf{F}(\mathit{id}, \mathsf{N}(\mathit{min}\, R) \cdot \mathit{trans} \cdot \mathit{moves})$.

The definition of Q can be simplified. When F is the base functor of non-empty cons-lists, *zip* is a coproduct $\mathit{zip} = \mathit{id} + \mathit{zip}'$, where $\mathit{zip}' : \mathsf{N}(A \times B) \leftarrow \mathsf{N}A \times \mathsf{N}B$, so we can write Q as a coproduct

$$Q \quad = \quad [\mathsf{N}\mathit{wrap}, \mathsf{N}\mathit{cons} \cdot \mathit{zip}' \cdot (\mathit{id} \times \mathsf{N}(\mathit{min}\, R) \cdot \mathit{trans} \cdot \mathit{moves})].$$

With this definition of Q, the solution is $\mathit{min}\, R \cdot \mathit{setify} \cdot (\![Q]\!)$.

The program

In the following Gofer program we replace both N and P by `list`, thereby representing both tuples and sets by (non-empty) lists. The function zip' is then implemented by the standard function `zip`. For efficiency, a path x is represented by the pair $(x, \mathit{sum}\, x)$. The relation $\mathit{min}\, R \cdot \mathit{setify}$ is implemented by the standard function `minlist r`, whose definition is given in the Appendix. The function `catallist` implements catamorphisms on non-empty cons-lists; its definition is also given in the Appendix.

With that, the Gofer program is:

```
> path = minlist r . catallist (list wrap', list cons' . step)
> step = zip . cross (id, list (minlist r) . trans . moves
> r    = leq . cross (outr, outr)

> wrap' = pair (wrap, id)
> cons' = cross (cons, plus) . dupl

> moves x = [up x, x, down x]
> up x    = tail x ++ [head x]
> down x  = [last x] ++ init x
```

Exercises

7.45 Did we use the fact that N was a relator in the derivation?

7.46 What change is needed to deal with a similar problem in which

$$moves\ x \quad = \quad \{up\,(up\ x), up\ x, x, down\ x, down\,(down\ x)\}?$$

7.47 What if we took $moves = \tau$?

7.5 The security van problem

Our final problem illustrates an important idea in the theory of greedy algorithms: when the desired monotonicity condition is not met, it may nevertheless still be possible to arrive at a greedy solution by refining the ordering.

The following problem, invented by Hans Zantema, is typical of the sort that can be specified using the idea of partitioning a list:

> Suppose a bank has a known sequence of deposits and withdrawals. For security reasons the total amount of cash in the bank should never exceed some fixed amount N, assumed to be at least as large as any single transaction. To cope with demand and supply, a security van can be called upon to deliver funds to the bank or to take away a surplus. The problem is to compute a schedule under which the van visits the bank a minimum number of times.

Let us call a sequence $[a_1, a_2, \ldots, a_n]$ of transactions *secure* if there is an amount r, indicating the bank's reserves at the beginning of the sequence of transactions, such that each of the sums

$$r, \quad r + a_1, \quad r + a_1 + a_2, \quad \ldots, r + a_1 + \cdots + a_n$$

lies between zero and N. For example, taking $N = 10$, the sequence $[2, -5, 7]$ is secure because the van can take away or deliver enough cash to ensure an initial reserve of between three and six units. Given the constraint that N is no smaller than any single transaction, every singleton sequence is secure, so a valid schedule certainly exists.

To formalise the constraint, define

$$\begin{aligned} ceiling &= max\ leq \cdot \Lambda(sum \cdot prefix) \\ floor &= min\ leq \cdot \Lambda(sum \cdot prefix), \end{aligned}$$

where $sum : Nat \leftarrow list\ Nat$ sums a list of numbers and $prefix$ is the prefix relation on non-empty lists. Then a sequence x of transactions is secure if there is an $r \geq 0$ such that

$$0 \leq r + floor\, x \leq N \quad \text{and} \quad 0 \leq r + ceiling\, x \leq N.$$

We leave it as a short exercise to show that this condition can be phrased in the equivalent form

$$bmax\,(ceiling\, x, ceiling\, x - floor\, x) \leq N.$$

Let *secure* be the coreflexive corresponding to this predicate. It is a simple consequence of the definition that if *secure* holds for a sequence x, then it also holds for an arbitrary prefix of x; in symbols,

$$prefix \cdot secure \quad \subseteq \quad secure \cdot prefix.$$

A coreflexive satisfying this property is called *prefix-closed*. For most of the derivation prefix-closure is the only property of *secure* that we will need. At the end, and only to obtain an efficient implementation of the greedy algorithm, we will use the less obvious fact that *secure* is also *suffix-closed*: if x is secure, then any suffix of x is secure.

Our problem can now be expressed as one of computing

$$min\, R \cdot \Lambda(list\ secure \cdot partition),$$

where $R = length^{\circ} \cdot leq \cdot length$ and $partition : list\,(list^{+}\, A) \leftarrow list\, A$ is the combinatorial relation discussed in Section 5.6.

Recall that one expression for *partition* is

$$partition \quad = \quad (\!|nil, new \cup glue|\!),$$

where

$$\begin{aligned} new &= cons \cdot (wrap \times id) \\ glue &= cons \cdot (cons \times id) \cdot assocl \cdot (id \times cons^{\circ}). \end{aligned}$$

Appeal to fusion (left as an exercise) shows that

$$list\ secure \cdot partition \quad = \quad (\!|nil, new \cup old|\!),$$

where

$$old \quad = \quad cons \cdot ((secure \cdot cons) \times id) \cdot assocl \cdot (id \times cons^{\circ}),$$

so the task is to compute $min\, R \cdot \Lambda(\!|nil, new \cup old|\!)$ efficiently.

Derivation

A greedy algorithm exists if $[nil, new \cup old]$ is monotonic on $R°$. The monotonicity condition is equivalent to two conditions:

$$\begin{array}{rcll} new \cdot (id \times R°) & \subseteq & R° \cdot (new \cup old) & (7.14) \\ old \cdot (id \times R°) & \subseteq & R° \cdot (new \cup old). & (7.15) \end{array}$$

Well, (7.14) is true but (7.15) is false.

To prove (7.14) we reason:

$$\begin{array}{ll} & new \cdot (id \times R°) \\ = & \quad \{\text{definition of } new\} \\ & cons \cdot (wrap \times R°) \\ \subseteq & \quad \{\text{since } cons \text{ is monotonic on } R° \text{ (exercise)}\} \\ & R° \cdot cons \cdot (wrap \times id) \\ = & \quad \{\text{definition of } new\} \\ & R° \cdot new \\ \subseteq & \quad \{\text{monotonicity of join}\} \\ & R° \cdot (new \cup old). \end{array}$$

To see why (7.15) is false, let $[x] +\!\!+\, xs$ and $[y] +\!\!+\, ys$ be two equal-length partitions of the same sequence, so, certainly,

$$([x] +\!\!+\, xs)\; R°\; ([y] +\!\!+\, ys).$$

Suppose also that $[a] +\!\!+\, x$ is secure. Then (7.15) states that one or other of the following two possibilities must hold:

$$\begin{array}{ll} \text{(i)} & ([[a] +\!\!+\, x] +\!\!+\, xs)\; R°\; ([[a]] +\!\!+\, [y] +\!\!+\, ys) \\ \text{(ii)} & ([[a] +\!\!+\, x] +\!\!+\, xs)\; R°\; ([[a] +\!\!+\, y] +\!\!+\, ys) \quad \text{and} \quad secure\,([a] +\!\!+\, y). \end{array}$$

Since $[x] +\!\!+\, xs$ and $[y] +\!\!+\, ys$ have equal length, the first possibility fails, and the second reduces to $secure\,([a] +\!\!+\, y)$. But, in general, there is no reason why $secure\,([a] +\!\!+\, x)$ should imply $secure\,([a] +\!\!+\, y)$.

However, the analysis given above does suggest a way out: if y is a prefix of x, then $secure\,([a] +\!\!+\, x)$ does imply $secure\,([a] +\!\!+\, y)$ because *secure* is prefix-closed. Suppose we refine the order R to $R\,;H$, where

$$H \;=\; (head° \cdot prefix \cdot head) \;\cup\; (nil \cdot nil°).$$

Recall from Chapter 4 that

$$R\,;H \quad = \quad R \cap (R^\circ \Rightarrow H).$$

In words, $[\,]\,(R\,;H)\,[\,]$ and $([y] + ys)\,(R\,;H)\,([x] + xs)$ if ys is strictly shorter than xs, or it has the same length and $y\ \mathit{prefix}\ x$. Since $R\,;H \subseteq R$ we can still obtain a greedy algorithm for our problem if we can show that S is monotonic on $(R\,;H)^\circ$ and that $([\mathit{min}\,(R\,;H) \cdot \Lambda S])$ can be refined to a function. The second task is easy since *old* returns a shorter result than *new* if it returns any result at all; in symbols, $\mathit{old} \subseteq (R\,;H) \cdot \mathit{new}$. Hence we obtain

$$([\mathit{wrap} \cdot \mathit{wrap}, (\mathit{ok} \rightarrow \mathit{glue}, \mathit{new})]) \quad \subseteq \quad ([\mathit{min}\,(R\,;H) \cdot \Lambda S]),$$

where the coreflexive *ok* holds on (a, xs) if $xs \neq [\,]$ and $[a] + \mathit{head}\ xs$ is secure.

It remains to show that S is monotonic on $(R\,;H)^\circ$, that is,

$$\mathit{new} \cdot (\mathit{id} \times (R\,;H)^\circ) \quad \subseteq \quad (R\,;H)^\circ \cdot (\mathit{new} \cup \mathit{old}) \qquad (7.16)$$

$$\mathit{old} \cdot (\mathit{id} \times (R\,;H)^\circ) \quad \subseteq \quad (R\,;H)^\circ \cdot (\mathit{new} \cup \mathit{old}). \qquad (7.17)$$

Condition (7.16) follows from the fact that

$$\mathit{new} \cdot (\mathit{id} \times \Pi) \quad \subseteq \quad H^\circ \cdot \mathit{new}. \qquad (7.18)$$

A formal proof of (7.18) is left as an exercise. Using it, we can argue:

$$\begin{array}{ll}
 & \mathit{new} \cdot (\mathit{id} \times (R\,;H)^\circ) \\
\subseteq & \quad \{\text{since } R\,;H \subseteq R\} \\
 & \mathit{new} \cdot (\mathit{id} \times R^\circ) \\
\subseteq & \quad \{\text{inclusions (7.14) and (7.18)}\} \\
 & (R^\circ \cdot \mathit{new}) \cap (H^\circ \cdot \mathit{new}) \\
= & \quad \{\text{since } \mathit{new} \text{ is a function}\} \\
 & (R^\circ \cap H^\circ) \cdot \mathit{new} \\
\subseteq & \quad \{\text{since } X \cap Y \subseteq (X\,;Y)\text{, and converses}\} \\
 & (R\,;H)^\circ \cdot \mathit{new}.
\end{array}$$

Condition (7.17) follows from three subsidiary claims, in which $|R|$, the *strict* part of R, is defined by $|R| = R \cap \neg R^\circ$:

$$\mathit{old} \cdot (\mathit{id} \times \Pi) \quad \subseteq \quad H^\circ \cdot \mathit{new} \qquad (7.19)$$

$$\mathit{old} \cdot (\mathit{id} \times |R|^\circ) \quad \subseteq \quad R^\circ \cdot \mathit{new} \qquad (7.20)$$

$$\mathit{old} \cdot (\mathit{id} \times (R^\circ \cap H^\circ)) \quad \subseteq \quad (R^\circ \cap H^\circ) \cdot \mathit{old}. \qquad (7.21)$$

Again, we leave proofs as exercises. Now we argue:

$$\begin{array}{ll} & \mathit{old} \cdot (\mathit{id} \times (R \,;\, H)^\circ) \\ = & \quad \{\text{since } X \,;\, Y = |X| \cup (X \cap Y) \text{ and } |X|^\circ = |X^\circ|\} \\ & \mathit{old} \cdot (\mathit{id} \times (|R^\circ| \cup (R^\circ \cap H^\circ))) \\ = & \quad \{\text{distributing join}\} \\ & (\mathit{old} \cdot (\mathit{id} \times |R^\circ|)) \ \cup \ (\mathit{old} \cdot (\mathit{id} \times (R^\circ \cap H^\circ))) \\ \subseteq & \quad \{\text{conditions (7.19), (7.20) and (7.21)}\} \\ & ((R^\circ \cap H^\circ) \cdot \mathit{new}) \ \cup \ ((R^\circ \cap H^\circ) \cdot \mathit{old}) \\ \subseteq & \quad \{\text{since } X \cap Y \subseteq X \,;\, Y, \text{ and converses}\} \\ & (R \,;\, H)^\circ \cdot (\mathit{new} \cup \mathit{old}). \end{array}$$

The greedy condition is established.

Up to this point we have used no property of *secure* other than the fact that it is prefix-closed. For the final program we need to implement the security test efficiently. To do this, recall from Section 5.6 that the prefix relation $\mathit{prefix} : \mathit{list}\, A \leftarrow \mathit{list}\, A$ can be defined as a catamorphism

$$\mathit{prefix} \quad = \quad (\![\mathit{nil}, \mathit{cons} \cup \mathit{nil}]\!).$$

Since $\mathit{sum} \cdot \mathit{prefix} = (\![\mathit{zero}, \mathit{plus} \cup \mathit{zero}]\!)$, two baby-sized applications of the greedy theorem yield:

$$\begin{array}{rcl} \mathit{ceiling} & = & (\![\mathit{zero}, \mathit{omax} \cdot \mathit{plus}]\!) \\ \mathit{floor} & = & (\![\mathit{zero}, \mathit{omin} \cdot \mathit{plus}]\!), \end{array}$$

where $\mathit{omax}\, a = \mathit{bmax}\,(a, 0)$ and $\mathit{omin}\, a = \mathit{bmin}\,(a, 0)$. Recall that x is secure if

$$\mathit{bmax}\,(\mathit{ceiling}\, x, \mathit{ceiling}\, x - \mathit{floor}\, x) \quad \leq \quad N.$$

Since $\mathit{bmax}\,(b, b - c) = b - \mathit{omin}\, c$, we obtain that $[a] +\!\!+\, x$ is secure if

$$\mathit{omax}\,(a + b) - \mathit{omin}\,(a + c) \quad \leq \quad N,$$

where $b = \mathit{ceiling}\, x$ and $c = \mathit{floor}\, x$. This condition implies $\mathit{omax}\, b - \mathit{omin}\, c \leq N$, so x is secure. This proves that *secure* is suffix-closed.

In summary, we have derived the following program for computing a valid schedule, in which *schedule* is parameterised by N and *ok* is expressed as a predicate rather than a coreflexive:

$$\begin{array}{rcl} \mathit{schedule}\, N & = & (\![\mathit{nil}, (\mathit{ok}\, N \to \mathit{glue}, \mathit{new})]\!) \\ \mathit{ok}\, N\,(a, [\,]) & = & \mathit{false} \\ \mathit{ok}\, N\,(a, [x] +\!\!+\, \mathit{xs}) & = & \mathit{omax}\,(a + \mathit{ceiling}\, x) - \mathit{omin}\,(a + \mathit{floor}\, x) \leq N. \end{array}$$

The program

In the final Gofer program we represent the empty partition by $([\,],(0,0))$ and a partition $[x] \mathbin{+\!\!+} xs$ by a pair

$$([x] \mathbin{+\!\!+} xs, (\mathit{ceiling}\, x, \mathit{floor}\, x)).$$

The standard function `cond p (f,g)` implements $(p \to f, g)$, and `catalist` is the standard catamorphism former for cons-lists. The function `split` implements cons°.

```
> schedule n = catalist (start, cond (ok n) (glue', new'))

> ok n   = cond empty (false, (<= n) . minus . outr . glue')
>          where empty = null . outl . outr
> start = ([], (0,0))
> glue' = cross (glue, augment) . dupl
>          where augment = cross (omax . plus, omin . plus) . dupl
> new'   = cross (new, augment) . dupl
>          where augment = pair (omax, omin) . outl
> glue   = cons . cross (cons, id) . assocl . cross (id, split)
> new    = cons . cross (wrap, id)

> omax = cond (>= 0) (id, zero)
> omin = cond (<= 0) (id, zero)
```

Exercises

7.48 Prove that $0 \leq r + \mathit{floor}\, x \leq N$ and $0 \leq r + \mathit{ceiling}\, x \leq N$ for some $r \geq 0$ if and only if

$$\mathit{bmax}\,(\mathit{ceiling}\, x, \mathit{ceiling}\, x - \mathit{floor}\, x) \leq N.$$

7.49 Prove formally that $\mathit{prefix} \cdot \mathit{secure} \subseteq \mathit{secure} \cdot \mathit{prefix}$.

7.50 If x is secure and y is an arbitrary subsequence of x, is it necessarily the case that y is secure?

7.51 Give details of the appeal to fusion that establishes

$$\mathit{list\ secure} \cdot \mathit{partition} \quad = \quad (\![\mathit{wrap} \cdot \mathit{wrap}, \mathit{new} \cup \mathit{old}]\!).$$

7.52 Prove that cons is monotonic on R°.

7.53 Justify the claims (7.18), (7.19), (7.20), and (7.21).

7.54 The greedy algorithm produces a minimum length partition with a shortest possible first component. This means that the security van may be called upon before it is absolutely necessary to do so. Such a schedule might seem curious to the security van company. Outline how, by switching to snoc-lists, it is possible to reverse this phenomenon, obtaining a greedy schedule in which later visits are more frequent than early ones.

7.55 Give details of the 'baby-sized' applications of the greedy theorem to computing *ceiling* and *floor*.

7.56 The paragraph problem is to break a sequence of words into a sequence of non-empty lines with the aim of forming a 'visually pleasing' paragraph. The constraint is that no line in the paragraph should have a width that exceeds some fixed quantity W, where the width of a line x is the sum of the lengths of the words in x, plus some suitable value for the interword spaces. Calling the associated coreflexive *fits*, argue that *fits* is both prefix- and suffix-closed. Why is the following formulation *not* a reasonable specification of the problem?:

$$\mathit{paragraph} \quad \subseteq \quad \mathit{min}\, R \cdot \Lambda(\mathit{list}\, \mathit{fits} \cdot \mathit{partition}),$$

where $R = \mathit{length}^{\circ} \cdot \mathit{leq} \cdot \mathit{length}$. (*Hint*: Consider Exercise 7.54.)

7.57 Consider the ordering Q characterised by $[\,]\; Q\; ys$ and

$$([x] +\!\!\!+\; xs)\; Q\; ([y] +\!\!\!+\; ys) \quad \equiv \quad (x\; \mathit{prefix}\; y) \;\wedge\; (y\; \mathit{prefix}\; x \Rightarrow xs\; Q\; ys).$$

One can also define Q more succinctly by

$$Q \quad = \quad (\mathit{nil}^{\circ} \cdot !) \;\cup\; (\mathit{prefix}\, ;\, (\mathit{tail}^{\circ} \cdot Q \cdot \mathit{tail})).$$

This defines a preorder, and a linear order on partitions of the same sequence. Using only the fact that *secure* is prefix-closed, show that both *new* and *old* are monotonic on Q°. Although it is not true that $Q \subseteq R$, we nevertheless do have

$$\mathit{min}\, Q \cdot \Lambda(\![S]\!) \quad \subseteq \quad \mathit{min}\, R \cdot \Lambda(\![S]\!),$$

provided we also use the fact that *secure* is suffix-closed. In words, although Q is not a refinement of R, it is still the case that the (unique) minimum partition under Q is a minimum partition under R. The proof is a slightly tricky combinatorial argument. The advantage of taking this Q is that we can replace R by a more general preorder $R = \mathit{cost}^{\circ} \cdot \mathit{leq} \cdot \mathit{cost}$ and establish general properties of *cost* under which the greedy algorithm works. What are they?

Bibliographical remarks

Our own interest in optimisation problems originated in the calculus of functions referred to in earlier chapters. That work culminated in a study of greedy algorithms (Bird 1990, 1991, 1992a, 1992b, 1992c). Jeuring's work also concerns various kinds of greedy algorithm (Jeuring 1990, 1993). A recurring problem with these functional developments was the inadequate treatment of indeterminate specifications. These difficulties motivated the generalisation to relations.

The calculus of minimum elements, in the context of categories of relations, was first explored in (Brook 1977). Most of the ideas found there are also apparent in earlier work on the relational calculus, for instance (Riguet 1948). We adapted those works for applications to optimisation problems in (De Moor 1992a). Of course, the definitions in relational calculus are obvious, and have also been applied by others, see e.g. (Schmidt, Berghammer, and Zierer 1989).

Many researchers have attempted a classification of greedy algorithms before. An overview can be found in (Korte, Lovasz, and Schrader 1991), which proposes a mathematical structure called *greedoids* as a basis for the study of greedy algorithms. More recently, (Helman, Moret, and Shapiro 1993) have proposed a refinement of greedoids. Although there are some obvious links to the material presented in this book, we have not yet investigated the connection in sufficient detail. The theory of greedoids is much more concerned with structural properties than with the synthesis of greedy algorithms for given specifications. Also, greedoids can be characterised by the optimality of the greedy solution for a specific class of cost functions; no such equivalence is presented here.

Chapter 8

Thinning Algorithms

In this chapter we continue to study problems of the form $min\,R \cdot \Lambda(\![S]\!)$. The greedy theorem of the last chapter gave a rather strong condition under which such a problem could be solved by maintaining a single partial solution at each stage. At the other extreme, the Eilenberg–Wright lemma shows that $\Lambda(\![S]\!)$ can always be implemented as a set-valued catamorphism. This leads to an exhaustive search algorithm in which all possible partial solutions are maintained at each stage. Between the two extremes of all and one, there is a third possibility: at each stage keep a representative collection of partial solutions, namely those that might eventually be extended to an optimal solution. Such algorithms are called *thinning algorithms* and are the topic of the present chapter.

8.1 Thinning

Given a relation $Q : A \leftarrow A$, the relation $thin\,Q : \mathsf{P}A \leftarrow \mathsf{P}A$ is defined by

$$thin\,Q \quad = \quad (\in\backslash\in) \cap ((\ni \cdot Q)/\ni). \tag{8.1}$$

Informally, $thin\,Q$ is a nondeterministic mapping that takes a set y, and returns some subset x of y with the property that all elements of y have a lower bound under Q in x. To see this, note that $x(\in\backslash\in)y$ means that x is a subset of y, and

$$x((\ni \cdot Q)/\ni)y \quad \equiv \quad (\forall b \in y : \exists a \in x : aQb).$$

Thus, to thin a set x with $thin\,Q$ means to reduce the size of x without losing the possibility of taking a minimum element of x under Q. Unlike the case of $min\,R$, we can implement $thin\,Q$ when Q is not a connected preorder (see Section 8.3).

Definition (8.1) can be restated as the universal property

$$X \subseteq thin\,Q \cdot \Lambda S \quad \equiv \quad \in \cdot X \subseteq S \;\text{ and }\; X \cdot S^{\circ} \subseteq \ni \cdot Q,$$

which, like other universal properties, is often more useful in calculations.

Properties of thinning

It is immediate from the definition that $Q \subseteq R$ implies that $thin\ Q \subseteq thin\ R$. Furthermore, it is an easy exercise to show that $thin\ Q$ is reflexive if Q is reflexive, and transitive if Q is transitive. We will suppose in what follows that Q is a preorder, so $thin\ Q$ is a preorder too.

We can introduce *thin* into an optimisation problem with the following rule, called *thin-introduction*:

$$min\ R \quad = \quad min\ R \cdot thin\ Q \quad \text{provided that } Q \subseteq R.$$

The proof, left as an exercise, depends on the assumption that Q and R are preorders.

We can also eliminate *thin* from an optimisation problem:

$$thin\ Q \quad \supseteq \quad \tau \cdot min\ Q, \tag{8.2}$$

where $\tau : \mathsf{P}A \leftarrow A$ returns singleton sets. However, unless Q is a connected preorder, the domain of $\tau \cdot min\ Q$ is smaller than that of $thin\ Q$. For instance, $thin\ id$ is entire but the domain of $\tau \cdot min\ id$ consists only of singleton sets. So, use of (8.2) may result in an infeasible refinement. At the other extreme, $thin\ Q \supseteq id$, so $thin\ Q$ can always be refined to the identity relation on sets.

There is a useful variant of thin-elimination:

$$thin\ Q \cdot \Lambda S \quad \supseteq \quad \tau \cdot min\ R \cdot \Lambda S \quad \text{provided that } R \cap (S \cdot S^\circ) \subseteq Q. \tag{8.3}$$

For the proof, observe that by the universal property of *thin* we have to show

$$\begin{aligned} \in \cdot\, \tau \cdot min\ R \cdot \Lambda S &\subseteq S \\ \tau \cdot min\ R \cdot \Lambda S \cdot S^\circ &\subseteq \ni \cdot\, Q. \end{aligned}$$

The first inclusion is immediate from $\in \cdot\, \tau = id$. For the second, we argue:

$$\begin{aligned} & \tau \cdot min\ R \cdot \Lambda S \cdot S^\circ \subseteq \ni \cdot\, Q \\ \equiv \quad & \{\text{shunting } \tau \text{ and } \in \cdot\, \tau = id\} \\ & min\ R \cdot \Lambda S \cdot S^\circ \subseteq Q \\ \equiv \quad & \{\text{context}\} \\ & min\,(R \cap (S \cdot S^\circ)) \cdot \Lambda S \cdot S^\circ \subseteq Q \\ \Leftarrow \quad & \{\text{since } \Lambda S \cdot S^\circ \subseteq \ni\} \\ & min\,(R \cap (S \cdot S^\circ)) \cdot \ni \subseteq Q \\ \Leftarrow \quad & \{\text{definition of } min\} \\ & R \cap (S \cdot S^\circ) \subseteq Q. \end{aligned}$$

Finally, it is left as an exercise to prove that *thin* distributes over union:

$$thin\ Q \cdot union \quad \supseteq \quad union \cdot \mathsf{P}(thin\ Q). \tag{8.4}$$

The basic theorem

The following theorem and corollary show how the use of thinning can be exploited in solving optimisation problems. Both exhaustive search and the greedy algorithm follow as special cases. As usual, F is the base functor of the catamorphism.

Theorem 8.1 If S is monotonic on Q°, then

$$(\!|thin\ Q \cdot \Lambda(S \cdot \mathsf{F}\in)|\!) \quad \subseteq \quad thin\ Q \cdot \Lambda(\!|S|\!).$$

Proof. By the universal property of *thin* we have two conditions to check:

$$\begin{array}{rcl} \in \cdot (\!|thin\ Q \cdot \Lambda(S \cdot \mathsf{F}\in)|\!) & \subseteq & (\!|S|\!) \\ (\!|thin\ Q \cdot \Lambda(S \cdot \mathsf{F}\in)|\!) \cdot (\!|S|\!)^\circ & \subseteq & \ni \cdot Q. \end{array}$$

The first is an easy exercise in fusion and, by the hylomorphism theorem, the second follows if we can show that

$$thin\ Q \cdot \Lambda(S \cdot \mathsf{F}\in) \cdot \mathsf{F}(\ni \cdot Q) \cdot S^\circ \quad \subseteq \quad \ni \cdot Q.$$

We reason:

$$\begin{array}{ll} & thin\ Q \cdot \Lambda(S \cdot \mathsf{F}\in) \cdot \mathsf{F}(\ni \cdot Q) \cdot S^\circ \\ \subseteq & \quad \{\text{since } \mathsf{F}Q \cdot S^\circ \subseteq S^\circ \cdot Q \text{ by monotonicity and converses}\} \\ & thin\ Q \cdot \Lambda(S \cdot \mathsf{F}\in) \cdot \mathsf{F}\ni \cdot S^\circ \cdot Q \\ \subseteq & \quad \{\text{since } \Lambda X \cdot X^\circ \subseteq \ni\} \\ & thin\ Q \cdot \ni \cdot Q \\ \subseteq & \quad \{\text{since } thin\ Q \cdot \ni \subseteq \ni \cdot Q\} \\ & \ni \cdot Q \cdot Q \\ = & \quad \{\text{transitivity of } Q\} \\ & \ni \cdot Q. \end{array}$$

□

The following corollary is immediate on appeal to thin-introduction:

Corollary 8.1 If $Q \subseteq R$ and S is monotonic on Q°, then

$$min\ R \cdot (\!|thin\ Q \cdot \Lambda(S \cdot \mathsf{F}\in)|\!) \quad \subseteq \quad min\ R \cdot \Lambda(\!|S|\!).$$

Exercises

8.1 Prove that $thin\ id = id$.

8.2 Prove that $thin\ Q$ is a preorder if Q is.

8.3 Prove that $min\ R \supseteq min\ R \cdot thin\ Q$ if $Q \subseteq R$.

8.4 Prove that $cup \cdot \langle thin\ R, thin\ Q \rangle \subseteq thin\ R$ by showing more generally that

$$cup \cdot \langle thin\ R, subset \rangle \quad \subseteq \quad thin\ R,$$

where *subset* is the inclusion relation on sets. You will need the inclusion

$$\in \times \in \quad \subseteq \quad \langle \in, \in \rangle \cdot cup,$$

so prove that as well. Does equality hold in the original inclusion when $Q = R$?

8.5 Prove that $minR = \tau^\circ \cdot thin\ R$ and hence prove the thin-elimination rule.

8.6 Prove that $thin\ Q \cdot \Lambda S = thin\,(Q \cap (S \cdot S^\circ)) \cdot \Lambda S$.

8.7 Prove (8.4). Is the converse inclusion true?

8.8 Prove that the greedy algorithm is a special case of Theorem 8.1.

8.9 Show that if Q is well-supported (see Exercise 7.30), then $\Lambda(mnl\ Q) \subseteq thin\ Q$.

8.2 Paths in a layered network

Let us now give a simple illustration of the ideas introduced so far. The example is similar to the paths on a cylinder problem given in the preceding chapter.

By definition, a *layered network* is a non-empty sequence of sets of vertices. A *path* in a layered network $xs = [x_0, x_1, \ldots, x_n]$ is a sequence of vertices $[a_0, a_1, \ldots, a_n]$ where $a_j \in x_j$ for $0 \leq j \leq n$. With each path is associated a cost, defined by

$$cost\,[a_0, a_1, \ldots, a_n] \quad = \quad (+j : 0 \leq j < n : wt\,(a_j, a_{j+1})),$$

where *wt* is some given function on pairs of vertices. We aim to derive an algorithm for finding a least cost path in a layered network.

To formalise the problem we will use non-empty cons-lists, thereby building paths from right to left. The choice is dictated solely by reasons of efficiency in the final functional program, since snoc-lists would have served equally well. Thus the input is an element of $list^+(\mathsf{P}A)$. Our problem takes the form

$$minpath \quad \subseteq \quad min\ R \cdot \Lambda(list^+ \in),$$

where $R = cost^\circ \cdot leq \cdot cost$. Using the definition of $list^+$ as a catamorphism, we obtain that

$$minpath \subseteq min\, R \cdot \Lambda(\![\alpha \cdot \mathsf{F}(\in, id)]\!),$$

where $\alpha = [wrap, cons]$ and F is the base bifunctor of non-empty cons-lists.

It remains to define $cost$. This is not a catamorphism on paths, but we do have

$$cost = outr \cdot (\![wrapz, consw]\!)$$

where $wrapz = \langle wrap, zero\rangle$ and

$$consw(a,(x,n)) = (cons\,(a,x), wt\,(a, head\, x) + n).$$

Thus $(\![wrapz, consw]\!) = \langle id, cost\rangle$.

Derivation

In this example we have $S = \alpha \cdot \mathsf{F}(\in, id)$. The corollary to the thinning theorem says that

$$min\, R \cdot (\![thin\, Q \cdot \Lambda(\alpha \cdot \mathsf{F}(\in, \in))]\!) \subseteq min\, R \cdot \Lambda(\![\alpha \cdot \mathsf{F}(\in, id)]\!)$$

for any $Q \subseteq R$ satisfying

$$\alpha \cdot \mathsf{F}(\in, Q^\circ) \subseteq Q^\circ \cdot \alpha \cdot \mathsf{F}(\in, id).$$

Of course, if we can take $Q = R$, then we can appeal to the greedy theorem, avoiding thinning altogether. To show that we cannot take $Q = R$, suppose p and q are two paths in the network $[x_1, \ldots, x_n]$ with $cost\, p \geq cost\, q$. Then the monotonicity condition with $Q = R$ says that for any set of vertices x_0, and any $a \in x_0$, there exists a $b \in x_0$ such that

$$cost\,([a] +\!\!+\, p) \geq cost([b] +\!\!+\, q).$$

In particular, this condition should hold when $x_0 = \{a\}$, and so $a = b$. Using $cost\,([a] +\!\!+\, p) = wt\,(a, head\, p) + cost\, p$, we therefore require

$$wt\,(a, head\, p) - wt\,(a, head\, q) \geq cost\, q - cost\, p.$$

However, since $wt(a, head\, q)$ can be arbitrarily large, this condition fails unless $head\, p = head\, q$. On the other hand, if $head\, p = head\, q$, then the inequality reduces to $cost\, p \geq cost\, q$, which is true by assumption.

It follows that $\alpha \cdot \mathsf{F}(\in, id)$ is monotonic on Q°, where $Q = R \cap (head^\circ \cdot head)$. Hence

$$minpath \subseteq min\, R \cdot (\![thin\, Q \cdot \Lambda(\alpha \cdot \mathsf{F}(\in, \in))]\!).$$

Operationally speaking, the catamorphism on the right maintains a set of partial solutions, with at least one solution for each starting vertex. But, clearly, only one partial solution needs to be maintained for each vertex v, namely, a shortest path beginning with v. This motivates the following calculation, in which the term *thin Q* is eliminated:

$$\begin{array}{ll}
 & \mathit{thin}\ Q \cdot \Lambda(\alpha \cdot \mathsf{F}(\in, \in)) \\
= & \quad \{\text{bifunctors}\} \\
 & \mathit{thin}\ Q \cdot \Lambda(\alpha \cdot \mathsf{F}(\mathit{id}, \in) \cdot \mathsf{F}(\in, \mathit{id})) \\
= & \quad \{\text{power transpose of composition}\} \\
 & \mathit{thin}\ Q \cdot \mathit{union} \cdot \mathsf{P}\Lambda(\alpha \cdot \mathsf{F}(\mathit{id}, \in)) \cdot \Lambda\mathsf{F}(\in, \mathit{id}) \\
\supseteq & \quad \{\mathit{thin}\text{ distributes over }\mathit{union}\text{ (8.4)}\} \\
 & \mathit{union} \cdot \mathsf{P}(\mathit{thin}\ Q \cdot \Lambda(\alpha \cdot \mathsf{F}(\mathit{id}, \in))) \cdot \Lambda\mathsf{F}(\in, \mathit{id}) \\
\supseteq & \quad \{\text{thin-elimination (8.3) -- see below}\} \\
 & \mathit{union} \cdot \mathsf{P}(\tau \cdot \mathit{min}\ R \cdot \Lambda(\alpha \cdot \mathsf{F}(\mathit{id}, \in))) \cdot \Lambda\mathsf{F}(\in, \mathit{id}) \\
= & \quad \{\text{since }\mathit{union} \cdot \mathsf{P}\tau = \mathit{id}\} \\
 & \mathsf{P}(\mathit{min}\ R \cdot \Lambda(\alpha \cdot \mathsf{F}(\mathit{id}, \in))) \cdot \Lambda\mathsf{F}(\in, \mathit{id}) \\
= & \quad \{\text{since }\mathsf{P} = \mathsf{E}\text{ on functions}\} \\
 & \mathsf{P}(\mathit{min}\ R \cdot \mathsf{P}\alpha \cdot \Lambda\mathsf{F}(\mathit{id}, \in)) \cdot \Lambda\mathsf{F}(\in, \mathit{id})
\end{array}$$

To justify the appeal to (8.3) we have to show that $R \cap (S \cdot S^\circ) \subseteq Q$:

$$\begin{array}{ll}
 & R \cap (S \cdot S^\circ) \subseteq Q \\
\Leftarrow & \quad \{\text{definition of } Q\} \\
 & S \cdot S^\circ \subseteq \mathit{head}^\circ \cdot \mathit{head} \\
\equiv & \quad \{\text{shunting}\} \\
 & (\mathit{head} \cdot S) \cdot (\mathit{head} \cdot S)^\circ \subseteq \mathit{id} \\
\Leftarrow & \quad \{\text{since }\mathit{head} \cdot S \subseteq [\mathit{id}, \mathit{outl}]\text{ (exercise), so }\mathit{head} \cdot S\text{ is simple}\} \\
 & \mathit{true}.
\end{array}$$

The above derivation is quite general and makes hardly any use of the specific datatype. For the base bifunctor F of non-empty cons-lists we have

$$\begin{array}{rcl}
\Lambda\mathsf{F}(\in, \mathit{id}) & = & \mathit{id} + \mathit{cpl} \\
\mathit{min}\ R \cdot \mathsf{P}\alpha \cdot \Lambda\mathsf{F}(\mathit{id}, \in) & = & [\mathit{wrap}, \mathit{step}] \\
\mathit{step} & = & \mathit{min}\ R \cdot \mathsf{P}\mathit{cons} \cdot \mathit{cpr},
\end{array}$$

where the functions *cpl* and *cpr* were defined in Section 5.6. Hence, finally, we have

$$\mathit{minpath} \subseteq \mathit{min}\ R \cdot (\![\mathsf{P}\mathit{wrap}, \mathsf{P}\mathit{step} \cdot \mathit{cpl}]\!).$$

The program

In the Gofer program we represent sets by lists in the usual way, and represent a path p by $(p, (head\,p, cost\,p))$. The program is parameterised by the function `wt`:

```
> path = minlist r . catallist (list wrap', list step . cpl)
> step = minlist r . list cons'. cpr
> r    = leq . cross (cost, cost)
> cost = outr . outr

> wrap'   = pair (wrap, pair (id, zero))
> cons'   = cross (cons, augment) . dupl
> augment = pair (outl, plus . cross (wt, id) . assocl)
```

Exercises

8.10 Can we replace the cons-list bifunctor with one or both of the following bifunctors?

$$\begin{aligned} \mathsf{F}(A, B) &= A + (A \times (B \times B)) \\ \mathsf{F}(A, B) &= A + (B \times B). \end{aligned}$$

What is the interpretation of the generalised layered network problem?

8.11 The derivation above is an instance of the following more general result. Suppose $Q \subseteq R$ and $S = S_1 \cdot S_2$ is monotonic on Q°. Furthermore, suppose $R \cap (S_1 \cdot S_1{}^\circ) \subseteq Q$. Then

$$min\,R \cdot (\![\mathsf{P}(min\,R \cdot \Lambda S_1) \cdot \Lambda(S_2 \cdot \mathsf{F}\!\in)]\!) \quad \subseteq \quad min\,R \cdot \Lambda(\![S]\!).$$

Prove this result.

8.3 Implementing thin

In the layered network example we were fortunate in that the thinning step could be eliminated, but most often we have to implement thinning as part of the final algorithm. As with $min\,R$ we cannot refine $thin\,Q$ to an implementable function except when $thin\,Q$ is applied to finite sets; unlike $min\,R$ we do not require the sets to be non-empty, nor that Q be a connected preorder.

The function $thinlist\,Q$ might be specified by

$$setify \cdot thinlist\,Q \quad \subseteq \quad thin\,Q \cdot setify,$$

where $setify : \mathsf{P}A \leftarrow list\, A$. However, we want to impose an extra condition upon $thinlist\; Q$, namely that

$$thinlist\; Q \;\subseteq\; subseq.$$

In words, we want $thinlist\; Q$ to preserve the relative order of the elements in the list. The reason for this additional restriction will emerge below.

The ideal implementation of $thinlist\; Q$ is a linear-time program that produces a shortest possible result. In particular, when Q is a connected preorder and x is a non-empty list, we want

$$thinlist\; Q\, x \;=\; [minlist\; Q\, x], \tag{8.5}$$

where $minlist\; Q$ was defined in the preceding chapter.

A legitimate, but not useful, implementation is to take $thinlist\; Q = id$. Another is to remove an element from a list if it is 'bumped' by one of its neighbours. This idea is formalised in the definition

$$thinlist\; Q \;=\; (\![nil, bump\; Q]\!),$$

where

$$\begin{aligned} bump\; Q\,(a, [\,]) &= [a] \\ bump\; Q\,(a, [b] +\!\!+ x) &= (aQb \rightarrow [a] +\!\!+ x,\, bQa \rightarrow [b] +\!\!+ x,\; [a] +\!\!+ [b] +\!\!+ x). \end{aligned}$$

This gives a linear-time algorithm in the number of evaluations of Q, though it is not always guaranteed to deliver a shortest result. There are other possible choices for $thinlist\; Q$, some of which are explored in the exercises.

Sorting sets

In the main theorem of this section we make use of the idea of maintaining a finite set as a sorted list. We will use a version of $sort$ from Chapter 6, taking

$$sort\; P \;=\; ordered\; P \cdot setify^{\circ},$$

where $P : A \leftarrow A$ is some connected preorder. Note that $sort\; P$ is not a function, even when P is a linear order: for example, $sort\; leq\,\{1,2,3\}$ may produce $[1,2,3]$ or $[1,1,2,3]$, or any one of a number of similar lists.

We will make use of a number of facts about $sort\; P$ including

$$thinlist\; Q \cdot sort\; P \;\subseteq\; sort\; P \cdot thin\; Q. \tag{8.6}$$

For the proof we argue:

$$
\begin{array}{cl}
& \mathit{thinlist}\ Q \cdot \mathit{sort}\ P \\
= & \quad \{\text{definition of } \mathit{sort}\ P\} \\
& \mathit{thinlist}\ Q \cdot \mathit{ordered}\ P \cdot \mathit{setify}^{\circ} \\
\subseteq & \quad \{\text{claim: } \mathit{thinlist}\ Q \cdot \mathit{ordered}\ P \subseteq \mathit{ordered}\ P \cdot \mathit{thinlist}\ Q\} \\
& \mathit{ordered}\ P \cdot \mathit{thinlist}\ Q \cdot \mathit{setify}^{\circ} \\
\subseteq & \quad \{\text{specification of } \mathit{thinlist}\ Q \text{ and shunting}\} \\
& \mathit{ordered}\ P \cdot \mathit{setify}^{\circ} \cdot \mathit{thin}\ Q \\
= & \quad \{\text{definition of } \mathit{sort}\ P\} \\
& \mathit{sort}\ P \cdot \mathit{thin}\ Q.
\end{array}
$$

For the claim it is sufficient to show that $\mathit{thinlist}\ Q \cdot \mathit{ordered}\ P \subseteq \mathit{ordered}\ P$:

$$
\begin{array}{cl}
& \mathit{thinlist}\ Q \cdot \mathit{ordered}\ P \\
\subseteq & \quad \{\text{specification of } \mathit{thinlist}\ Q\} \\
& \mathit{subseq} \cdot \mathit{ordered}\ P \\
\subseteq & \quad \{\text{since } \mathit{subseq} \cdot \mathit{ordered}\ P \subseteq \mathit{ordered}\ P \text{ if } P \text{ is connected}\} \\
& \mathit{ordered}\ P.
\end{array}
$$

It is important to note that the choice of P can affect the success of the subsequent thinning process; ideally, $\mathit{sort}\ P$ should bring together elements that are comparable under Q. In particular, if Q is connected and we take $P = Q$, then thinning is accomplished by simply returning the first element as a singleton list.

There are five other properties about $\mathit{sort}\ P$ that we will need. Proofs are left as exercises. The first four are

$$
\begin{array}{rcll}
\mathit{minlist}\ Q \cdot \mathit{sort}\ P & \subseteq & \mathit{min}\ Q & (8.7) \\
\mathit{list}\ f \cdot \mathit{sort}\,(f^{\circ} \cdot P \cdot f) & \subseteq & \mathit{sort}\ P \cdot \mathsf{P}f & (8.8) \\
\mathit{filter}\ p \cdot \mathit{sort}\ P & \subseteq & \mathit{sort}\ P \cdot \mathsf{E}p & (8.9) \\
\mathit{merge}\ P \cdot (\mathit{sort}\ P)^2 & \subseteq & \mathit{sort}\ P \cdot \mathit{cup}. & (8.10)
\end{array}
$$

In (8.9) the relation p is assumed to be a coreflexive, and in (8.10) the function $\mathit{merge}\ P$ is as defined in Exercise 6.28.

The fifth property deals with an implementation of the general cartesian product function $\mathit{cp}\,(\mathsf{F}) = \Lambda(\mathsf{F}\!\in)$ described in Section 5.6. We met the special case $\mathit{cp}\,(\mathit{list})$ in the paths in a layered network example. The function $\mathit{cp}\,(\mathsf{F})$ is a natural transformation of type $\mathsf{PF} \leftarrow \mathsf{FP}$, so we are looking for a function $\mathit{listcp}\,(\mathsf{F})$ with type $\mathit{list} \cdot \mathsf{F} \leftarrow \mathsf{F} \cdot \mathit{list}$. Moreover, we want this function to satisfy the condition

$$\mathit{listcp}\,(\mathsf{F}) \cdot \mathsf{F}(\mathit{sort}\ P) \quad \subseteq \quad \mathit{sort}\,(\mathsf{F}P) \cdot \mathit{cp}\,(\mathsf{F}). \qquad (8.11)$$

Not every functor F admits an implementation of *listcp* (F) satisfying (8.11); one requirement is that F distributes over arbitrary joins. It is left as an exercise to define *listcp* (F) for each polynomial functor F. It follows that if F is polynomial and distributes over arbitrary joins (such a functor is called *linear*), then (8.11) can be satisfied. In what follows we will assume that (8.11) can be satisfied.

Inclusions (8.8), (8.9) and (8.11) are used in the proof of the following lemma, which is required in the theorem to come:

Lemma 8.1 If f is monotonic on R and p is a coreflexive, then

$$\mathit{filter}\ p \cdot \mathit{list}\ f \cdot \mathit{listcp}\,(\mathsf{F}) \cdot \mathsf{F}(\mathit{sort}\ R) \quad \subseteq \quad \mathit{sort}\ R \cdot \Lambda(p \cdot f \cdot \mathsf{F}{\in}).$$

Proof. The proof is a simple calculation:

$$\begin{array}{ll}
 & \mathit{sort}\ P \cdot \Lambda(p \cdot f \cdot \mathsf{F}{\in}) \\
= & \quad \{\Lambda \text{ of composition and } \mathit{cp}\,(\mathsf{F}) = \Lambda\mathsf{F}{\in}\} \\
 & \mathit{sort}\ P \cdot \mathsf{E}(p \cdot f) \cdot \mathit{cp}\,(\mathsf{F}) \\
= & \quad \{\mathsf{E} \text{ is a functor and agrees with } \mathsf{P} \text{ on functions}\} \\
 & \mathit{sort}\ P \cdot \mathsf{E}p \cdot \mathsf{P}f \cdot \mathit{cp}\,(\mathsf{F}) \\
\supseteq & \quad \{(8.9)\} \\
 & \mathit{filter}\ p \cdot \mathit{sort}\ P \cdot \mathsf{P}f \cdot \mathit{cp}\,(\mathsf{F}) \\
\supseteq & \quad \{(8.8)\} \\
 & \mathit{filter}\ p \cdot \mathit{list}\ f \cdot \mathit{sort}\,(f^\circ \cdot P \cdot f) \cdot \mathit{cp}\,(\mathsf{F}) \\
\supseteq & \quad \{\text{since } f \text{ is monotonic on } P\} \\
 & \mathit{filter}\ p \cdot \mathit{list}\ f \cdot \mathit{sort}(\mathsf{F}P) \cdot \mathit{cp}\,(\mathsf{F}) \\
\supseteq & \quad \{(8.11)\} \\
 & \mathit{filter}\ p \cdot \mathit{list}\ f \cdot \mathit{listcp}\,(\mathsf{F}) \cdot \mathsf{F}(\mathit{sort}\ P).
\end{array}$$

□

Binary thinning

With these preliminaries out of the way, the main theorem of this section can now be stated. It will be referred to subsequently as the *binary* thinning theorem.

Theorem 8.2 Suppose the following three conditions are satisfied:

1. $S = (p_1 \cdot f_1) \cup (p_2 \cdot f_2)$, where p_1 and p_2 are coreflexives.

2. Q is a preorder with $Q \subseteq R$ and such that $p_1 \cdot f_1$ and $p_2 \cdot f_2$ are both monotonic on Q°.

3. P is a connected preorder such that f_1 and f_2 are both monotonic on P.

Then

$$minlist\ R \cdot (\![thinlist\ Q \cdot merge\ P \cdot \langle g_1, g_2 \rangle \cdot listcp]\!) \quad \subseteq \quad min\ R \cdot \Lambda(\![S]\!),$$

where $g_i = filter\ p_i \cdot list\ f_i$.

Proof. We reason:

$$\begin{array}{ll} & min\ R \cdot \Lambda(\![S]\!) \\ \supseteq & \quad \{\text{thinning theorem since } S \text{ is monotonic on } Q^\circ\} \\ & min\ R \cdot (\![thin\ Q \cdot \Lambda(S \cdot \mathsf{F}\in)]\!) \\ \supseteq & \quad \{(8.7)\} \\ & minlist\ R \cdot sort\ P \cdot (\![thin\ Q \cdot \Lambda(S \cdot \mathsf{F}\in)]\!) \\ \supseteq & \quad \{\text{fusion}\} \\ & minlist\ R \cdot (\![thinlist\ Q \cdot merge\ P \cdot \langle g_1, g_2 \rangle \cdot listcp]\!). \end{array}$$

The condition for fusion in the last step is verified as follows:

$$\begin{array}{ll} & sort\ P \cdot thin\ Q \cdot \Lambda(S \cdot \mathsf{F}\in) \\ \supseteq & \quad \{(8.6)\} \\ & thinlist\ Q \cdot sort\ P \cdot \Lambda(S \cdot \mathsf{F}\in) \\ = & \quad \{\text{definition of } S\} \\ & thinlist\ Q \cdot sort\ P \cdot cup \cdot \langle \Lambda(p_1 \cdot f_1 \cdot \mathsf{F}\in), \Lambda(p_2 \cdot f_2 \cdot \mathsf{F}\in) \rangle \\ \supseteq & \quad \{(8.10)\} \\ & thinlist\ Q \cdot merge\ P \cdot \langle sort\ P \cdot \Lambda(p_1 \cdot f_1 \cdot \mathsf{F}\in), sort\ P \cdot \Lambda(p_2 \cdot f_2 \cdot \mathsf{F}\in) \rangle \\ \supseteq & \quad \{\text{Lemma 8.1}\} \\ & thinlist\ Q \cdot merge\ P \cdot \langle g_1, g_2 \rangle \cdot listcp \cdot \mathsf{F}(sort\ P). \end{array}$$

□

The theorem can be generalised in the obvious way when S is a collection $S = (p_1 \cdot f_1) \cup \cdots \cup (p_n \cdot f_n)$. We leave details as an exercise.

Exercises

8.12 Another definition of *thinlist* Q is as a catamorphism $(\![id, bump\ Q]\!)$ on snoc-lists. Define *bump* Q and give an example to show that this version of *thinlist* Q differs from that of the text.

8.13 Yet another definition is

$$\begin{array}{rcl} \mathit{thinlist}\ Q\,[\,] & = & [\,] \\ \mathit{thinlist}\ Q\,[a] & = & [a] \\ \mathit{thinlist}\ Q\,([a] \mathbin{+\!\!+} [b] \mathbin{+\!\!+} x) & = & \begin{cases} \mathit{thinlist}\ Q\,([a] \mathbin{+\!\!+} x), & \text{if } aQb \\ \mathit{thinlist}\ Q\,([b] \mathbin{+\!\!+} x), & \text{if } bQa \\ [a] \mathbin{+\!\!+} \mathit{thinlist}\ Q\,([b] \mathbin{+\!\!+} x), & \text{otherwise} \end{cases} \end{array}$$

Give examples to show that this version of *thinlist* Q may return a shorter or longer result than that of the text.

8.14 Yet another definition arises from the specification

$$\mathit{thinlist}\ Q \ \subseteq\ \mathit{list}\,(\mathit{minlist}\ Q) \cdot \mathit{min}\ L \cdot \Lambda(\mathit{list}^{+}(\mathit{connected}\ Q) \cdot \mathit{partition}),$$

where $L = \mathit{length}^{\circ} \cdot \mathit{leq} \cdot \mathit{length}$ and the coreflexive *connected* Q is defined by the associated predicate

$$\mathit{connected}\ Q\ x \ \equiv\ (\forall a : a\ \mathit{inlist}\ x : (\forall b : b\ \mathit{inlist}\ x : aQb \vee bQa)).$$

In words, we partition a list into the smallest number of components, each of whose elements are all connected under Q, and then take a minimum under Q of each component. Use the fact that *connected* Q is prefix-closed (in fact, subsequence-closed) to give a greedy algorithm for the optimisation problem on the right. Apply type functor fusion to obtain a catamorphism for *thinlist* Q.

How can the catamorphism be expressed as a more efficient algorithm if it is assumed that $Q \cdot Q^{\circ} \subseteq Q \cup Q^{\circ}$?

8.15 Repeat the above exercise, replacing *connected* Q by *leftmin* Q, where

$$\mathit{leftmin}\ Q\,([a] \mathbin{+\!\!+} x) \ \equiv\ (\forall b : b\ \mathit{inlist}\ x : aQb).$$

8.16 A best possible implementation of *thinlist* Q would be an algorithm that returned the subsequence of minimal elements under Q. Can such an algorithm be implemented in linear time in the number of Q evaluations?

8.17 Prove that $\mathit{subseq} \cdot \mathit{sort}\ P \subseteq \mathit{sort}\ P \cdot \mathit{subset}$ provided P is a connected preorder.

8.18 Prove (8.8) and (8.9).

8.19 Give functions for $\mathit{listcp}\,(\mathsf{F} \times \mathsf{G})$ and $\mathit{listcp}\,(\mathsf{F} + \mathsf{G})$ in terms of $\mathit{listcp}\,(\mathsf{F})$ and $\mathit{listcp}\,(\mathsf{G})$. What is $\mathit{listcp}\,(\mathsf{F})$ when F is the identity functor, or the constant functor $\mathsf{K}A$?

8.20 Can you define $\mathit{listcp}\,(\mathsf{T})$ for an arbitrary type functor T?

8.21 Give a counter-example showing that (8.11) fails for non-linear polynomial relators.

8.22 Formalise and prove a version of binary thinning in which the algebra S takes the form $S = (f_1 \cdot p_1) \cup (f_2 \cdot p_2)$.

8.4 The knapsack problem

The standard example of binary thinning is the well-known knapsack problem (Martello and Toth 1990). The objective is to pack items in a knapsack in the best possible way. Given is a list of items which might be packed, each of which has a given *weight* and *value*, both of which are non-negative real numbers. The knapsack has a finite capacity w, giving an upper bound to the total weight of the packed items, and the object of the exercise is to pack items with a greatest total value, subject to the capacity of the knapsack not being exceeded.

Let *Item* denote the type of items to be packed and $val, wt : Real \leftarrow Item$ the associated value and weight functions. The input consists of an element x of type *list Item* and a given capacity w.

We will model selections as subsequences of the given list of items. The relation $subseq : list\ A \leftarrow list\ A$ can be expressed in the form

$$subseq \quad = \quad (\![[nil, cons] \cup [nil, outr]]\!).$$

The total value and weight of a selection are given by two functions $value, weight : Real \leftarrow list\ Item$, defined by

$$\begin{aligned} value &= sum \cdot list\ val \\ weight &= sum \cdot list\ wt. \end{aligned}$$

Our problem is to find a function $knapsack\ w$ satisfying

$$knapsack\ w \quad \subseteq \quad max\ R \cdot \Lambda(within\ w \cdot subseq),$$

where $R = value^{\circ} \cdot leq \cdot value$ and $within\ w\ x = (weight\ x \leq w)$. Equivalently, replacing R by R° we obtain

$$knapsack\ w \quad \subseteq \quad min\ R \cdot \Lambda(within\ w \cdot subseq),$$

where $R = value^{\circ} \cdot geq \cdot value$ and $geq = leq^{\circ}$.

An appeal to fusion, using the fact that weights are non-negative, gives

$$within\ w \cdot subseq \quad = \quad (\![(within\ w \cdot [nil, cons]) \ \cup \ [nil, outr]]\!).$$

Of course, the right-hand side simplifies to $(\![nil, (within\ w \cdot cons) \cup outr]\!)$; the form above suggests that binary thinning might be applicable.

Derivation

We first check to see whether $(within\ w \cdot [nil, cons]) \cup [nil, outr]$ is monotonic on $R^\circ = value^\circ \cdot leq \cdot value$; if it is, then a greedy algorithm is possible. It is easy to prove that $[nil, cons]$ and $[nil, outr]$ are both monotonic on R°, but the problem is that $within\ w \cdot [nil, cons]$ is not. It does not follow that if $value\ x \leq value\ y$ and $within\ w\ ([a] +\!\!\!+\ x)$, then either $within\ w\ ([a] +\!\!\!+\ y)$ or $value\ ([a] +\!\!\!+\ x) \leq value\ y$.

On the other hand, it is easy to prove that $within\ w \cdot [nil, cons]$ is monotonic on Q°, where

$$Q \quad = \quad R \cap (weight^\circ \cdot leq \cdot weight).$$

Furthermore, $[nil, outr]$ is monotonic on Q°. Since the base functor of cons-lists is linear, all the conditions of the binary thinning theorem are in place if we take $P = R$, thereby sorting in descending order of *value*.

The result is that we can implement $knapsack\ w$ as the function

$$minlist\ R \cdot (\![thinlist\ Q \cdot merge\ R \cdot \langle g_1, g_2 \rangle \cdot listcp]\!),$$

where $g_1 = filter\ (within\ w) \cdot list\ [nil, cons]$ and $g_2 = list\ [nil, outr]$.

The implementation can be simplified. For the functor $\mathsf{F}A = 1 + (Item \times A)$ we have

$$listcp \quad = \quad listcp(\mathsf{F}) \quad = \quad wrap + cpr.$$

Furthermore, $g_1 = [list\ nil, h_1]$ and $g_2 = [list\ nil, h_2]$, where

$$\begin{aligned} h_1 &\quad = \quad filter\ (within\ w) \cdot list\ cons \\ h_2 &\quad = \quad list\ outr. \end{aligned}$$

An easy simplification now yields:

$$knapsack\ w \quad = \quad minlist\ R \cdot (\![nil, thinlist\ Q \cdot merge\ R \cdot \langle h_1, h_2 \rangle \cdot cpr]\!).$$

Finally, since packings are produced in descending order of value, we can replace $minlist\ R$ by *head*.

The program

We represent a list x of items by the pair $(x, (value\, x, weight\, x))$. The following program is parameterised by the functions `val` and `wt`:

```
> knapsack w = head . catalist (start, thinlist q . merge r . step w)

> start      = [([],(0,0))]
> step w     = pair (filter (within w) . list cons', list outr) . cpr
> within w   = (<= w) . weight

> cons'   = cross (cons, augment) . dupl
> augment = cross (addin val, addin wt) . dupl
> addin f = plus . cross (f, id)

> r = geq . cross (value, value)
> p = leq . cross (weight, weight)
> q = meet (p,r)

> value  = outl . outr

> weight = outr . outr
```

The algorithm, though it takes exponential time in the worst case, is quite efficient in practice. The knapsack problem is presented in many text books as an application of dynamic programming, in which a recursive formulation of the problem is implemented efficiently under the assumption that the weights and capacity are integers. Dynamic programming will be the topic of the next chapter, but the thinning approach to knapsack gives a simpler algorithm that does not depend on the inputs being integers. Moreover, if the weights and capacity are integers, then the algorithm is as efficient as the dynamic programming scheme.

8.5 The paragraph problem

The next application of the binary thinning theorem is to the paragraph problem (Bird 1986; Knuth and Plass 1981). The problem has already been touched on briefly in Exercise 7.56. Three inputs are given: a non-empty sequence of words, a function *length* that returns the length of a word, and a number w giving the maximum possible line width. The width of a line is the sum of the widths of its words plus some measure of the interword spaces. It is assumed that w is sufficiently large that any word will at least fit on a line by itself.

By definition, a line is a non-empty sequence of words, and a paragraph is a non-empty sequence of lines; thus

$$\begin{aligned} \mathit{Line} &= \mathit{list}^+\ \mathit{Word} \\ \mathit{Para} &= \mathit{list}^+\ \mathit{Line}. \end{aligned}$$

We will build paragraphs from right to left, so our lists are cons-lists. Certainly, no sensible greedy algorithm for the paragraph problem can be based on cons-lists (see Exercise 7.56), but thinning algorithms consider all possibilities and are not sensitive to the kind of list being used.

The problem is to find a function *paragraph w* satisfying

$$\mathit{paragraph}\ w \quad \subseteq \quad \mathit{min}\ R \cdot \Lambda(\mathit{list}^+\ (\mathit{fits}\ w) \cdot \mathit{partition}),$$

where $R = (\mathit{waste}\ w)^\circ \cdot \mathit{leq} \cdot (\mathit{waste}\ w)$ and *waste w* is a measure of the waste incurred by a particular paragraph given the maximum width w.

To complete the specification we need to define *waste w*, *fits w* and *partition*. The type assigned to *partition* is $\mathit{Para} \leftarrow \mathit{list}^+\ \mathit{Word}$ and we can define it as a catamorphism on non-empty lists by changing the definition given in Section 5.6 slightly:

$$\mathit{partition} \quad = \quad (\![\mathit{wrap} \cdot \mathit{wrap}, \mathit{new} \cup \mathit{glue}]\!),$$

where

$$\begin{aligned} \mathit{new}\,(a, xs) &= [[a]] +\!\!+\ xs \\ \mathit{glue}\,(a, xs) &= [[a] +\!\!+\ \mathit{head}\ xs] +\!\!+\ \mathit{tail}\ xs. \end{aligned}$$

Note that *glue* is a (total) function on non-empty lists, but only a partial function on possibly empty lists. We will need the fact that *glue* is a function in the thinning algorithm to come.

The coreflexive *fits w* holds on a line x if $\mathit{width}\ x \leq w$, where *width* is given by a catamorphism on non-empty lists:

$$\mathit{width} \quad = \quad (\![\mathit{length}, \mathit{succ} \cdot \mathit{plus} \cdot (\mathit{length} \times \mathit{id})]\!).$$

It is assumed that interword spaces contribute one unit toward the width of a line, which accounts for the term *succ* in the catamorphism above.

Finally, the function *waste w* depends on the 'white-space' that occurs at the end of all the lines of the paragraph, *except* for the very last line, which, by definition, has no white-space associated with it. Formally,

$$\mathit{waste}\ w \quad = \quad \mathit{collect} \cdot \mathit{list}\,(\mathit{white}\ w) \cdot \mathit{init},$$

Before proceeding with the derivation of an algorithm, we note that the obvious greedy algorithm does not solve this specification.	Before proceeding with the derivation of an algorithm, we note that the obvious greedy algorithm does not solve this specification.

Figure 8.1: A greedy and an optimal paragraph.

where $\mathit{init} : \mathit{list}\ A \leftarrow \mathit{list}^+\ A$ removes the last element from a list, and

$$\mathit{white}\ w\ x \quad = \quad (w - \mathit{width}\ x)\cdot$$

Provided it satisfies certain properties, the precise definition of *collect* is not too important, but for concreteness we will take

$$\mathit{collect} \quad = \quad \mathit{sum} \cdot \mathit{list}\ \mathit{sqr},$$

where $\mathit{sqr}\ m = m^2$. This definition is suggested in (Knuth and Plass 1981).

After an appeal to fusion, using the assumption that each individual word will fit on a line by itself, we can phrase the paragraph problem in the form

$$\mathit{paragraph}\ w \quad \subseteq \quad \mathit{min}\ R \cdot \Lambda(\![\mathit{wrap} \cdot \mathit{wrap}, \mathit{new} \cup (\mathit{ok}\ w \cdot \mathit{glue})]\!),$$

where $\mathit{ok}\ w$ holds on $([x] +\!\!+\ \mathit{xs})$ if $\mathit{width}\ x \leq w$. Since an individual word will fit on a line by itself, we can rewrite the algebra of the catamorphism in the form

$$[\mathit{wrap} \cdot \mathit{wrap}, \mathit{new}] \cup \mathit{ok}\ w \cdot [\mathit{wrap} \cdot \mathit{wrap}, \mathit{glue}].$$

Since *new* and *glue* are both functions, we see that the problem is of a kind to which binary thinning may be applicable.

Derivation

Before proceeding with the derivation of an algorithm, we note that the obvious greedy algorithm does not solve this specification. The greedy algorithm is a left to right algorithm, filling lines for as long as possible before starting a new line. The left-hand side of Figure 8.1 shows the output of the greedy algorithm on the opening sentence of this section, and an optimal paragraph (with the given definition of *collect*) on the right.

One reason why the greedy algorithm fails is that *glue* is not monotonic on R°. Even for paragraphs $[x] \mathbin{+\!\!+} xs$ and $[y] \mathbin{+\!\!+} ys$ of the same input, the implication

$$\begin{aligned} & waste\,([x] \mathbin{+\!\!+} xs) \geq waste\,([y] \mathbin{+\!\!+} ys) \\ & \quad \Rightarrow waste\,([[a] \mathbin{+\!\!+} x] \mathbin{+\!\!+} xs) \geq waste\,([[a] \mathbin{+\!\!+} y] \mathbin{+\!\!+} ys) \end{aligned}$$

does not hold unless $x = y$. Even then, we require an extra condition, namely that *cons* is monotonic under $collect^\circ \cdot leq \cdot collect$. This condition holds for the given definition of *collect*, among others.

Given this property of *collect*, we do have that both *new* and $ok\,w \cdot glue$ are monotonic on Q°, where

$$Q \quad = \quad R \cap (head^\circ \cdot head).$$

We leave the formal justification as an exercise. So all the conditions for binary thinning are in place, except for the choice of the connected preorder P. Unlike the case of the knapsack problem we cannot take $P = R$. The choice of P is a sensitive one because sorting with P should bring together paragraphs with the same first line, enabling *thinlist* Q to thin them to a single candidate. A logical choice is to weaken the equivalence relation $head^\circ \cdot head$ to a connected preorder, taking

$$P \quad = \quad head^\circ \cdot L \cdot head,$$

where L is some linear order on lines. Given context, we can take $L = prefix$, because this is a linear order on first lines of paragraphs of the same input. And it is easy to show that both *new* and *glue* are monotonic on P. However, all this is overkill because a much simpler choice of P suffices, namely, $P = \Pi$, the universal relation. Trivially, all functions are monotonic on Π. The reason why Π works is because we have

$$merge\,\Pi \quad = \quad cat,$$

and so the term g_1 in the implementation given below automatically brings together all partial solutions with the same first line.

With this choice of P the binary thinning theorem gives

$$paragraph\,w \quad = \quad minlist\,R \cdot (\!|thinlist\,Q \cdot cat \cdot \langle g_1, g_2\rangle \cdot listcp|\!),$$

where

$$\begin{aligned} g_1 &\quad = \quad list\,[wrap \cdot wrap, new] \\ g_2 &\quad = \quad filter\,(ok\,w) \cdot list\,[wrap \cdot wrap, glue]. \end{aligned}$$

For the functor $\mathsf{F}A = Word + (Word \times A)$ we have

$$listcp \;=\; listcp(\mathsf{F}) \;=\; wrap + cpr.$$

Hence rewriting g_1 and g_2 as coproducts, we obtain

$$paragraph\ w \quad = \quad minlist\ R \cdot (\!|start, thinlist\ Q \cdot cat \cdot \langle h_1, h_2 \rangle \cdot cpr|\!),$$

where

$$\begin{aligned} start &= wrap \cdot wrap \cdot wrap \\ h_1 &= list\ new \\ h_2 &= filter\,(ok\ w) \cdot list\ glue. \end{aligned}$$

The program

For efficiency, a partition $[x] +\!\!+\ xs$ is represented by the pair

$$([x] +\!\!+\ xs, (w - width\ x, waste\ w\ xs)).$$

Since $waste\ w\ [\,]$ is not defined, we will assume that it is some large negative quantity $-\infty$; then we have that the waste of a partition $(xs, (m, n))$ is $max\{m^2 + n, 0\}$.

The resulting program is shown below. Some additional input and output formatting has been added to make the program more useful: `words` divides a string into consecutive words, leaving out spaces and newline characters; `unwords` does the opposite, joining the words with single spaces; and `unlines` joins lists of lines with single newline characters. These formatting functions are provided in Gofer's standard prelude and are also defined in the Appendix:

```
> paragraph w = unpara . para w . words

> unpara      = unlines . list unwords . outl

> para w = minlist r . catallist (start w, thinlist q . step w)
> step w = cat . pair (list (new' w), filter ok . list glue') . cpr

> start w = wrap . pair (wrap . wrap, augment)
>           where augment = pair ((w-) . length, neginf)
> new' w  = cross (new, augment) . dupl
>           where augment = cross ((w-) . length, waste)
> glue'   = cross (glue, augment) . dupl
>           where augment = cross (reduce . swap, outr) . dupl
>                 reduce  = minus . cross (id, succ . length)
> new     = cons . cross (wrap, id)
> glue    = cons . cross (cons, id) . assocl . cross (id, split)
```

```
> r = leq . cross (waste . outr, waste . outr)
> p = eql . cross (outl . outr, outl . outr)
> q = meet (r,p)

> waste  = omax . plus . cross (sqr, id)
> omax   = cond (>= 0) (id, zero)
> sqr    = times . pair (id, id)
> ok     = (>= 0) . outl . outr
> neginf = const (-10000)
```

Exercises

8.23 Show $\mathit{list}^+\,(\mathit{fits}\ w)\cdot \mathit{partition} = (\![\mathit{wrap}\cdot \mathit{wrap}, \mathit{new}\cup(\mathit{ok}\ w\cdot \mathit{glue})]\!)$.

8.24 One possible choice for the function f in the definition of *waste* is $f = \mathit{sum}$. This leads to less pleasing paragraphs, but a greedy algorithm is possible provided we switch to snoc-lists. Derive this algorithm.

8.6 Bitonic tours

As a final application of thinning we solve a generalisation of the following problem, which is taken from (Cormen et al. 1990):

> The *euclidean traveling-salesman* problem is the problem of determining a shortest closed tour that connects a given set of n points in the plane. On the left in Figure 8.2 is the solution to a 7-point problem. The general problem is NP-complete, and its solution is therefore believed to require more than polynomial time.
>
> J.L. Bentley has suggested that we simplify the problem by restricting our attention to *bitonic tours*, that is, tours that start at the leftmost point, go strictly left to right to the rightmost point, and then go strictly right to left back to the starting point. On the right in Figure 8.2 is the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.
>
> Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate. (*Hint:* Scan right to left, maintaining optimal possibilities for the two parts of the tour.)

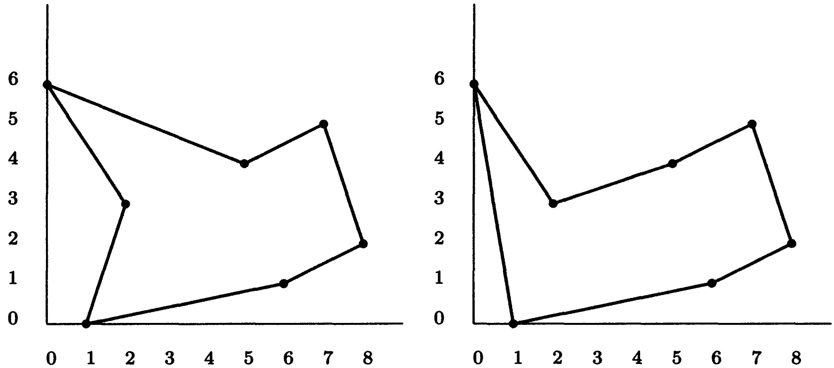

Figure 8.2: An optimal and an optimal bitonic tour.

We will solve a generalised version of the bitonic tours problem in which distances are not necessarily euclidean nor necessarily symmetric. We suppose only that with each ordered pair (a, b) of points (called *cities* below) is associated a travelling cost $tc\,(a, b)$, not necessarily positive nor necessarily equal to $tc\,(b, a)$. The final algorithm will take $O(n^2)$ time, where n is the length of the input, assuming that tc can be computed in constant time.

It does not make sense to talk about a bitonic tour of one city, so we will assume that the input is a list of at least two cities, the order of the cities in the list being relevant. We will take the hint in the formulation of the problem and build tours from right to left, but this is only because cons-lists are more efficient than snoc-lists in functional programming. Formally, all this means that we are dealing with the base functor

$$\mathsf{F}A \;=\; (City \times City) + (City \times A)$$

of cons-lists of length at least two.

We will describe tours, much as a travel agent would, by a pair of lists (x, y), where x represents the outward journey and y the return (reading from right to left). For example, the tour that proceeds directly from *New York* to *Rome* but visits *London* on its return is represented by the itinary

$$([New\ York, Rome], [New\ York, London, Rome]).$$

This is a different itinary to

$$([New\ York, London, Rome], [New\ York, Rome]),$$

because the travelling costs may depend on the direction of travel. As we have described them, both parts of the tour are subsequences of the input, and have lengths at least two.

Suppose the first example is extended to include *Los Angeles* as the new starting point. This gives rise to two extended tours:

$$([\mathit{Los\ Angeles}, \mathit{New\ York}, \mathit{Rome}], [\mathit{Los\ Angeles}, \mathit{London}, \mathit{Rome}])$$
$$([\mathit{Los\ Angeles}, \mathit{Rome}], [\mathit{Los\ Angeles}, \mathit{New\ York}, \mathit{London}, \mathit{Rome}]).$$

It is a requirement of a tour that no city should be visited twice, so *New York* has to be dropped from either the outward journey or the return.

With these assumptions, we can define *tour* by

$$\mathit{tour} \;=\; (\![\mathit{start}, \mathit{dropl} \cup \mathit{dropr}]\!),$$

where $\mathit{start}\,(a, b) = ([a, b], [a, b])$ and

$$\begin{aligned} \mathit{dropl}\,(a, ([b] +\!\!+\, x, y)) &= ([a] +\!\!+\, x, [a] +\!\!+\, y) \\ \mathit{dropr}\,(a, (x, [b] +\!\!+\, y)) &= ([a] +\!\!+\, x, [a] +\!\!+\, y). \end{aligned}$$

Each partial tour (x, y) maintains the property that the first elements of x and y are the same, as are the last elements.

The total cost of a tour is given by a function *cost* defined by

$$\mathit{cost}\,(x, y) \;=\; \mathit{outcost}\, x + \mathit{incost}\, y,$$

where

$$\begin{aligned} \mathit{outcost}\,[a_0, a_1, \ldots, a_n] &= (+j : 0 \le j < n : \mathit{tc}\,(a_j, a_{j+1})) \\ \mathit{incost}\,[a_0, a_1, \ldots, a_n] &= (+j : 0 < j \le n : \mathit{tc}\,(a_j, a_{j-1})). \end{aligned}$$

Our problem now is to find a function *mintour* that refines $\mathit{min}\, R \cdot \Lambda \mathit{tour}$, where $R = \mathit{cost}^\circ \cdot \mathit{leq} \cdot \mathit{cost}$.

Derivation

As usual, analysis of why $[\mathit{start}, \mathit{dropl} \cup \mathit{dropr}]$ is not monotonic on R° will help to suggest an appropriate Q for the thinning step. The monotonicity condition comes down to two inclusions:

$$\begin{aligned} \mathit{dropl} \cdot (\mathit{id} \times R^\circ) &\subseteq R^\circ \cdot \mathit{dropl} \\ \mathit{dropr} \cdot (\mathit{id} \times R^\circ) &\subseteq R^\circ \cdot \mathit{dropr}. \end{aligned}$$

To see what these conditions mean, observe that $cost\,(dropl\,(a,(x,y)))$ equals

$$cost\,(x,y) + tc\,(a, next\,x) - tc\,(head\,x, next\,x) + cost\,(head\,y, a),$$

where $next\,([a] \mathbin{+\!\!+} [b] \mathbin{+\!\!+} x) = b$. Dually, $cost\,(dropr\,(a,(x,y)))$ equals

$$cost\,(x,y) + tc\,(a, head\,x) - tc\,(next\,y, head\,y) + tc\,(next\,y, a).$$

Now, the first condition says that if $cost\,(x,y) \leq cost\,(u,v)$, then

$$\begin{array}{l} cost\,(x,y) + tc\,(a, next\,x) - tc\,(head\,x, next\,x) + cost\,(head\,y, a) \\ \leq cost\,(u,v) + tc\,(a, next\,u) - tc\,(head\,u, next\,u) + cost\,(head\,v, a). \end{array}$$

The second condition is similar. Neither holds under an arbitrary function $cost$ unless

$$(head\,x, head\,y) = (head\,u, head\,v) \;\wedge\; (next\,x, next\,y) = (next\,u, next\,v).$$

The first conjunct will hold whenever (x,y) and (u,v) are tours of the same input. It is now clear that we have to define Q by

$$Q \;=\; R \cap (next^2)^{\circ} \cdot next^2,$$

for then $dropl$ and $dropr$ are both monotonic under Q° (and Q).

All the conditions for the binary thinning theorem are in place, except for the choice of the connected preorder P. As in the paragraph problem, we cannot take $P = R$ because the monotonicity condition is not satisfied. And, also as in the paragraph problem, we can take $P = \Pi$. The reason is basically the same as before and Exercise 8.25 goes into details.

Since $merge\,\Pi = cat$, we can appeal to binary thinning and take

$$mintour \;=\; minlist\,R \cdot (\!|thinlist\,Q \cdot cat \cdot \langle g_1, g_2\rangle \cdot listcp|\!),$$

where $g_1 = list\,[start, dropl]$ and $g_2 = list\,[start, dropr]$. As before, we have $listcp = wrap + cpr$, so $mintour$ simplifies to

$$minlist\,R \cdot (\!|wrap \cdot start, thinlist\,Q \cdot cat \cdot \langle list\,dropl, list\,dropr\rangle \cdot cpr|\!).$$

The algorithm takes quadratic time because just two new tours are added to the list of partial solutions at each stage (see Exercise 8.25). If the list of partial solutions grows linearly and it takes linear time to generate the new tours, then the total time is quadratic in the length of the input.

The program

For efficiency a tour t is represented by the pair $(t, \mathit{cost}\ p)$. The Gofer program is parameterised by the function `tc`:

```
> mintour = minlist r . cata2list (wrap . start, thinlist q . step)
> step    = cat . pair (list dropl, list dropr) . cpr

> start (a,b)          = (([a,b],[a,b]), tc (a,b) + tc (b,a))
> dropl (a,((x,y),m)) = ((a:tail x, a:y), m + adjustl (a,x,y))
> dropr (a,((x,y),m)) = ((a:x, a:tail y), m + adjustr (a,x,y))

> adjustl (a, b:c:x, d:y) = tc (a,c) - tc (b,c) + tc (d,a)
> adjustr (a, b:x, d:e:y) = tc (a,b) - tc (e,d) + tc (e,a)

> r     = leq . cross (outr, outr)
> p     = eql . cross (next2, next2)
> q     = meet (r,p)
> next2 = cross (next, next) . outl
> next  = head . tail

> cata2list (f,g) [a,b] = f (a,b)
> cata2list (f,g) (a:x) = g (a, cata2list (f,g) x)
```

Exercises

8.25 Determine $\mathit{next}\,(\mathit{dropl}\,(a,(x,y)))$ and $\mathit{next}\,(\mathit{dropr}\,(a,(x,y)))$. Hence show by induction that the *next* values of the list of tours maintained after processing the input $[a_0, a_1, \ldots, a_n]$ are:

$$(a_n, a_1),\ (a_{n-1}, a_1),\ \ldots, (a_2, a_1),\ (a_1, a_2), \ldots, (a_1, a_n).$$

8.26 Consider the case where *tc* is symmetric, so the tour (y, x) is essentially the same as (x, y). Show how *dropl* and *dropr* can be modified to avoid generating the same tour twice. What is the resulting algorithm?

8.27 One basic assumption of the problem was that a city could not be visited both on the outward and inward part of the journey. Reformulate the problem to remove this restriction. What is the algorithm?

8.28 The other assumption was that each city should be visited at least once. Reformulate the problem to remove this restriction. What is the algorithm?

8.29 The longest upsequence problem is to compute $max\ R \cdot \Lambda(ordered \cdot subseq)$, where $R = length^{\circ} \cdot leq \cdot length$. Derive a thinning algorithm to solve this problem.

8.30 The rally driver's problem is described as follows: imagine a long stretch of road along which n gas stations are placed. At gas station $i\,(1 \leq i \leq n)$ is a quantity of fuel f_i. The distance from station i to station $i+1$ (or to the end of the road if $i = n$) is a known quantity d_i, where both fuel and distance are measured in terms of the same unit. Imagine that the rally driver is at the beginning of the road, with a quantity f_0 of fuel in the tank. Suppose also that the capacity of the fuel tank is some fixed quantity c. Assuming that the rally driver can get to the end of the road, devise a thinning algorithm for determining a minimum number of stops to pick up fuel. (*Hint*: model the problem using partitions.)

8.31 Solve the following exercise from (Denardo 1982) by a thinning algorithm: A long one-way street consists of m blocks of equal length. A bus runs 'uptown' from one end of the street to the other. A fixed number n of bus stops are to be located so as to minimise the total distance walked by the population. Assume that each person taking an uptown bus trip walks to the nearest bus stop, gets on the bus, rides, gets off at the stop nearest his or her destination, and walks the rest of the way. During the day, exactly B_j people from block j start uptown bus trips, and C_j complete uptown bus trips at block j. Write a program that finds an optimal location of bus stops.

Bibliographical remarks

The motivation of this chapter was to capture the essence of *sequential decision processes* as first introduced by Bellman (Bellman 1957), and rigorously defined by (Karp and Held 1967). In particular, Theorem 8.2 could be seen as a 'generic program' for sequential decision processes (De Moor 1995). In that paper it is indicated how the abstract relational expressions of Theorem 8.2 can actually be written as an executable computer program.

The relation passed as an argument to *thin* corresponds roughly to what other authors call a *dominance relation*. Dominance relations have received a lot of attention in the algorithm design literature (Eppstein, Galil, Giancarlo, and Italiano 1992; Galil and Giancarlo 1989; Hirschberg and Larmore 1987; Yao 1980, 1982). Most of this work is concerned with improving the time complexity of naive dynamic programming algorithms.

In programming methodology, our work is very much akin to that of (Smith and Lowry 1990; Smith 1991). Smith's notion of *problem reduction generators* is quite similar to the generic algorithm presented here in fact, but bears a closer resemblance to the results of the following chapter.

The idea of implementing dynamic programming algorithms through merging is well known in operations research. In the context of the 0/1 knapsack problem, it was first suggested by (Ahrens and Finke 1975). Recently, this method has been improved (through an extension of methods described in this book) to obtain a novel solution to the 0/1 knapsack problem that outperforms all others in practice (Ning 1997).

Dynamic Programming

We turn now to methods for solving the optimisation problem

$$min\, R \cdot \Lambda((\![S]\!) \cdot (\![T]\!)^\circ).$$

However, we will only consider the case where $S = h$, a function. This chapter discusses *dynamic programming* solutions, while Chapter 10 considers another class of greedy algorithms. In outline, dynamic programming is based on the observation that, for many problems, an optimal solution is composed of optimal solutions to subproblems, a property known as the *principle of optimality*.

If the principle of optimality is satisfied, then one can decompose the problem in all possible ways into subproblems, solve the subproblems recursively, and assemble an optimal solution from the partial results. This is the content of Theorem 9.1. Sometimes it is known that certain decompositions can never contribute to an optimum solution and can be discarded; this is the content of Theorem 9.2. In the extreme case, all but a single decomposition can be discarded, leading to a class of greedy algorithms to be studied in Chapter 10.

The sets of decompositions associated with different subproblems are usually not disjoint, so a naive approach to solving the subproblems recursively will involve repeating work. For this reason there is a second phase of dynamic programming in which the subproblems are solved more efficiently. There are two complementary schemes: *memoisation* and *tabulation*. (The terminology is standard, but tabulation has nothing to do with tabular allegories.)

The memoisation scheme is top-down; the computation follows that of the recursive program but solutions to subproblems are recorded and retrieved for subsequent use. Some functional languages provide a built-in memoisation facility as an optional extra. By contrast, the tabulation scheme is bottom-up; using an analysis of the dependencies between subproblems, the problems are solved in order of dependency, and stored in a specially constructed table to make subsequent retrieval easy. Although the dependency analysis is usually simple, the implementation of a tabulation scheme can be rather complicated to describe and justify. We will, however, give full details of tabulations for two of the applications described in this chapter.

9.1 Theory

As mentioned above, we consider only the case that $S = h$, a function. To save ink, define $H = (\![h]\!) \cdot (\![T]\!)^\circ$ in all that follows, where h and T are F-algebras. The basic theorem about dynamic programming is the following one.

Theorem 9.1 Let $M = \mathit{min}\, R \cdot \Lambda H$. If h is monotonic on R, then

$$(\mu X : \mathit{min}\, R \cdot \mathsf{P}(h \cdot \mathsf{F}X) \cdot \Lambda T^\circ) \quad \subseteq \quad M.$$

Proof. It follows from Knaster–Tarski that the conclusion holds if we can show that

$$\mathit{min}\, R \cdot \mathsf{P}(h \cdot \mathsf{F}M) \cdot \Lambda T^\circ \quad \subseteq \quad M. \tag{9.1}$$

Using the universal property of *min* we can rewrite (9.1) as two inclusions:

$$\mathit{min}\, R \cdot \mathsf{P}(h \cdot \mathsf{F}M) \cdot \Lambda T^\circ \quad \subseteq \quad H \tag{9.2}$$

$$\mathit{min}\, R \cdot \mathsf{P}(h \cdot \mathsf{F}M) \cdot \Lambda T^\circ \cdot H^\circ \quad \subseteq \quad R. \tag{9.3}$$

To prove (9.2) and (9.3) we will need the rule

$$\mathit{min}\, R \cdot \mathsf{P}X \quad \subseteq \quad (X \cdot \in) \cap ((R \cdot X)/\ni) \tag{9.4}$$

proved in Chapter 7.

For (9.2) we argue:

$$\begin{array}{cl}
& \mathit{min}\, R \cdot \mathsf{P}(h \cdot \mathsf{F}M) \cdot \Lambda T^\circ \\
\subseteq & \quad \{\text{since (9.4) gives } \mathit{min}\, R \cdot \mathsf{P}X \subseteq X \cdot \in\} \\
& h \cdot \mathsf{F}M \cdot \in \cdot \Lambda T^\circ \\
= & \quad \{\Lambda \text{ cancellation}\} \\
& h \cdot \mathsf{F}M \cdot T^\circ \\
\subseteq & \quad \{\text{definition of } M \text{ and universal property of } \mathit{min}\} \\
& h \cdot \mathsf{F}H \cdot T^\circ \\
= & \quad \{\text{definition of } H \text{ and hylomorphism theorem (Theorem 6.2)}\} \\
& H.
\end{array}$$

To prove (9.3) we argue:

$$
\begin{array}{ll}
 & \mathit{min}\, R \cdot \mathsf{P}(h \cdot \mathsf{F}M) \cdot \Lambda T^\circ \cdot H^\circ \\
\subseteq & \quad \{\text{since (9.4) gives } \mathit{min}\, R \cdot \mathsf{P}X \subseteq (R \cdot X)/\ni\} \\
 & ((R \cdot h \cdot \mathsf{F}M)/\ni) \cdot \Lambda T^\circ \cdot H^\circ \\
= & \quad \{\text{definition of } H \text{ and hylomorphism theorem}\} \\
 & ((R \cdot h \cdot \mathsf{F}M)/\ni) \cdot \Lambda T^\circ \cdot T \cdot \mathsf{F}H^\circ \cdot h^\circ \\
\subseteq & \quad \{\text{since } \Lambda X^\circ \cdot X \subseteq \ni\text{; division; functors}\} \\
 & R \cdot h \cdot \mathsf{F}(M \cdot H^\circ) \cdot h^\circ \\
\subseteq & \quad \{\text{definition of } M \text{ and universal property of } \mathit{min}\} \\
 & R \cdot h \cdot \mathsf{F}R \cdot h^\circ \\
\subseteq & \quad \{\text{assumption } h \cdot \mathsf{F}R \cdot h^\circ \subseteq R\} \\
 & R \cdot R \\
\subseteq & \quad \{\text{since } R \text{ is transitive}\} \\
 & R.
\end{array}
$$

□

Theorem 9.1 describes a recursive scheme in which the input is decomposed in all possible ways. However, with some problems we can tell that certain decompositions will never lead to better results than others. The basic theorem can be refined by bringing in a thinning step to eliminate unprofitable decompositions. This leads to the following version of dynamic programming; the proof follows the preceding one very closely, and we leave it as an exercise:

Theorem 9.2 Let $M = \mathit{min}\, R \cdot \Lambda H$. If h is monotonic on R and Q is a preorder satisfying $h \cdot \mathsf{F}H \cdot Q^\circ \subseteq R^\circ \cdot h \cdot \mathsf{F}H$, then

$$(\mu X : \mathit{min}\, R \cdot \mathsf{P}(h \cdot \mathsf{F}X) \cdot \mathit{thin}\, Q \cdot \Lambda T^\circ) \quad \subseteq \quad M.$$

Both theorems conclude that an optimal solution can be computed as a least fixed point of a certain equation. Theorem 6.3 says that the equation has a unique fixed point if $\mathit{member}\,(\mathsf{F}) \cdot T^\circ$ is an inductive relation. Furthermore, if ΛT° returns finite non-empty sets and R is a connected preorder, then the unique solution is entire. By suitably refining $\mathit{min}\, R$ and $\mathit{thin}\, Q \cdot \Lambda T^\circ$, we can then implement the solution as a recursive function.

Since Q is a relation on $\mathsf{F}A$ (for some A), and $\mathsf{F}A$ is often a coproduct, we can appeal to the following proposition to instantiate the conclusion of Theorem 9.2. The proof is left as an exercise.

Proposition 9.1 Suppose that V_1 and V_2 have disjoint ranges, that is, suppose that $V_1{}^\circ \cdot V_2 = \emptyset$. Then

$$min\, R \cdot \mathsf{P}[U_1, U_2] \cdot thin\,(Q_1 + Q_2) \cdot \Lambda[V_1, V_2]^\circ \quad = \quad (ran\ V_1 \rightarrow W_1,\ W_2),$$

where $W_i = min\, R \cdot \mathsf{P}\, U_i \cdot thin\ Q_i \cdot \Lambda\, V_i{}^\circ$ for $i = 1, 2$.

Checking the conditions

The two conditions of dynamic programming are:

$$\begin{array}{rcl} h \cdot \mathsf{F}R & \subseteq & R \cdot h \\ h \cdot \mathsf{F}H \cdot Q^\circ & \subseteq & R^\circ \cdot h \cdot \mathsf{F}H. \end{array}$$

To ease the task of checking these conditions, we can often appeal to one or other of the following results; the first could have been given in an earlier chapter.

Proposition 9.2 If for some functions *cost* and k we have

$$\begin{array}{rcl} R & = & cost^\circ \cdot leq \cdot cost \\ cost \cdot h & = & k \cdot \mathsf{F}\, cost \\ k \cdot \mathsf{F} leq & \subseteq & leq \cdot k, \end{array}$$

then $h \cdot \mathsf{F}R \subseteq R \cdot h$.

Proof. We argue:

$$\begin{array}{ll} & h \cdot \mathsf{F}R \subseteq R \cdot h \\ \equiv & \quad \{\text{definition of } R \text{ and shunting}\} \\ & cost \cdot h \cdot \mathsf{F}R \subseteq leq \cdot cost \cdot h \\ \equiv & \quad \{\text{assumption on } cost\} \\ & k \cdot \mathsf{F}(cost \cdot R) \subseteq leq \cdot k \cdot \mathsf{F}\, cost \\ \Leftarrow & \quad \{\text{since } cost \cdot R \subseteq leq \cdot cost\} \\ & k \cdot \mathsf{F}(leq \cdot cost) \subseteq leq \cdot k \cdot \mathsf{F}\, cost \\ \Leftarrow & \quad \{\text{functors}\} \\ & k \cdot \mathsf{F}\, leq \subseteq leq \cdot k \\ \Leftarrow & \quad \{\text{assumption that } k \text{ is monotonic on } leq\} \\ & true. \end{array}$$

□

The following result establishes a monotonicity in context property (see also Exercise 9.2).

Proposition 9.3 If for some functions *cost* and k we have

$$\begin{array}{rcl} R & = & cost^{\circ} \cdot leq \cdot cost \\ cost \cdot h & = & k \cdot \mathsf{F}\langle cost, H^{\circ}\rangle \\ k \cdot \mathsf{F}(leq \times id) & \subseteq & leq \cdot k, \end{array}$$

and if H° is simple, then $h \cdot \mathsf{F}(R \cap (H \cdot H^{\circ})) \subseteq R \cdot h$.

Proof. We argue:

$$\begin{array}{ll} & h \cdot \mathsf{F}(R \cap (H \cdot H^{\circ})) \subseteq R \cdot h \\ \equiv & \quad \{\text{definition of } R \text{ and shunting}\} \\ & cost \cdot h \cdot \mathsf{F}((cost^{\circ} \cdot leq \cdot cost) \cap (H \cdot H^{\circ})) \subseteq leq \cdot cost \cdot h \\ \equiv & \quad \{\text{products}\} \\ & cost \cdot h \cdot \mathsf{F}(\langle cost, H^{\circ}\rangle^{\circ} \cdot \langle leq \cdot cost, H^{\circ}\rangle) \subseteq leq \cdot cost \cdot h \\ \equiv & \quad \{\text{assumption on } cost\} \\ & k \cdot \mathsf{F}(\langle cost, H^{\circ}\rangle \cdot \langle cost, H^{\circ}\rangle^{\circ} \cdot \langle leq \cdot cost, H^{\circ}\rangle) \subseteq leq \cdot k \cdot \mathsf{F}\langle cost, H^{\circ}\rangle \\ \Leftarrow & \quad \{\text{since } H^{\circ} \text{ simple implies } \langle cost, H^{\circ}\rangle \text{ simple}\} \\ & k \cdot \mathsf{F}\langle leq \cdot cost, H^{\circ}\rangle \subseteq leq \cdot k \cdot \mathsf{F}\langle cost, H^{\circ}\rangle \\ \Leftarrow & \quad \{\text{products; functors}\} \\ & k \cdot \mathsf{F}(leq \times id) \subseteq leq \cdot k \\ \Leftarrow & \quad \{\text{assumption on } k\} \\ & true. \end{array}$$

□

In the next result we take F to be a bifunctor, writing $\mathsf{F}(id, X)$ rather than $\mathsf{F}X$.

Proposition 9.4 Suppose U and V are two preorders such that

$$h \cdot \mathsf{F}(U, R) \subseteq R \cdot h \quad \text{and} \quad H \cdot V^{\circ} \subseteq R^{\circ} \cdot H.$$

Then the conditions of Theorem 9.2 are satisfied by taking $Q = \mathsf{F}(U, V)$.

Proof. Monotonicity follows at once from the reflexivity of U. For the second part we argue as follows:

$$\begin{array}{cl} & h \cdot \mathsf{F}(id, H) \cdot Q^\circ \\ = & \quad \{\text{taking } Q = \mathsf{F}(U, V)\text{; converse; bifunctors}\} \\ & h \cdot \mathsf{F}(U^\circ, H \cdot V^\circ) \\ \subseteq & \quad \{\text{assumption on } V\} \\ & h \cdot \mathsf{F}(U^\circ, R^\circ \cdot H) \\ \subseteq & \quad \{\text{bifunctors}\} \\ & h \cdot \mathsf{F}(U^\circ, R^\circ) \cdot \mathsf{F}(id, H) \\ \subseteq & \quad \{\text{assumption on } h \text{ (taking converse and shunting)}\} \\ & R^\circ \cdot h \cdot \mathsf{F}(id, H). \end{array}$$

□

Exercises

9.1 Why is Theorem 9.1 a special case of Theorem 9.2?

9.2 The conditions of dynamic programming can be weakened by bringing in context. More precisely, it is sufficient to show that

$$\begin{array}{rcl} h \cdot \mathsf{F}(R \cap (H \cdot H^\circ)) & \subseteq & R \cdot h \\ h \cdot \mathsf{F}H \cdot (Q \cap (T^\circ \cdot T))^\circ & \subseteq & R^\circ \cdot h \cdot \mathsf{F}H. \end{array}$$

Prove this result.

9.3 Prove Theorem 9.2.

9.4 Prove that the thinning condition of Theorem 9.2 can be satisfied by taking $Q = \mathsf{F}(M^\circ \cdot R \cdot M)$. Why may this not be a good choice in practice?

9.5 This exercise deals with the proof of Proposition 9.1. Relations V_1 and V_2 have disjoint ranges if $\mathit{ran}\ V_2 \subseteq \sim \mathit{ran}\ V_1$, where $\sim$ is the complementation operator on coreflexives. Show that V_1 and V_2 have disjoint ranges if and only if

$$\mathit{ran}\ V_2 \quad = \quad \mathit{ran}\ V_2 \cdot \sim \mathit{ran}\ V_1.$$

Use this result to show that

$$\Lambda[V_1, V_2]^\circ \quad = \quad (\mathit{ran}\ V_1 \rightarrow \Lambda(\mathit{inl} \cdot V_1{}^\circ),\ \Lambda(\mathit{inr} \cdot V_2{}^\circ)).$$

Now show that

$$\mathit{thin}\,(Q_1 + Q_2) \cdot \mathsf{E}\mathit{inl} \quad = \quad \mathsf{E}\mathit{inl} \cdot \mathit{thin}\ Q_1$$

$$thin\,(Q_1 + Q_2) \cdot \mathsf{E}inr \quad = \quad \mathsf{E}inr \cdot thin\, Q_2.$$

Using these results, prove Proposition 9.1.

9.2 The string edit problem

In the string edit problem two strings x and y are given, and it is required to transform one string into the other by performing a sequence of editing operations. There are many possible choices for these operations, but for simplicity we assume that we are given just three: *copy*, *delete* and *insert*. Their meanings are as follows:

copy a	copy character a from x to y;
delete a	delete character a from x;
insert a	insert character a in y.

The point about these operations is that if we swap the roles of *delete* and *insert*, then we obtain a sequence transforming the target string back into the source. In fact, the operations contain enough information to construct both strings from scratch: we merely have to interpret *copy a* as meaning "append a to both strings"; *delete a* as "append a to the left string"; and *insert a* as "append a to the right string". Since there are many different edit sequences from which the two strings can be reconstituted, we ask for a *shortest* edit sequence.

To specify the problem formally we will use cons-lists for both strings and edit sequences; thus a string is an element of *list Char* and an edit sequence is an element of *list Op*, where

$$Op \quad ::= \quad cpy\ Char \mid del\ Char \mid ins\ Char.$$

The function $edit : (list\ Char \times list\ Char) \leftarrow list\ Op$ reconstitutes the two strings:

$$edit \quad = \quad (\!|base, step|\!),$$

where $base$ returns the pair $([\,],[\,])$ and

$$\begin{aligned} step\,(cpy\ a,(x,y)) \quad &= \quad ([a] \mathbin{+\!\!+} x, [a] \mathbin{+\!\!+} y) \\ step\,(del\ a,(x,y)) \quad &= \quad ([a] \mathbin{+\!\!+} x, y) \\ step\,(ins\ a,(x,y)) \quad &= \quad (x, [a] \mathbin{+\!\!+} y). \end{aligned}$$

The specification of string edit problem is to find a function *mle* (short for "minimum length edit") satisfying

$$mle \quad \subseteq \quad min\ R \cdot \Lambda edit^{\circ},$$

where $R = length^{\circ} \cdot leq \cdot length$.

Derivation

To apply basic dynamic programming we have to show that $\alpha = [nil, cons]$ is monotonic under R. But this is immediate from Proposition 9.2 using

$$length \quad = \quad (\![zero, succ \cdot outr]\!)$$

and the monotonicity of *succ* under *leq*.

For this problem we can go further and make good use of a thinning step. The intuition is that a *copy* operation, when available, leads to a shorter result than *delete* or *insert*. We therefore investigate whether we can find a preorder Q over the type $\mathsf{F}(Op, String \times String)$, where $\mathsf{F}(A, B) = 1 + (A \times B)$, satisfying

$$\alpha \cdot \mathsf{F}(id, edit^\circ) \cdot Q^\circ \quad \subseteq \quad R^\circ \cdot \alpha \cdot \mathsf{F}(id, edit^\circ).$$

From Proposition 9.4 we know that it is sufficient to take $Q = \mathsf{F}(U, V)$ for some preorders U and V satisfying the two conditions:

$$\alpha \cdot \mathsf{F}(U, R) \subseteq R \cdot \alpha \quad \text{and} \quad V \cdot edit \subseteq edit \cdot R.$$

Since $\alpha \cdot \mathsf{F}(\Pi, R) \subseteq R \cdot \alpha$ (exercise) we can always take $U = \Pi$. There is also an obvious choice for V: take $V = suffix \times suffix$. With this choice of V, the second condition can be broken down into two inclusions:

$$\begin{aligned} (id \times suffix) \cdot edit \quad &\subseteq \quad edit \cdot R \\ (suffix \times id) \cdot edit \quad &\subseteq \quad edit \cdot R. \end{aligned}$$

Since $suffix = tail^*$, it is sufficient to show that

$$\begin{aligned} (id \times tail) \cdot edit \quad &\subseteq \quad edit \cdot R \\ (tail \times id) \cdot edit \quad &\subseteq \quad edit \cdot R, \end{aligned}$$

because $A \cdot B \subseteq B \cdot C$ implies $A^* \cdot B \subseteq B \cdot C^*$. We give an informal proof of the first inclusion (a point-free proof is left as an exercise); the second one follows by a symmetrical argument. Suppose $edit\ es = (x, cons\ (b, y))$, and let e be the element of es that produces b. If $e = cpy\ b$, then replace e by $del\ b$ in es; if $e = ins\ b$, then remove e from es. The result is an edit sequence fs that is no longer than es and satisfies $edit\ fs = (x, y)$.

The result of this analysis is that a shortest edit sequence can be obtained by computing the least fixed point of the recursion equation

$$X \quad = \quad min\ R \cdot \mathsf{P}[nil, cons \cdot (id \times X)] \cdot thin\ Q \cdot \Lambda[base, step]^\circ,$$

where $Q = id + (U \times V)$, and U and V are given above.

Since *base* and *step* have disjoint ranges we can appeal to Proposition 9.1 and obtain

$$X \;=\; (\mathit{empty} \rightarrow \mathit{nil},\; \mathit{min}\, R \cdot \mathsf{P}(\mathit{cons} \cdot (\mathit{id} \times X)) \cdot \mathit{thin}\,(U \times V) \cdot \Lambda \mathit{step}^{\circ}),$$

where $\mathit{empty}\,(x, y)$ holds if both x and y are empty lists.

We can implement $\mathit{thin}\,(U \times V) \cdot \Lambda \mathit{step}^{\circ}$ as a list-valued function *unstep*, defined by

$$\begin{array}{lcl} \mathit{unstep}\,([a] \mathbin{+\!\!+} x, [\,]) & = & [(\mathit{del}\, a, (x, [\,]))] \\ \mathit{unstep}\,([\,], [b] \mathbin{+\!\!+} y) & = & [(\mathit{ins}\, b, ([\,], y))] \end{array}$$

and

$$\begin{array}{l} \mathit{unstep}\,([a] \mathbin{+\!\!+} x, [b] \mathbin{+\!\!+} y) = \\ \quad \begin{cases} [(\mathit{cpy}\, a, (x, y))], & \text{if } a = b \\ [(\mathit{del}\, a, (x, [b] \mathbin{+\!\!+} y)), (\mathit{ins}\, b, ([a] \mathbin{+\!\!+} x, y))], & \text{otherwise.} \end{cases} \end{array}$$

The relation $\mathit{min}\, R$ is implemented as the function $\mathit{minlist}\, R$ on lists. The result is that *mle* can be implemented by the program

$$\mathit{mle} \;=\; (\mathit{empty} \rightarrow \mathit{nil},\; \mathit{minlist}\, R \cdot \mathit{list}\,(\mathit{cons} \cdot (\mathit{id} \times \mathit{mle})) \cdot \mathit{unstep}).$$

The program terminates because the second components of $\mathit{unstep}\,(x, y)$ are pairs of strings whose combined length is strictly smaller than that of x and y.

The problem with this implementation is that the running time is an exponential function of the sizes of the two input strings. The reason is that the same subproblem is solved many times over. A suitably chosen tabulation scheme can bring this down to quadratic time, and this is the next part of the derivation.

Tabulation

The tabulation scheme for *mle* is motivated by the observation that in order to compute $\mathit{mle}\,(x, y)$ we also need to compute $\mathit{mle}\,(u, v)$ for all tails u of x and tails v of y. It is helpful to imagine these values arranged in columns: for example,

$$\begin{array}{lll} \mathit{mle}\,(a_1 a_2 a_3, b_1 b_2) & \mathit{mle}\,(a_1 a_2 a_3, b_2) & \mathit{mle}\,(a_1 a_2 a_3, [\,]) \\ \mathit{mle}\,(a_2 a_3, b_1 b_2) & \mathit{mle}\,(a_2 a_3, b_2) & \mathit{mle}\,(a_2 a_3, [\,]) \\ \mathit{mle}\,(a_3, b_1 b_2) & \mathit{mle}\,(a_3, b_2) & \mathit{mle}\,(a_3, [\,]) \\ \mathit{mle}\,([\,], b_1 b_2) & \mathit{mle}\,([\,], b_2) & \mathit{mle}\,([\,], [\,]). \end{array}$$

If we define the curried function

$$\mathit{column}\, x\, y \;=\; [\mathit{mle}\,(u, y) \mid u \leftarrow \mathit{tails}\, x],$$

then the rightmost column is $column\, x\, [\,]$ and the leftmost one is $column\, x\, y$. The topmost entry in the leftmost column is the required value $mle\,(x, y)$. We will build the columns one by one from right to left, using each column to construct the next one. Thus, we aim to express $column\, x$ as a cons-list catamorphism

$$column\, x \quad = \quad (\!|fstcol\, x, nextcol\, x|\!).$$

It is easy to check from the definition of mle that $fstcol = tails \cdot list\ del$. The function $nextcol$ is to satisfy the equation

$$column\, x\, ([b] +\!\!+\, y) \quad = \quad nextcol\, x\, (b, column\, x\, y).$$

The general idea is to implement $nextcol$ as a catamorphism, building the next column from bottom to top. From the recursive characterisation of mle we have

$$mle\,([a] +\!\!+\, u, [b] +\!\!+\, y) \quad = \quad \begin{cases} [cpy\ a] +\!\!+\, mle\,(u, y), & \text{if } a = b \\ [del\ a] +\!\!+\, mle\,(u, [b] +\!\!+\, y), & \text{if } m \leq n \\ [ins\ b] +\!\!+\, mle\,([a] +\!\!+\, u, y), & \text{otherwise,} \end{cases}$$

where m and n are the lengths of $mle\,(u, [b] +\!\!+\, y)$ and $mle\,([a] +\!\!+\, u, y)$ respectively. In terms of column entries the picture is

$\underline{column\, x\, ([b] +\!\!+\, y)}$	$\underline{column\, x\, y}$
.	.
.	.
$mle\,([a] +\!\!+\, u, [b] +\!\!+\, y)$	$mle\,([a] +\!\!+\, u, y)$
$mle\,(u, [b] +\!\!+\, y)$	$mle\,(u, y)$
.	.
.	.

Thus, each entry in the left column may depend on the one below it (if a delete is best), the one to the right (if an insert is best), and the one below that (if a copy is best).

In order to have all the necessary information available in the right place, the catamorphism for $nextcol$ is applied to the sequence

$$zip\,(x, zip\,(init\,(column\, x\, y), tail\,(column\, x\, y))).$$

The elements of x are needed for the case analysis in the definition of mle, and adjacent pairs of elements in $column\, x\, y$ are needed to determine the value of mle. The bottom element of $nextcol\, x\, (b, column\, x\, y)$ is obtained from the bottom element of $column\, x\, y$ as a special case. With this explanation, the definition of $nextcol$ is

$$nextcol\, x\, (b, us) \quad = \quad (\!|base\,(b, last\ us), step\ b|\!)\ xus,$$

where

$$\begin{aligned} xus &= zip\,(x, zip\,(init\; us, tail\; us)) \\ base\,(b, u) &= [[ins\; b] \mathbin{+\!\!+} u], \end{aligned}$$

and

$$step\; b\,((a,(u,v)), ws) = \begin{cases} [[cpy\; a] \mathbin{+\!\!+} v] \mathbin{+\!\!+} ws, & \text{if } a = b \\ [bmin\; R\,([del\; a] \mathbin{+\!\!+} w, [ins\; b] \mathbin{+\!\!+} u)] \mathbin{+\!\!+} ws, & \text{otherwise} \\ \text{where } w = head\; ws. \end{cases}$$

The program

The only change in the Gofer program is that an edit sequence v is represented by the pair $(v, length\; v)$ for efficiency. The program is

```
> data Op    = Cpy Char | Del Char | Ins Char
> mle (x,y) = outl (head (column x y))
> column x   = catalist (fstcol x, nextcol x)
> fstcol x   = zip (tails (list Del x), countdown (length x))

> nextcol x (b,us) = catalist ([ins b (last us)], step b) xus
>                    where xus = zip (x, zip (init us, tail us))

> step b ((a,(u,v)),ws)
>              = [cpy a v] ++ ws,                  if a == b
>              = [bmin r (del a w, ins b u)] ++ ws, otherwise
>                where r = leq . cross (outr, outr)
>                      w = head ws

> cpy b (ops,n) = (Cpy b : ops, n+1)
> del a (ops,n) = (Del a : ops, n+1)
> ins a (ops,n) = (Ins a : ops, n+1)

> countdown 0     = [0]
> countdown (n+1) = (n+1) : countdown n
```

Finally, let us show that this program takes quadratic time. The evaluation of $column\; x\; y$ requires q evaluations of *nextcol*, where q is the length of y, and the time to compute each evaluation of *nextcol* is $O(p)$ steps, where p is the length of x. Hence the time to construct $column\; x\; y$ is $O(p \times q)$ steps.

Exercises

9.6 Prove that $cons \cdot (\Pi \times R) \subseteq R \cdot cons$ where $R = length^{\circ} \cdot leq \cdot length$.

9.7 Prove formally that $(id \times tail) \cdot edit \subseteq edit \cdot R$, where $R = length^{\circ} \cdot leq \cdot length$.

9.3 Optimal bracketing

A standard application of dynamic programming is to the problem of building a minimum cost binary tree. The problem is often formulated as one of bracketing an expression $a_1 \oplus a_2 \oplus \cdots \oplus a_n$ in the best possible way. It is assumed that $\oplus$ is an associative operation, so the way in which the expression is bracketed does not affect its value. However, different bracketings may have different costs, and the objective is to find a bracketing of minimum cost. Specific instances of the bracketing problem are explored in the exercises.

The obvious choice of datatype to represent bracketings is a binary tree with values in the tips:

$$tree\ A \quad ::= \quad tip\ A \mid bin\,(tree\ A, tree\ A).$$

For example, the bracketing $(a_1 \oplus a_2) \oplus (a_3 \oplus a_4)$ is represented by the tree

$$bin\,(bin\,(tip\ a_1, tip\ a_2), bin\,(tip\ a_3, tip\ a_4)),$$

while the alternative bracketing $a_1 \oplus ((a_2 \oplus a_3) \oplus a_4)$ is represented by the tree

$$bin\,(tip\ a_1, bin\,(bin\,(tip\ a_2, tip\ a_3), tip\ a_4)).$$

A tree can be flattened by the function $flatten : list^+\ A \leftarrow tree\ A$ defined by

$$flatten \quad = \quad (\![wrap, cat]\!),$$

where $cat : list^+\ A \leftarrow (list^+\ A)^2$. This function produces the list of tip values in left to right order. Our problem, therefore, is to find a function mct (short for "minimum cost tree") satisfying

$$mct \quad \subseteq \quad min\ R \cdot \Lambda(\![wrap, cat]\!)^{\circ},$$

where $R = cost^{\circ} \cdot leq \cdot cost$.

The interesting part is the definition of $cost$. Here is the general scheme:

$$\begin{aligned} cost\,(tip\ a) &= 0 \\ cost\,(bin\,(x, y)) &= cb\,(size\ x, size\ y) + cost\ x + cost\ y \\ size\,(tip\ a) &= st\ a \\ size\,(bin\,(x, y)) &= sb\,(size\ x, size\ y). \end{aligned}$$

In words, the cost of building a single tip is zero, while the cost of building a node is some function cb of the sizes of the expressions associated with the two subtrees, plus the cost of building the two subtrees. The function $size$ (which, by the way, has no relation to the function that returns the number of elements in a tree) is a catamorphism on trees, where st gives the size of an atomic expression and sb the size of a compound expression in terms of its two subexpressions. Formally,

$$\langle cost, size \rangle \quad = \quad (\![opt, opb]\!),$$

where $opt = \langle zero, st \rangle$ and

$$opb\,((cx, sx), (cy, sy)) = (cb\,(sx, sy) + cx + cy, sb\,(sx, sy)).$$

We illustrate the abstract definition of $cost$ with one specific example. Consider the problem of computing $x_1 +\!\!+\, x_2 +\!\!+ \cdots +\!\!+\, x_n$ in the best possible way. If $+\!\!+$ is implemented on cons-lists, then the cost of evaluating $x +\!\!+\, y$ is proportional to the length of x, and the size of the result is the sum of the lengths of x and y. For this problem, $cb\,(m, n) = m$, $sb\,(m, n) = m + n$, and $st = length$. It turns out in this instance that the bracketing

$$x_1 +\!\!+ (x_2 +\!\!+ (\cdots +\!\!+ (x_{n-1} +\!\!+ x_n)))$$

is always optimal, which is one reason why $concat$ is defined as a catamorphism on cons-lists in functional programming.

Derivation

For this problem we have $h = [tip, bin]$ and $(\![T]\!) = flatten$, a function. There is no obvious criterion for preferring some decompositions over others, so the thinning step is omitted and we will aim for an application of Theorem 9.1. To establish the monotonicity condition we will need the assumption that the size of an expression is dependent only on its atomic constituents, not on the bracketing. This condition is satisfied if the function sb in the definition of $size$ is associative. It follows that $size = sz \cdot flatten$ for some function sz.

For the monotonicity condition we will use Proposition 9.3 and choose a function g satisfying

$$cost \cdot [tip, bin] \quad = \quad g \cdot (id + \langle cost, flatten \rangle^2) \tag{9.5}$$
$$g \cdot (id + (leq \times id)^2) \quad \subseteq \quad leq \cdot g. \tag{9.6}$$

We take

$$g \quad = \quad [zero, outl \cdot opb \cdot (id \times sz)^2],$$

where sz is the function introduced above.

For (9.5) we argue:

$$
\begin{array}{cl}
& g \cdot (id + \langle cost, flatten\rangle^2) \\
= & \quad \{\text{definition of } g\text{; coproducts and products}\} \\
& [zero, outl \cdot opb \cdot \langle cost, sz \cdot flatten\rangle^2] \\
= & \quad \{\text{assumption } sz \cdot flatten = size\} \\
& [zero, outl \cdot opb \cdot \langle cost, size\rangle^2] \\
= & \quad \{\text{since } \langle cost, size\rangle = (\!|opt, opb|\!)\} \\
& [zero, outl \cdot \langle cost, size\rangle \cdot bin] \\
= & \quad \{\text{since } cost \cdot tip = zero\text{; products}\} \\
& [cost \cdot tip, cost \cdot bin] \\
= & \quad \{\text{products}\} \\
& cost \cdot [tip, bin].
\end{array}
$$

For (9.6) we argue:

$$
\begin{array}{cl}
& g \cdot (id + (leq \times id)^2) \\
= & \quad \{\text{definition of } g\} \\
& [zero, outl \cdot opb \cdot (leq \times sz)^2] \\
\subseteq & \quad \{\text{definition of } opb \text{ and } + \text{ monotonic}\} \\
& [zero, leq \cdot outl \cdot opb \cdot (id \times sz)^2] \\
\subseteq & \quad \{leq \text{ reflexive}\} \\
& leq \cdot [zero, outl \cdot opb \cdot (id \times sz)^2] \\
= & \quad \{\text{definition of } g\} \\
& leq \cdot g.
\end{array}
$$

The dynamic programming theorem is therefore applicable and says that we can compute a minimum cost tree by computing the least fixed point of the recursion equation

$$X \quad = \quad min\, R \cdot \mathsf{P}[tip, bin \cdot (X \times X)] \cdot \Lambda[wrap, cat]^\circ.$$

Since *wrap* and *cat* have disjoint ranges appeal to Proposition 9.1 gives

$$X \quad = \quad (single \to tip \cdot wrap^\circ, min\, R \cdot \mathsf{P}(bin \cdot (X \times X)) \cdot \Lambda cat^\circ),$$

where $single\, x$ holds if x is a singleton list. The recursion can be implemented by representing Λcat° as the function *splits*, where

$$splits \quad = \quad zip \cdot \langle inits^+, tails^+\rangle,$$

and $inits^+$ and $tails^+$ return the lists of proper initial and tail segments of a list;

$inits^+$ is an implementation of $\Lambda init^+$, where $init^+$ is the transitive closure of *init* and describes the proper prefix relation; dually, $tails^+$ is an implementation of $\Lambda tail^+$.

Then we can implement *mct* by the recursive program:

$$mct \;=\; (single \rightarrow tip \cdot head, minlist\ R \cdot list\,(bin \cdot (mct \times mct)) \cdot splits).$$

The program terminates because the recursive calls are on shorter arguments: if (y, z) is an element of $splits\ x$, then both y and z are shorter than x. As in the case of the string editing problem, multiple evaluations of the same subproblem mean that the program has an exponential running time, so, once again, we need a tabulation scheme.

Tabulation

In order to compute $mct\ x$ we also need to compute $mct\ y$ for every non-empty segment y of x. It is helpful to picture these values as a two-dimensional array:

$$\begin{array}{llll}
mct\,(a_1) & & & \\
mct\,(a_1 a_2) & mct\,(a_2) & & \\
mct\,(a_1 a_2 a_3) & mct\,(a_2 a_3) & mct\,(a_3) & \\
mct\,(a_1 a_2 a_3 a_4) & mct\,(a_2 a_3 a_4) & mct\,(a_3 a_4) & mct\,(a_4).
\end{array}$$

The object is to compute the bottom entry of the leftmost column. We will represent the array as a list of rows, although we will also need to consider individual columns. Hence we define

$$\begin{array}{rcl}
array & = & list\ row \cdot inits \\
row & = & list\ mct \cdot tails \\
col & = & list\ mct \cdot inits.
\end{array}$$

The functions *inits* and *tails* both have type $list^*(list^+\ A) \leftarrow list^+\ A$; the function *inits* returns the list of non-empty initial segment in increasing order of length, and *tails* the tail segments in decreasing order of length.

In order to tackle the main calculation, which is to show how to compute *array*, we will need various subsidiary identities, so let us begin by expressing *mct* in terms of *row* and *col*. For the recursive case we can argue:

$$\begin{array}{ll}
 & mct \\
= & \quad \{\text{recursive case of } mct \text{ and definition of } splits\} \\
 & minlist\ R \cdot list\,(bin \cdot (mct \times mct)) \cdot zip \cdot \langle inits^+, tails^+ \rangle
\end{array}$$

$$\begin{array}{cl}
= & \{\text{since } \mathit{list}\,(f \times g) \cdot \mathit{zip} = \mathit{zip} \cdot (\mathit{list}\,f \times \mathit{list}\,g)\} \\
& \mathit{minlist}\,R \cdot \mathit{list}\,\mathit{bin} \cdot \mathit{zip} \cdot \langle \mathit{list}\,\mathit{mct} \cdot \mathit{inits}^+, \mathit{list}\,\mathit{mct} \cdot \mathit{tails}^+ \rangle \\
= & \{\text{introducing } \mathit{mix} = \mathit{minlist}\,R \cdot \mathit{list}\,\mathit{bin} \cdot \mathit{zip}\} \\
& \mathit{mix} \cdot \langle \mathit{list}\,\mathit{mct} \cdot \mathit{inits}^+, \mathit{list}\,\mathit{mct} \cdot \mathit{tails}^+ \rangle \\
= & \{\text{since } \mathit{inits}^+ = \mathit{inits} \cdot \mathit{init} \text{ and } \mathit{tails}^+ = \mathit{tails} \cdot \mathit{tail}\} \\
& \mathit{mix} \cdot \langle \mathit{list}\,\mathit{mct} \cdot \mathit{inits} \cdot \mathit{init}, \mathit{list}\,\mathit{mct} \cdot \mathit{tails} \cdot \mathit{tail} \rangle \\
= & \{\text{definition of } \mathit{row} \text{ and } \mathit{col}\} \\
& \mathit{mix} \cdot \langle \mathit{col} \cdot \mathit{init}, \mathit{row} \cdot \mathit{tail} \rangle.
\end{array}$$

Hence

$$\begin{array}{rcl}
\mathit{mct} & = & (\mathit{single} \rightarrow \mathit{tip} \cdot \mathit{head}, \mathit{mix} \cdot \langle \mathit{col} \cdot \mathit{init}, \mathit{row} \cdot \mathit{tail} \rangle) \\
\mathit{mix} & = & \mathit{minlist}\,R \cdot \mathit{list}\,\mathit{bin} \cdot \mathit{zip}.
\end{array} \tag{9.7}$$

Next, let us express *col* in terms of *row* and *col*. For the recursive case, we argue:

$$\begin{array}{cl}
& \mathit{col} \\
= & \{\text{definition of } \mathit{col}\} \\
& \mathit{list}\,\mathit{mct} \cdot \mathit{inits} \\
= & \{\text{since } \mathit{inits} = \mathit{snoc} \cdot \langle \mathit{inits} \cdot \mathit{init}, \mathit{id} \rangle \text{ on non-singletons}\} \\
& \mathit{list}\,\mathit{mct} \cdot \mathit{snoc} \cdot \langle \mathit{inits} \cdot \mathit{init}, \mathit{id} \rangle \\
= & \{\text{since } \mathit{list}\,f \cdot \mathit{snoc} = \mathit{snoc} \cdot (\mathit{list}\,f \times f)\} \\
& \mathit{snoc} \cdot \langle \mathit{col} \cdot \mathit{init}, \mathit{mct} \rangle \\
= & \{(9.7) \text{ on non-singletons}\} \\
& \mathit{snoc} \cdot \langle \mathit{col} \cdot \mathit{init}, \mathit{mix} \cdot \langle \mathit{col} \cdot \mathit{init}, \mathit{row} \cdot \mathit{tail} \rangle \rangle \\
= & \{\text{introducing } \mathit{next} = \mathit{snoc} \cdot \langle \mathit{outl}, \mathit{mix} \rangle\} \\
& \mathit{next} \cdot \langle \mathit{col} \cdot \mathit{init}, \mathit{row} \cdot \mathit{tail} \rangle.
\end{array}$$

Hence

$$\begin{array}{rcl}
\mathit{col} & = & (\mathit{single} \rightarrow \mathit{wrap} \cdot \mathit{tip} \cdot \mathit{head}, \mathit{next} \cdot \langle \mathit{col} \cdot \mathit{init}, \mathit{row} \cdot \mathit{tail} \rangle) \\
\mathit{next} & = & \mathit{snoc} \cdot \langle \mathit{outl}, \mathit{mix} \rangle.
\end{array} \tag{9.8}$$

Equation (9.8) can be used to justify the implementation of *col* as a loop (see Exercise 9.13):

$$\mathit{col} \;=\; \mathit{loop}\,\mathit{next} \cdot \langle \mathit{wrap} \cdot \mathit{tip} \cdot \mathit{head}, \mathit{list}\,\mathit{row} \cdot \mathit{inits} \cdot \mathit{tail} \rangle.$$

Below, we will need this equation in the equivalent form:

$$\begin{array}{rcl}
\mathit{col} \cdot \mathit{cons} & = & \mathit{process} \cdot (\mathit{id} \times \mathit{array}) \\
\mathit{process} & = & \mathit{loop}\,\mathit{next} \cdot ((\mathit{wrap} \cdot \mathit{tip}) \times \mathit{id}).
\end{array} \tag{9.9}$$

As a final preparatory step, we express *row* in terms of *mct* and *col*. For the recursive case we can argue:

$$\begin{array}{ll} & \mathit{row} \\ = & \quad \{\text{definition of } \mathit{row}\} \\ & \mathit{list\ mct} \cdot \mathit{tails} \\ = & \quad \{\text{since } \mathit{tails} = \mathit{cons} \cdot \langle \mathit{id}, \mathit{tails} \cdot \mathit{tail} \rangle \text{ on non-singletons}\} \\ & \mathit{list\ mct} \cdot \mathit{cons} \cdot \langle \mathit{id}, \mathit{tails} \cdot \mathit{tail} \rangle \\ = & \quad \{\text{since } \mathit{list}\, f \cdot \mathit{cons} = \mathit{cons} \cdot (f \times \mathit{list}\, f)\} \\ & \mathit{cons} \cdot \langle \mathit{mct}, \mathit{row} \cdot \mathit{tail} \rangle. \end{array}$$

Hence

$$\mathit{row} \quad = \quad (\mathit{single} \rightarrow \mathit{wrap} \cdot \mathit{tip} \cdot \mathit{head}, \mathit{cons} \cdot \langle \mathit{mct}, \mathit{row} \cdot \mathit{tail} \rangle) \tag{9.10}$$

Now for the main calculation. We will compute *array* as a catamorphism on cons-lists, building columns from right to left, and then using the column entries to extend each row. Hence we want

$$\mathit{array} \quad = \quad (\![\mathit{fstcol}, \mathit{addcol}]\!),$$

for appropriate functions *fstcol* and *addcol*. It is easy to check that

$$\mathit{fstcol} \quad = \quad \mathit{wrap} \cdot \mathit{wrap} \cdot \mathit{tip},$$

so the problem is to compute *addcol*. We reason:

$$\begin{array}{ll} & \mathit{array} \cdot \mathit{cons} \\ = & \quad \{\text{definition of } \mathit{array}\} \\ & \mathit{list\ row} \cdot \mathit{inits} \cdot \mathit{cons} \\ = & \quad \{\text{since } \mathit{inits} \cdot \mathit{cons} = \mathit{cons} \cdot \langle \mathit{wrap} \cdot \mathit{outl}, \mathit{tail} \cdot \mathit{inits} \cdot \mathit{cons} \rangle\} \\ & \mathit{list\ row} \cdot \mathit{cons} \cdot \langle \mathit{wrap} \cdot \mathit{outl}, \mathit{tail} \cdot \mathit{inits} \cdot \mathit{cons} \rangle \\ = & \quad \{\text{since } \mathit{list}\, f \cdot \mathit{cons} = \mathit{cons} \cdot (f \times \mathit{list}\, f)\} \\ & \mathit{cons} \cdot \langle \mathit{row} \cdot \mathit{wrap} \cdot \mathit{outl}, \mathit{list\ row} \cdot \mathit{tail} \cdot \mathit{inits} \cdot \mathit{cons} \rangle \\ = & \quad \{(9.10)\} \\ & \mathit{cons} \cdot \langle \mathit{wrap} \cdot \mathit{tip} \cdot \mathit{outl}, \mathit{list}\,(\mathit{cons} \cdot \langle \mathit{mct}, \mathit{row} \cdot \mathit{tail} \rangle) \cdot \mathit{tail} \cdot \mathit{inits} \cdot \mathit{cons} \rangle. \end{array}$$

We continue by simplifying the second term, abbreviating $\mathit{tail} \cdot \mathit{inits} \cdot \mathit{cons}$ by *tic*:

$$\begin{array}{ll} & \mathit{list}\,(\mathit{cons} \cdot \langle \mathit{mct}, \mathit{row} \cdot \mathit{tail} \rangle) \cdot \mathit{tic} \\ = & \quad \{\text{since } \mathit{list}\,\langle f, g \rangle = \mathit{zip} \cdot \langle \mathit{list}\, f, \mathit{list}\, g \rangle\} \\ & \mathit{list\ cons} \cdot \mathit{zip} \cdot \langle \mathit{list\ mct}, \mathit{list}\,(\mathit{row} \cdot \mathit{tail}) \rangle \cdot \mathit{tic} \end{array}$$

$$
\begin{aligned}
&= \quad \{\text{products}\} \\
&\quad \mathit{list\,cons} \cdot \mathit{zip} \cdot \langle \mathit{list\,mct} \cdot \mathit{tic}, \mathit{list}\,(\mathit{row} \cdot \mathit{tail}) \cdot \mathit{tic} \rangle \\
&= \quad \{\text{since } \mathit{list\,f} \cdot \mathit{tail} = \mathit{tail} \cdot \mathit{list\,f}\text{; definition of } \mathit{col}\} \\
&\quad \mathit{list\,cons} \cdot \mathit{zip} \cdot \langle \mathit{tail} \cdot \mathit{col} \cdot \mathit{cons}, \mathit{list}\,(\mathit{row} \cdot \mathit{tail}) \cdot \mathit{tic} \rangle \\
&= \quad \{\text{since } \mathit{list\,tail} \cdot \mathit{tic} = \mathit{inits} \cdot \mathit{outr}\text{; definition of } \mathit{array}\} \\
&\quad \mathit{list\,cons} \cdot \mathit{zip} \cdot \langle \mathit{tail} \cdot \mathit{col} \cdot \mathit{cons}, \mathit{array} \cdot \mathit{outr} \rangle \\
&= \quad \{(9.9)\} \\
&\quad \mathit{list\,cons} \cdot \mathit{zip} \cdot \langle \mathit{tail} \cdot \mathit{process} \cdot (\mathit{id} \times \mathit{array}), \mathit{array} \cdot \mathit{outr} \rangle \\
&= \quad \{\text{products}\} \\
&\quad \mathit{list\,cons} \cdot \mathit{zip} \cdot \langle \mathit{tail} \cdot \mathit{process}, \mathit{outr} \rangle \cdot (\mathit{id} \times \mathit{array}).
\end{aligned}
$$

Summarising, we have shown that $\mathit{array} \cdot \mathit{cons} = \mathit{addcol} \cdot (\mathit{id} \times \mathit{array})$, where

$$
\begin{aligned}
\mathit{addcol} &= \mathit{cons} \cdot \langle \mathit{wrap} \cdot \mathit{tip} \cdot \mathit{outl}, \mathit{step} \rangle \\
\mathit{step} &= \mathit{list\,cons} \cdot \mathit{zip} \cdot \langle \mathit{tail} \cdot \mathit{process}, \mathit{outr} \rangle.
\end{aligned}
$$

The program

The following Gofer program follows the above scheme, except that we label the trees with cost and size information. More precisely, the tree $\mathit{bin}\,(x, y)$ is represented by $\mathit{bin}\,(c, s)\,(x, y)$, where $c = \mathit{cost}\,(\mathit{bin}\,(x, y))$ and $s = \mathit{size}\,(\mathit{bin}\,(x, y))$:

```
> data Tree a = Tip a | Bin (Int,a) (Tree a, Tree a)

> mct     = head . last . array
> array   = catallist (fstcol, addcol)
> fstcol  = wrap . wrap . tip
> addcol  = cons . pair (wrap . tip . outl, step)
> step    = list cons . zip . pair (tail . process, outr)
> process = loop next . cross (wrap . tip, id)
> next    = snoc . pair (outl, minlist r . list bin . zip)
>           where r = leq . cross (cost, cost)

> cost (Tip a)       = 0
> cost (Bin (c,s) ts) = c

> size (Tip a)       = a
> size (Bin (c,s) ts) = s
```

```
> tip       = Tip
> bin (x,y) = Bin (c,s) (x,y)
>             where c = cb (size x, size y) + cost x + cost y
>                   s = sb (size x, size y)
```

Finally, let us estimate the running time of the program. To build an $(n \times n)$ array, the operation *addcol* is performed $n-1$ times. For execution of *addcol* on an array of size $(m \times m)$, the operation *step* takes $O(m^2)$ steps since *next* is executed m times and takes $O(m)$ steps. So the total is $O(n^3)$ steps.

Exercises

9.8 Consider the problem of computing the sum $x_1 + x_2 + \cdots + x_n$ in the most efficient manner, where each x_j is a decimal numeral. What are the functions *cost* and *size* for this bracketing problem?

9.9 Same questions as in the preceding exercise, but for the problem of computing the product $x_1 \times x_2 \times \cdots \times x_n$.

9.10 Same questions, but for *matrix* multiplication in which we want to compute $M_1 \times M_2 \times \cdots M_n$, where M_j is an (r_{j-1}, r_j) matrix.

9.11 Prove the claim that *concat* is best evaluated in terms of a catamorphism on cons-lists.

9.12 Show that if h is associative, then

$$(\!|g, h|\!) \cdot (\!|wrap, cat|\!) \quad = \quad (\!|g, h|\!),$$

where the catamorphism $(\!|g, h|\!)$ on the left is over non-empty lists, and $(\!|g, h|\!)$ on the right is over trees.

9.13 The standard function $loop\, f$ is defined in the Appendix by the equations

$$\begin{aligned} loop\, f \cdot (id \times nil) &= outl \\ loop\, f \cdot (id \times cons) &= loop\, f \cdot (f \times id) \cdot assocl. \end{aligned}$$

An equivalent characterisation of $loop\, f$ in terms of snoc-lists is:

$$\begin{aligned} loop\, f \cdot (id \times nil) &= outl \\ loop\, f \cdot (id \times snoc) &= f \cdot (loop\, f \times id) \cdot assocl. \end{aligned}$$

Using this characterisation, prove that

$$k \quad = \quad loop\, f \cdot (g \times list\, h \cdot inits)$$

if and only if

$$\begin{aligned} k \cdot (id \times nil) &= g \cdot outl \\ k \cdot (id \times snoc) &= f \cdot \langle k \cdot (id \times outl), h \cdot snoc \cdot outr \rangle. \end{aligned}$$

Hence prove (9.9).

9.14 The optimal bracketing problem can be phrased, like the knapsack problem, in terms of catamorphisms. Using the converse function theorem, express $\mathit{flatten}^{\circ}$ as a catamorphism, and hence find a thinning algorithm for the problem. (This is a research problem.)

9.15 Explore the variation of the bracketing problem in which $\oplus$ is assumed to be commutative as well as associative. This gives us the freedom to choose an optimal bracketing from among all possible permutations of the input $[a_1, a_2, \ldots, a_n]$.

9.4 Data compression

In the method of data compression by textual substitution the data to be compressed is a string of characters. The compressed data is an element of *list Code*, where an element of *Code* is either a character or a pointer to a substring of the part of the string already processed:

$$Code \quad ::= \quad sym\ Char \mid ptr\,(String, String^{+}).$$

A pointer is defined as a pair of strings (but see below), the idea being that the second string identifies the non-empty portion of the input concerned, while the first indicates where it is to be found. We make this idea precise by describing the process of decoding a code sequence.

We will need to use snoc-lists, so for this section suppose that

$$\begin{aligned} list\ A \quad &::= \quad nil \mid snoc\,(list\ A, A) \\ list^{+}\ A \quad &::= \quad wrap\ A \mid snoc\,(list^{+}\ A, A). \end{aligned}$$

In particular, $String = list\ Char$ and $String^{+} = list^{+}\ Char$. The partial function $decode : String \leftarrow list\ Code$ is defined as the catamorphism

$$decode \quad = \quad (\![nil, extend]\!),$$

where

$$\begin{aligned} extend\,(x, sym\ a) \quad &= \quad x +\!\!+ [a] \\ extend\,(x, ptr\,(y, z)) \quad &= \quad x +\!\!+ z, \quad \text{provided } (y +\!\!+ z)\ init^{+}\,(x +\!\!+ z). \end{aligned}$$

The relation $init^+$ is the transitive closure of *init* and describes the proper prefix relation. Note in the second equation that it is not required that $y +\!\!+ z$ be a prefix of x; in particular, we have

$$extend\,(\text{“aba”}, ptr\,(\text{“a”}, \text{“bab”})) \quad = \quad \text{“ababab”}.$$

The function *decode* is partial – if the very first code element is a pointer, then *decode* is undefined since there is no y for which $y +\!\!+ z$ is a proper prefix of z. Note also that the range of *decode* is the set of all possible strings, so all strings can be encoded.

We have chosen to define pointers as pairs of strings, but the success of data compression in practice results from representing each pointer (y, z) simply by the lengths of y and z. For this new representation, the decoding of a pointer is given by

$$extend\,(x, ptr\,(m, n)) \quad = \quad x \otimes (m, n),$$

where the operator $\otimes$ is defined recursively:

$$\begin{aligned} x \otimes (m, 0) \quad &= \quad x \\ x \otimes (m, n+1) \quad &= \quad (x +\!\!+ [x_m]) \otimes (m+1, n). \end{aligned}$$

Here, x_m is the mth element of x (counting from 0). This change of representation yields a compact representation of strings. For instance,

$$decode\,[\text{‘a’}, (0, 9)] \quad = \quad \text{“aaaaaaaaaa”}.$$

A slightly more involved example is

$$decode\,[\text{‘a’}, \text{‘a’}, \text{‘b’}, (1, 3), \text{‘c’}, (1, 2)] \quad = \quad \text{“aababacab”}.$$

Bearing the new representation of pointers in mind, we define the size of a code sequence by

$$size \quad = \quad (\![zero, plus \cdot [id \times c, id \times p] \cdot distr]\!),$$

where c and p are given constant functions returning the amount of space to store symbols and pointers. Typically, symbols require one byte, while pointers require four bytes (three bytes for the first number, and one byte for the second). Both c and p are determined by the implementation of the algorithm on a particular computer.

The function *size* induces a preorder $R = size^\circ \cdot leq \cdot size$, so our problem is to compute a function *encode* satisfying

$$encode \quad \subseteq \quad min\, R \cdot \Lambda\, decode^\circ.$$

Derivation

The monotonicity condition is easy to verify, so the basic form of dynamic programming is applicable. But we can do better with a suitable thinning step. For general c and p it is not possible to determine at each stage whether it is better to pick a symbol or a pointer, assuming that both choices are possible. On the other hand, it is possible to choose between pointers: a pointer (y, z) should be better than (y', z') whenever z is longer than z' because a longer portion of the input will then be consumed. More precisely, suppose

$$w = \mathit{extend}\,(x, \mathit{ptr}\,(y, z)) \quad \text{and} \quad w = \mathit{extend}\,(x', \mathit{ptr}\,(y', z')),$$

so $w = x \mathbin{+\!\!+} z = x' \mathbin{+\!\!+} z'$. Now, z is longer than z' if and only if z' is a suffix of z. Equivalently, z is longer than z' if and only if x is a prefix of x'.

This reasoning suggests one possible choice for the thinning relation Q: take

$$Q \quad = \quad \mathsf{F}(\Pi + \Pi, \mathit{prefix}),$$

where the first Π is the universal relation on symbols, and the second Π is the universal relation on pointers. The functor F is given by

$$\mathsf{F}(\mathit{Code}, \mathit{String}) \quad = \quad \mathit{id} + (\mathit{String} \times \mathit{Code}).$$

By Proposition 9.4 we have to check that

$$\alpha \cdot \mathsf{F}(\Pi + \Pi, R) \subseteq R \cdot \alpha \quad \text{and} \quad \mathit{prefix} \cdot \mathit{decode} \subseteq \mathit{decode} \cdot R.$$

The first condition is routine using the fact that the sizes of symbols and pointers are constants (i.e. $[c, d] \cdot (\Pi + \Pi) = [c, d]$), and we leave details as an exercise. The second condition follows if we can show

$$\mathit{init} \cdot \mathit{decode} \quad \subseteq \quad \mathit{decode} \cdot R.$$

We give an informal proof. Suppose $\mathit{decode}\ \mathit{cs} = \mathit{snoc}\,(x, a)$; either cs ends with the code element $\mathit{sym}\ a$, in which case drop it from cs, or it ends with the code element $\mathit{ptr}\,(y, z \mathbin{+\!\!+} [a])$ for some y and z; in the second case, replace it by $\mathit{ptr}\,(y, z)$ if $z \neq [\,]$, or drop it if $z = [\,]$. The result is a new code sequence that decodes to x, and which has cost no greater than cs.

The dynamic programming theorem states that the data compression problem can be solved by computing the least fixed point of the equation

$$X \quad = \quad \mathit{min}\ R \cdot \mathsf{P}[\mathit{nil}, \mathit{snoc} \cdot (X \times \mathit{id})] \cdot \mathit{thin}\ Q \cdot \Lambda[\mathit{nil}, \mathit{extend}]^{\circ},$$

where $Q = \mathit{id} + (U \times V)$ and $U = \mathit{prefix}$ and $V = \Pi + \Pi$. Since nil and extend have disjoint ranges, we can appeal to Proposition 9.1 and obtain

$$X \quad = \quad (\mathit{null} \rightarrow \mathit{nil},\ \mathit{min}\ R \cdot \mathsf{P}(\mathit{snoc} \cdot (X \times \mathit{id})) \cdot \mathit{thin}\,(U \times V) \cdot \Lambda \mathit{extend}^{\circ}).$$

The final task is to implement $thin\,(U \times V) \cdot \Lambda extend^{\circ}$. Since

$$\begin{aligned}&(\Lambda extend^{\circ})\,(w +\!\!+ [a]) = \\ &\qquad \{(w, sym\,a)\} \cup \{(x, ptr\,(y,z)) \mid x +\!\!+ z = w +\!\!+ [a] \;\wedge\; (y +\!\!+ z)\,prefix\,w\},\end{aligned}$$

we can define lrt (short for "longest repeated tail") by

$$lrt\,w \quad = \quad min\,(U \times V)\,\{(x,(y,z)) \mid x +\!\!+ z = w \;\wedge\; (y +\!\!+ z)\,init^{+}\,w\},$$

and so implement $thin\,(U \times V) \cdot \Lambda extend^{\circ}$ by a function $reduce$ defined by

$$reduce\,(w +\!\!+ [a]) \quad = \quad \begin{cases} [(w, sym\,a), (x, ptr\,(y,z))], & \text{if } z \neq [\,] \\ [(w, sym\,a)], & \text{otherwise} \\ \text{where } (x,(y,z)) = lrt\,(w +\!\!+ [a]). \end{cases}$$

There is a fast algorithm for computing lrt (Crochemore 1986) but we give only a simple implementation.

Summarising, we can compute $encode$ by the recursive program

$$encode \quad = \quad (null \rightarrow nil, minlist\,R \cdot list\,(snoc \cdot (encode \times id)) \cdot reduce).$$

As with preceding problems, the computation of $encode$ is inefficient since the same subproblem may be computed more than once. We will not, however, go into the details of a tabulation phase; although the general scheme is clear, namely, to compute $encode$ on all initial segments of the input string, the details are messy.

The program

In the following program a code sequence x is stored as a pair $(x, size\,x)$. The program is parameterised by the function $bytes : Nat \leftarrow Code$ that returns the sizes of symbols and pointers:

```
> data Code = Sym Char | Ptr (String, String)
> encode    = outl . encode'
> encode'   = cond null (nil', minlist r . list f . reduce)
>             where f = snoc' . cross (encode', id)
>                   r = leq . cross (outr, outr)

> nil'      = const ([],0)
> snoc'     = cross (snoc, plus . cross (id, bytes)) . dupr
> reduce w  = [(init w, Sym (last w)), (x, Ptr (y,z))], if z /= []
>           = [(init w, Sym (last w))],                  otherwise
>             where (x,(y,z)) = lrt w
```

```
> lrt w = head [(x,(y,z)) | (x,z) <- splits w, y <- locs (w, z)]
> locs (w,z) = [y | (y, v) <- splits (init w), prefix (z, v)]

> prefix ([], v)    = True
> prefix (z, [])    = False
> prefix (a:z,b:v) = (a == b) && prefix (z,v)
```

Exercises

9.16 Prove that $\alpha \cdot \mathsf{F}(\Pi + \Pi, R) \subseteq R \cdot \alpha$.

9.17 Prove formally that $\mathit{init} \cdot \mathit{decode} \subseteq \mathit{decode} \cdot R$.

9.18 Why can't we take $Q = \mathsf{F}(\Pi, \mathit{prefix})$, where Π is the universal relation on code elements?

9.19 What simplification to the algorithm is possible if it is assumed that $c = p$?

9.20 We can turn *decode* into a total and surjective function by redefining code sequences so that if such a sequence is not empty, then it always begins with a symbol. This means that the converse function theorem is applicable, so decode° can be expressed as a catamorphism. Develop a thinning algorithm to solve the dictionary coding problem. (This is a research problem.)

Bibliographical remarks

In 1957, Bellman published the first book on dynamic programming (Bellman 1957). Bellman showed that the use of dynamic programming is governed by the principle of optimality, and many authors have since considered the formalisation of that principle as a monotonicity condition, e.g. (Bonzon 1970; Mitten 1964; Karp and Held 1967; Sniedovich 1986). The paper by Karp and Held places a lot of emphasis on the sequential nature of dynamic programming, essentially by concentrating on list-based programming problems. The preceding chapter deals with that type of problem.

(Helman and Rosenthal 1985; Helman 1989a) present a wider view of dynamic programming, generalising from lists to more general tree-like datatypes. Our approach is a natural reformulation of those ideas to a categorical setting, making the definitions and proofs more compact by parameterising specifications and programs with functors. Furthermore, the relational calculus admits a clean treatment of indeterminacy.

The work of Smith (Smith and Lowry 1990; Smith 1991) shows close parallels with the view of dynamic programming put forward here: in fact the main difference is in the style of presentation. Smith's work has the additional aim of mechanising the algorithm design process. To this end, Smith has built a system that implements his ideas (Smith 1990), and has illustrated its use with an impressive number of examples. As said before, we have not investigated whether the results of this book are amenable to mechanical application, although we believe they are. The ideas underlying Smith's work are also of an algebraic nature (Smith 1993), but, again, this is rather different in style from the approach taken here.

Another very similar approach to dynamic programming is that of (Gnesi, Montanari, and Martelli 1981), which also starts with algebraic foundations. There it is shown how dynamic programming can be reduced to a graph searching problem. It is in fact possible to view our basic theorem about dynamic programming in these terms (Ning 1997). One advantage of that view is that it allows a smooth combination of branch-and-bound with dynamic programming. Branch-and-bound has been studied in a calculational style by (Fokkinga 1991).

Besides Bellman's original book, there are many other texts on dynamic programming, e.g. (Bellman and Dreyfus 1962; Denardo 1982; Dreyfus and Law 1977).

There is a fair amount of work on tabulation, and on ways in which tabulation schemes may be formally derived (Bird 1980; Boiten 1992; Cohen 1979; Pettorossi 1984). These methods are, however, still *ad-hoc*, and a more generic solution to the problem of tabulation remains elusive.

Finally, a few remarks on the applications considered in this chapter. In the special case of matrix chain multiplication, the bracketing problem admits a much better solution than the one derived here (Hu and Shing 1982, 1984; Yao 1982). The part of the data compression algorithm that we have ignored (finding the longest repeated tail) is discussed in a functional setting by (Giegerich and Kurtz 1995).

Greedy Algorithms

As we said in the preceding chapter, greedy algorithms can be viewed as an extreme case of dynamic programming in which all but a single decomposition of the input are weeded out. The theory is essentially the same as that given in Chapter 9, so most of what follows is devoted to applications.

10.1 Theory

As in the preceding chapter, define $H = (\!|h|\!) \cdot (\!|T|\!)^\circ$, where h and T are F-algebras. The proof of the following theorem is very similar to that of Theorem 9.2 and is left as an exercise:

Theorem 10.1 Let $M = \mathit{min}\, R \cdot \Lambda H$. If h is monotonic on R and Q satisfies $h \cdot \mathsf{F}H \cdot Q^\circ \subseteq R^\circ \cdot h \cdot \mathsf{F}H$, then

$$(\mu X : h \cdot \mathsf{F}X \cdot \mathit{min}\, Q \cdot \Lambda T^\circ) \quad \subseteq \quad M.$$

Theorem 10.1 has exactly the same hypotheses as Theorem 9.2 but appears to give a much stronger result. Indeed it does, but the crucial point is that it is much harder to refine the result to a computationally useful program. To do so, we need, in addition to the conditions described in the preceding chapter, the further—and very strong—condition that Q is a connected preorder on sets returned by ΛT°. This was not the case with the examples given in the preceding chapter. Since Q is a relation on $\mathsf{F}A$ (for some A) and $\mathsf{F}A$ is often a coproduct, we can make use of the following result, which is a variation on Proposition 9.1.

Proposition 10.1 Suppose that V_1 and V_2 have disjoint ranges, that is, suppose that $V_1{}^\circ \cdot V_2 = \emptyset$. Then

$$[U_1,\, U_2] \cdot \mathit{min}\,(Q_1 + Q_2) \cdot \Lambda[V_1,\, V_2]^\circ \quad = \quad (\mathit{ran}\, V_1 \rightarrow W_1,\; W_2),$$

where $W_i = U_i \cdot \mathit{min}\, Q_i \cdot \Lambda V_i{}^\circ$ for $i = 1, 2$.

Recall also Proposition 9.4, which states that the hypotheses of the greedy theorem can be satisfied by taking $Q = \mathsf{F}(U, V)$, where U and V are preorders such that

$$h \cdot \mathsf{F}(U, R) \subseteq R \cdot h \quad \text{and} \quad H \cdot V^{\circ} \subseteq R^{\circ} \cdot H.$$

However, such a choice of Q is not always appropriate when heading for a greedy algorithm since we also require $\mathit{min}\ Q \cdot \Lambda T^{\circ}$ to be entire.

10.2 The detab–entab problem

The following two exercises are taken from (Kernighan and Ritchie 1988):

> **Exercise 1-20.** Write a program *detab* that replaces tabs in the input with the proper number of blanks to space to the next tab stop. Assume a fixed set of tab stops, say every n columns. Should n be a variable or a symbolic parameter?
>
> **Exercise 1-21.** Write a program *entab* that replaces strings of blanks by the minimum number of tabs and blanks to achieve the same spacing. Use the same tab stops as for *detab*. When either a tab or a single blank would suffice to reach a tab stop, which should be given preference?

Our aim in this section is to solve these two exercises. They go together because *entab* is specified as an optimum converse to *detab*.

Detab

The function *detab* is defined as a catamorphism over snoc-lists:

$$\mathit{detab} = (\![\mathit{nil}, \mathit{expand}]\!),$$

where

$$\begin{aligned} \mathit{expand}\,(x, a) &= (a = \mathit{TB} \rightarrow \mathit{fill}\,x,\ x +\!\!+ [a]) \\ \mathit{fill}\,x &= x +\!\!+ \mathit{blanks}\,(n - (\mathit{col}\,x) \bmod n), \end{aligned}$$

and

$$\begin{aligned} \mathit{col} &= (\![\mathit{zero}, \mathit{count}]\!) \\ \mathit{count}\,(c, a) &= (a = \mathit{NL} \rightarrow 0,\ c + 1). \end{aligned}$$

The expression *blanks* m returns a string of m blanks, TB denotes the tab character, and NL the newline character. The function *col* counts the columns in each line of the input, and tab stops occur every n columns.

The specification of *detab* is an executable program, except that it isn't particularly efficient. For greater efficiency we can tuple *detab* and *col* · *detab* to give

$$\langle \mathit{detab}, \mathit{col} \cdot \mathit{detab} \rangle \quad = \quad (\![\mathit{base}, \mathit{step}]\!),$$

where *base* returns $([\,], 0)$ and

$$\mathit{step}\,((x, c), a) \quad = \quad \begin{cases} (x \mathbin{+\!\!+} [\mathit{NL}], 0), & \text{if } a = \mathit{NL} \\ (x \mathbin{+\!\!+} \mathit{blanks}\, m, c + m), & \text{if } a = \mathit{TB} \\ (x \mathbin{+\!\!+} [a], c + 1), & \text{otherwise} \\ \text{where } m = n - c \bmod n. \end{cases}$$

In the following functional program, we implement the snoc-list catamorphism by a loop:

$$(\![\mathit{base}, \mathit{step}]\!) \cdot \mathit{convert} \quad = \quad \mathit{loop}\,\mathit{step} \cdot \langle \mathit{base}, \mathit{id} \rangle,$$

where *convert* converts cons-lists to snoc-lists. The resulting Gofer program is:

```
> detab = outl . loop step . pair (pair (nil, zero), id)
> step ((x,c),a) = (x ++ ['\n'], 0),      if a == '\n'
>                = (x ++ blanks m, c+m), if a == '\t'
>                = (x ++ [a], c+1),       otherwise
>                  where m = n - (c 'mod' n)
> blanks 0       = []
> blanks (m+1)   = ' ' : blanks m
```

There is another optimisation that improves efficiency still further. Observe that *base* and *step* take a particular form, namely,

$$\begin{aligned} \mathit{base} \quad &= \quad \langle \mathit{nil}, c_0 \rangle \\ \mathit{step}\,((x, c), a) \quad &= \quad (x \mathbin{+\!\!+} f\,(c, a), g\,(c, a)), \end{aligned}$$

for some constant c_0 and functions f and g. When *base* and *step* have this form, we have

$$\mathit{outl} \cdot \mathit{loop}\,\mathit{step} \cdot \langle \mathit{base}, \mathit{id} \rangle \quad = \quad \mathit{loop}'\,(f, g) \cdot \langle c_0, \mathit{id} \rangle,$$

where $\mathit{loop}'\,(f, g)$ is defined by the two equations

$$\begin{aligned} \mathit{loop}'\,(f, g)\,(c, [\,]) \quad &= \quad [\,] \\ \mathit{loop}'\,(f, g)\,(c, [a] \mathbin{+\!\!+} x) \quad &= \quad f\,(c, a) \mathbin{+\!\!+} \mathit{loop}'\,(f, g)\,(g\,(c, a), x). \end{aligned}$$

The proof is left as an exercise.

To see what this transformation buys, let $c_{i+1} = g\,(c_i, a_i)$ and $x_{i+1} = f\,(c_i, a_i)$ for $0 \le i < n$. Then,

$$\begin{array}{rcl} h\,[a_0, a_1, \ldots, a_{n-1}] & = & ((x_1 +\!\!+\, x_2) +\!\!+ \cdots) +\!\!+\, x_n \\ h'\,[a_0, a_1, \ldots, a_{n-1}]) & = & x_1 +\!\!+\, (x_2 +\!\!+\, (\cdots +\!\!+\, x_n)), \end{array}$$

where $h = \mathit{outl} \cdot \mathit{loop}\ \mathit{step} \cdot \langle \mathit{base}, \mathit{id} \rangle$ and $h' = \mathit{loop}'\,(f, g) \cdot \langle c_0, \mathit{id} \rangle$. The second form is asymptotically more efficient to compute in any functional language in which $+\!\!+$ is defined in terms of *cons*.

Applying this transformation, and writing $\mathit{detab}' = \mathit{loop}'\,(f, g)$, we obtain the following program:

```
> detab x          = detab'(0,x)
> detab'(c,[])     = []
> detab'(c,a:x)    = ['\n'] ++ detab'(0,x),       if a == '\n'
>                  = blanks m ++ detab'(c+m,x),  if a == '\t'
>                  = [a] ++ detab'(c+1,x),       otherwise
>                    where m = n - c 'mod' n
```

Entab

The more interesting problem is that of computing *entab*. We begin by specifying *entab* formally. The statement that 'strings of blanks are to be replaced by the minimum number of tabs and blanks to achieve the same spacing' can be interpreted as asking for a *shortest* possible output. The other condition on *entab* is that $\mathit{detab} \cdot \mathit{entab} = \mathit{id}$. These two conditions can be combined to give our specification:

$$\mathit{entab} \;\subseteq\; \mathit{min}\, R \cdot \Lambda \mathit{detab}^{\circ},$$

where $R = \mathit{length}^{\circ} \cdot \mathit{leq} \cdot \mathit{length}$.

Derivation

We aim to solve the problem with a greedy algorithm. Since *nil* and *expand* have disjoint ranges we can try to express Q as a coproduct $Q = \mathsf{F}(U, V)$, where $\mathsf{F}(U, V) = \mathit{id} + (V \times U)$. Furthermore, according to Proposition 9.4, the greedy condition holds if we can find U and V to satisfy the two conditions

$$\alpha \cdot \mathsf{F}(U, R) \subseteq R \cdot \alpha \quad \text{and} \quad V \cdot \mathit{detab} \subseteq \mathit{detab} \cdot R,$$

where $\alpha = [\mathit{nil}, \mathit{snoc}]$. Bear in mind, however, the additional requirement that $\mathit{min}\, Q \cdot \Lambda[\mathit{nil}, \mathit{expand}]^{\circ}$ be entire.

Let us see whether we can take $Q = \mathsf{F}(U, V)$ for appropriate U and V. Since $\alpha \cdot \mathsf{F}(\Pi, R) \subseteq R \cdot \alpha$ (see Exercise 10.3), we can choose U to be any preorder we like on characters, including aUb if $a = TB$ or $a = b$. This choice prefers tabs over blanks. It might seem reasonable to choose $V = \mathit{prefix}$, but this idea doesn't work. To see why, suppose $n = 8$ and consider the following example:

$$\mathit{detab}\,[a, b, c, d, e, TB] \quad = \quad [a, b, c, d, e, BL, BL, BL].$$

Although $[a, b, c, d, e, BL, BL]$ is a prefix of the right-hand side, it is longer than $[a, b, c, d, e, TB]$, so the condition $\mathit{prefix} \cdot \mathit{detab} \subseteq \mathit{detab} \cdot R$ fails.

The resolution is to allow only those prefixes that do not cross tab stops; more precisely, define

$$V \quad = \quad \mathit{prefix} \cap (\mathit{fill}^{\circ} \cdot \mathit{fill}).$$

To prove $V \cdot \mathit{detab} \subseteq \mathit{detab} \cdot R$ we reason:

$$\begin{array}{cl}
 & V \cdot \mathit{detab} \\
= & \quad \{\text{since } \mathit{detab} \text{ is a catamorphism}\} \\
 & V \cdot [\mathit{nil}, \mathit{expand}] \cdot \mathsf{F}(\mathit{id}, \mathit{detab}) \cdot [\mathit{nil}, \mathit{snoc}]^{\circ} \\
= & \quad \{\text{coproducts and } V \cdot \mathit{nil} = \mathit{nil}\} \\
 & [\mathit{nil}, V \cdot \mathit{expand}] \cdot \mathsf{F}(\mathit{id}, \mathit{detab}) \cdot [\mathit{nil}, \mathit{snoc}]^{\circ} \\
\subseteq & \quad \{\text{claim: } V \cdot \mathit{expand} \subseteq \mathit{expand} \cup (V \cdot \mathit{outl})\} \\
 & [\mathit{nil}, \mathit{expand} \cup (V \cdot \mathit{outl})] \cdot \mathsf{F}(\mathit{id}, \mathit{detab}) \cdot [\mathit{nil}, \mathit{snoc}]^{\circ} \\
= & \quad \{\text{distributing } \cup\text{; catamorphisms and definition of } \mathsf{F}\} \\
 & \mathit{detab} \ \cup \ (V \cdot \mathit{outl} \cdot (\mathit{detab} \times \mathit{id}) \cdot \mathit{snoc}^{\circ}) \\
= & \quad \{\text{naturality of } \mathit{outl} \text{ and } \mathit{init} = \mathit{outl} \cdot \mathit{snoc}^{\circ}\} \\
 & \mathit{detab} \ \cup \ (V \cdot \mathit{detab} \cdot \mathit{init}).
\end{array}$$

Leaving aside the claim for the moment, we have shown that $X = V \cdot \mathit{detab}$ is a solution of the inequation $X \subseteq \mathit{detab} \cup (X \cdot \mathit{init})$. But init is an inductive relation, so the greatest solution of this inequation is the unique solution of the corresponding equation, namely $X = \mathit{detab} \cup (X \cdot \mathit{init})$. But the unique solution is $X = \mathit{detab} \cdot \mathit{prefix}$, so $V \cdot \mathit{detab} \subseteq \mathit{detab} \cdot \mathit{prefix}$. It is immediate that $\mathit{prefix} \subseteq R$, so we are done.

It remains to prove the claim. We argue:

$$\begin{array}{cl}
 & V \cdot \mathit{expand} \\
= & \quad \{\text{definition of } \mathit{expand}\} \\
 & V \cdot (\mathit{istab} \cdot \mathit{outr} \rightarrow \mathit{fill} \cdot \mathit{outl}, \mathit{snoc}) \\
= & \quad \{\text{conditionals}\} \\
 & (\mathit{istab} \cdot \mathit{outr} \rightarrow V \cdot \mathit{fill} \cdot \mathit{outl}, V \cdot \mathit{snoc})
\end{array}$$

$$
\begin{aligned}
= & \quad \{\text{claim: } V \cdot \mathit{fill} = \mathit{fill} \text{ (exercise)}\} \\
& (\mathit{istab} \cdot \mathit{outr} \rightarrow \mathit{fill} \cdot \mathit{outl},\ V \cdot \mathit{snoc}) \\
\subseteq & \quad \{\text{claim: } V \cdot \mathit{snoc} \subseteq \mathit{snoc} \cup (V \cdot \mathit{outl}) \text{ (exercise)}\} \\
& (\mathit{istab} \cdot \mathit{outr} \rightarrow \mathit{fill} \cdot \mathit{outl},\ \mathit{snoc} \cup (V \cdot \mathit{outl})) \\
\subseteq & \quad \{\text{definition of } \mathit{expand}\} \\
& \mathit{expand} \cup (V \cdot \mathit{outl}).
\end{aligned}
$$

The conditions of the greedy theorem are established, so we can solve our problem by computing the least fixed point of the equation

$$X \quad = \quad [\mathit{nil}, \mathit{snoc}] \cdot (\mathit{id} + (X \times \mathit{id})) \cdot \mathit{min}\ Q \cdot \Lambda[\mathit{nil}, \mathit{expand}]^{\circ},$$

where $Q = \mathit{id} + (V \times U)$. Appeal to Proposition 10.1 gives

$$X \quad = \quad (\mathit{null} \rightarrow \mathit{nil}, \mathit{snoc} \cdot (X \times \mathit{id}) \cdot \mathit{min}\,(V \times U) \cdot \Lambda \mathit{expand}^{\circ}).$$

It remains to implement $\mathit{min}\,(V \times U) \cdot \Lambda \mathit{expand}^{\circ}$. Since

$$\Lambda \mathit{expand}^{\circ}\,(x +\!\!+ [a]) \quad = \quad \{(y, TB) \mid \mathit{fill}\, y = x +\!\!+ [a]\} \ \cup \ \{(x, a)\},$$

and

$$(\exists y : \mathit{fill}\, y = x +\!\!+ [a]) \quad \equiv \quad a = BL \ \wedge \ \mathit{col}\,(x +\!\!+ [a]) \bmod n = 0,$$

we have

$$
(\mathit{min}\,(V \times U) \cdot \Lambda \mathit{expand}^{\circ})\,(x +\!\!+ [a]) = \\
\begin{cases}
\mathit{min}\ V\, S, & \text{if } a = BL \text{ and } \mathit{col}\,(x +\!\!+ [a]) \bmod n = 0 \\
(x, a), & \text{otherwise} \\
\text{where } S = \{(y, TB) \mid \mathit{fill}\, y = x +\!\!+ [a]\}.
\end{cases}
$$

Furthermore,

$$\mathit{min}\ V\,\{(y, TB) \mid \mathit{fill}\, y = x +\!\!+ [a]\} \quad = \quad (\mathit{unfill}\, x, TB),$$

where $\mathit{unfill}\, x$ is the shortest prefix of x satisfying

$$\mathit{fill}\,(\mathit{unfill}\, x) \quad = \quad \mathit{fill}\, x.$$

We can define unfill by

$$
\begin{aligned}
\mathit{unfill}\,[\,] \quad & = \quad [\,] \\
\mathit{unfill}\,(x +\!\!+ [a]) \quad & = \quad \begin{cases} \mathit{unfill}\, x, & \text{if } a = BL \text{ and } \mathit{col}\,(x +\!\!+ [a]) \bmod n \neq 0 \\ x +\!\!+ [a], & \text{otherwise} \end{cases}
\end{aligned}
$$

Writing the resulting greedy algorithm as a Gofer program, we obtain

```
> entab x = [],                if null x
>         = entab y ++ [a], otherwise
>           where (y,a) = contract x

> contract x
>         = (unfill y,'\t'), if a == ' ' && (col x) 'mod' n == 0
>         = (y,a),              otherwise
>           where (y,a) = (init x,last x)

> unfill x = [],          if null x
>          = unfill y, if a == ' ' && col x 'mod' n /= 0
>          = x,          otherwise
>            where (y,a) = (init x,last x)

> col      = loop op . pair (zero, id)
> op (c,a) = 0,    if a == '\n'
>          = c+1, otherwise
```

The program for *entab* involves recomputations of *col*. To improve efficiency, we will express a generalisation of *entab* as a snoc-list catamorphism, and then apply the same transformation that we did for *detab*.

The idea is to define a function *tbc* (short for 'trailing blanks count') satisfying

$$entab\,x \quad = \quad entab\,(unfill\,x) +\!\!+ \; blanks\,(tbc\,x). \tag{10.1}$$

Using the definition of *entab* we obtain

$$\begin{aligned} tbc\,[\,] \quad &= \quad 0 \\ tbc\,(x +\!\!+ [a]) \quad &= \quad \begin{cases} 0, & \text{if } a = BL \text{ and } col\,(x +\!\!+ [a]) \bmod n = 0 \\ tbc\,x + 1, & \text{otherwise.} \end{cases} \end{aligned}$$

The pair $\langle tbc, col\rangle$ can now be defined as a snoc-list catamorphism:

$$\langle tbc, col\rangle \quad = \quad (\![base, op]\!),$$

where *base* returns $(0,0)$ and

$$op\,((t,c),a) \quad = \quad \begin{cases} (t+1, c+1), & \text{if } a = BL \text{ and } (c+1) \bmod n \neq 0 \\ (0, c+1), & \text{if } a = BL \text{ and } (c+1) \bmod n = 0 \\ (0,0), & \text{if } a = NL \\ (0, c+1), & \text{otherwise.} \end{cases}$$

Furthermore, the function $triple = \langle entab \cdot unfill, \langle tbc, col\rangle\rangle$ can also be expressed as a snoc-list catamorphism:

$$triple \;=\; (\!|base, op|\!),$$

where *base* returns $([\,],(0,0))$ and

$$op\,((x,(t,c)),a) = \begin{cases} (x,(t+1,c+1)), & \text{if } a = BL \text{ and } (c+1) \bmod n \neq 0 \\ (x \mathbin{+\!\!+} [TB],(0,c+1)), & \text{if } a = BL \text{ and } (c+1) \bmod n = 0 \\ (x \mathbin{+\!\!+} blanks\,t \mathbin{+\!\!+} [NL],(0,0)), & \text{if } a = NL \\ (x \mathbin{+\!\!+} blanks\,t \mathbin{+\!\!+} [a],(0,c+1)), & \text{otherwise.} \end{cases}$$

Using (10.1) we have

$$entab \;=\; cat \cdot (id \times blanks) \cdot outl \cdot assocl \cdot triple.$$

Finally, applying the same transformation to *triple* as we did to *detab*, we obtain

```
> entab x = entab'(0,0,x)

> entab'(t,c,[]) = blanks t
> entab'(t,c,a:x)
>      = entab'(t+1,c+1,x),                    if a == ' ' && d /= 0
>      = ['\t'] ++ entab'(0,c+1,x),            if a == ' ' && d == 0
>      = blanks t ++ ['\n'] ++ entab'(0,0,x),  if a == '\n'
>      = blanks t ++ [a] ++ entab'(0,c+1,x),   otherwise
>        where d = (c+1) `mod` n
```

Exercises

10.1 Justify $outl \cdot loop\ step \cdot \langle base, id\rangle = loop'\,(f,g) \cdot \langle c_0, id\rangle$.

10.2 In the specification of *entab* why not say that the number of tabs in the output should be maximised?

10.3 Prove that $\alpha \cdot \mathsf{F}(\Pi, R) \subseteq R \cdot \alpha$. How does the algorithm for *entab* change if a single blank is to be preferred over a single tab?

10.4 For $V = prefix \cap (fill^{\circ} \cdot fill)$ prove that

$$\begin{aligned} V \cdot nil &= nil \\ V \cdot fill &= fill \\ V \cdot snoc &\subseteq snoc \cup (V \cdot outl). \end{aligned}$$

Which of these conditions does not hold for $V = prefix$?

10.3 The minimum tardiness problem

The minimum tardiness problem is a scheduling problem from Operations Research (Hochbaum and Shamir 1989; Lawler 1973). Given a bag of jobs, it is required to find some permutation of the bag that minimises the maximum penalty incurred if jobs are not completed on time. The permutation is called a *schedule*, so the specification is

$$\begin{aligned} \mathit{schedule} &\subseteq \mathit{min}\, R \cdot \Lambda \mathit{bagify}^{\circ} \\ R &= \mathit{cost}^{\circ} \cdot \mathit{leq} \cdot \mathit{cost}, \end{aligned}$$

where *bagify* turns a list into a bag. The function *cost* is defined in terms of three positive quantities associated with each job j: (i) the *completion time* $ct\, j$, which determines how long the job j takes to complete; (ii) the *due time* $dt\, j$, which determines the latest time at which j should be completed (measured from the start of the schedule); and (iii) a weighting $wt\, j$, which measures the importance attached to job j. Given these quantities, the penalty $\mathit{penalty}\,(x, j)$ incurred when j is placed at the end of schedule x is defined by

$$\mathit{penalty}\,(x,j) \;=\; (\mathit{sum}\,(\mathit{list}\; ct\; x) + ct\, j - dt\, j) \times wt\, j.$$

The term $\mathit{sum}\,(\mathit{list}\; ct\; x)$ gives the completion time of schedule x. If, when added to $ct\, j$, this gives a time for completing j that is greater than the due time of j, then a penalty is incurred, its size being proportional to the importance of j. If the completion time is less than the due time, then the penalty is negative. Negative penalties are bonuses, but bonuses are ignored in the definition of cost, which measures only the maximum penalty incurred:

$$\mathit{cost} \;=\; \mathit{max}\; \mathit{leq} \cdot \mathsf{P}([\mathit{zero}, \mathit{penalty}] \cdot \alpha^{\circ}) \cdot \Lambda \mathit{prefix},$$

where $\alpha = [\mathit{nil}, \mathit{snoc}]$. We can also describe *cost* recursively by:

$$\begin{aligned} \mathit{cost}\,[\,] &= 0 \\ \mathit{cost}\,(x +\!\!+ [j]) &= \mathit{bmax}\,(\mathit{cost}\, x, \mathit{penalty}\,(x,j)). \end{aligned}$$

It follows that costs are never negative, and a schedule costing zero is one in which all jobs are completed by their due time.

To illustrate the tardiness problem, consider the following three jobs:

	1	2	3
ct	5	10	15
dt	10	20	20
wt	1	3	3

The best schedules are $[2,3,1]$ and $[3,2,1]$, each with a cost of 20; for example:

	2	3	1
time	10	25	30
dt	20	20	10
penalty	0	15	20

The definition of *cost* is given in terms of snoc-lists, although we can use either snoc-lists or cons-lists to build schedules.

As we have seen in Chapter 7 the choice of what kind of list to use can be critical in the success of a greedy approach. Suppose we did go for snoc-lists. Then the final greedy algorithm, if it exists, will take the form

$$\mathit{schedule}\ u \;=\; \begin{cases} [\,], & \text{if } \mathit{emptybag}\ u \\ \mathit{schedule}\ v +\!\!+ [j], & \text{otherwise} \\ \text{where } (v,j) = \mathit{pick}\ u & \end{cases}$$

for some function *pick*. At each stage, therefore, we pick the job that is best placed at the *end* of the schedule. Such algorithms are known as *backward* greedy algorithms. If schedules are described by cons-lists, then the greedy algorithm would involve picking a job that is best placed first in the schedule. In general, it does not follow that if a greedy algorithm exists for snoc-lists, then a similar algorithm exists for cons-lists.

However, armed with foresight, we will use snoc-lists in building schedules. As a function on snoc-lists, *bagify* is defined by the catamorphism

$$\mathit{bagify} \;=\; (\![\mathit{nil}, \mathit{snag}]\!),$$

where *nil* returns the empty bag, and *snag* (a contraction of *snoc* and *bag*, somewhat more attractive than *bsnoc*) takes a pair (u,j) and places j in the bag u, thereby 'snagging' it.

There is another strategic decision that should be mentioned at this point. In the final algorithm the input will be presented as a list rather than a bag. That means we are, in effect, seeking a permutation of the input that minimises cost, so we could have started out with the specification

$$\mathit{schedule} \;\subseteq\; \mathit{min}\ R \cdot \Lambda \mathit{perm}.$$

With this specification another avenue of attack is opened up. The relation *perm* can be defined as a snoc-list catamorphism (see Section 5.6):

$$perm = (\![nil, add]\!),$$

where $add\,(x, j) = y \mathbin{+\!\!+} [j] \mathbin{+\!\!+} z$ for some decomposition $x = y \mathbin{+\!\!+} z$.

In this form, the minimum tardiness problem might be solvable by the greedy method of Chapter 7. However, no greedy algorithm based on catamorphisms exists—or at least no simple one, which is why the problem appears in this chapter and not earlier. To appreciate why, recall that a greedy method based on snoc-list catamorphisms solves not only the problem associated with the given list, but also the problems associated with all its prefixes. Dually, one based on cons-list catamorphisms solves all suffixes of the input.

Now, consider again the three example jobs described above. With the input presented as $[1,2,3]$, the best schedule for prefix $[1,2]$ is $[1,2]$ itself, incurring zero cost. However, this schedule cannot be extended to either $[2,3,1]$ or $[3,2,1]$, the two best solutions for the three jobs. Dually, with a cons-list catamorphism, suppose the input is presented as $[3,2,1]$; again a best schedule for $[2,1]$ is $[1,2]$, but $[1,2]$ cannot be extended to either $[2,3,1]$ or $[3,2,1]$.

Derivation

Although *nil* and *snag* have disjoint ranges, a development along the lines of the detab–entab problem does not work here. For this problem we need to bring context into both the monotonicity and greedy conditions. As a result, the proof of the greedy condition is a little tricky.

With $\alpha = [nil, snoc]$, $\beta = [nil, snag]$ and $\mathsf{F}X = 1 + (X \times Job)$, the monotonicity and greedy conditions read:

$$\alpha \cdot \mathsf{F}(R \cap (bagify^\circ \cdot bagify)) \subseteq R \cdot \alpha \tag{10.2}$$

$$\alpha \cdot \mathsf{F}bagify^\circ \cdot (Q^\circ \cap (\beta^\circ \cdot \beta)) \subseteq R^\circ \cdot \alpha \cdot \mathsf{F}bagify^\circ, \tag{10.3}$$

To prove (10.2) we need the fact that *cost* can be expressed in the form

$$\begin{aligned} cost\,[\,] &= 0 \\ cost\,(x \mathbin{+\!\!+} [j]) &= bmax\,(cost\,x, penalty\,(perm\,x, j)). \end{aligned}$$

This is identical to the earlier recursive characterisation of *cost*, except for the term $perm\,x$. It holds because $penalty\,(x, j)$ depends only on the jobs in the schedule x, not on their order. The reason is that $penalty\,(x, j)$ is defined in terms of the sum of the completion times of jobs in x, and *sum* applied to a list returns the same result as *sum* applied to the underlying bag.

Using $perm = bagify^\circ \cdot bagify$, the new expression for $cost$ can be put in the form for which Proposition 9.3 applies:

$$\begin{aligned} cost \cdot \alpha &= k \cdot \mathsf{F}\langle cost, bagify\rangle \\ k &= [zero, bmax \cdot (id \times (penalty \cdot (bagify^\circ \times id))) \cdot assocr]. \end{aligned}$$

It is easy to check that

$$k \cdot \mathsf{F}(leq \times id) \subseteq leq \cdot k$$

so (10.2) follows on appeal to Proposition 9.3.

For the greedy condition (10.3) we will need the fact that the original definition of $cost$ can be rewritten in the form

$$\begin{aligned} cost \cdot \alpha &= bmax \cdot \langle g, h\rangle && (10.4)\\ g &= [zero, penalty] && (10.5)\\ h &= [zero, cost \cdot outl]. && (10.6) \end{aligned}$$

We will also need two additional facts. Firstly, the cost of a schedule can only increase when more jobs are added to it. In symbols,

$$add \subseteq R^\circ \cdot outl, \qquad (10.7)$$

where add is the relation for which $perm = (\![nil, add]\!)$. A formal proof of (10.7) is left as Exercise 10.7.

The second fact is that $bagify^\circ$ is a catamorphism on bags, that is,

$$bagify^\circ \cdot \beta = [nil, add] \cdot \mathsf{F}bagify^\circ. \qquad (10.8)$$

The proof is left as Exercise 10.8. Putting (10.7) and (10.8) together, we obtain

$$\begin{aligned} & bagify^\circ \cdot \beta \\ = \quad & \{(10.8)\} \\ & [nil, add] \cdot \mathsf{F}bagify^\circ \\ \subseteq \quad & \{(10.7)\} \\ & [nil, R^\circ \cdot outl] \cdot \mathsf{F}bagify^\circ \\ \subseteq \quad & \{\text{definition of } R \text{ and } nil \subseteq cost^\circ \cdot geq \cdot zero\} \\ & cost^\circ \cdot geq \cdot [zero, cost \cdot outl] \cdot \mathsf{F}bagify^\circ \\ = \quad & \{\text{definition of } h\} \\ & cost^\circ \cdot geq \cdot h \cdot \mathsf{F}bagify^\circ. \end{aligned}$$

Now for the proof of (10.3). We start by reasoning:

$$\alpha \cdot \mathsf{F}bagify^\circ \cdot (Q^\circ \cap (\beta^\circ \cdot \beta))$$

$$\begin{array}{ll}
\subseteq & \{\text{monotonicity of composition}\} \\
& (\alpha \cdot \mathsf{F}\mathit{bagify}^{\circ} \cdot Q^{\circ}) \cap (\alpha \cdot \mathsf{F}\mathit{bagify}^{\circ} \cdot \beta^{\circ} \cdot \beta) \\
= & \{\text{catamorphisms, since } \mathit{bagify} = (\![\beta]\!)\} \\
& (\alpha \cdot \mathsf{F}\mathit{bagify}^{\circ} \cdot Q^{\circ}) \cap (\mathit{bagify}^{\circ} \cdot \beta) \\
\subseteq & \{\text{calculation above}\} \\
& (\alpha \cdot \mathsf{F}\mathit{bagify}^{\circ} \cdot Q^{\circ}) \cap (\mathit{cost}^{\circ} \cdot \mathit{geq} \cdot h \cdot \mathsf{F}\mathit{bagify}^{\circ}) \\
\subseteq & \{\text{modular law}\} \\
& \alpha \cdot ((\mathsf{F}\mathit{bagify}^{\circ} \cdot Q^{\circ} \cdot \mathsf{F}\mathit{bagify}) \cap (\alpha^{\circ} \cdot \mathit{cost}^{\circ} \cdot \mathit{geq} \cdot h)) \cdot \mathsf{F}\mathit{bagify}^{\circ} \\
= & \{\text{choose } Q \text{ to satisfy } \mathsf{F}\mathit{bagify}^{\circ} \cdot Q^{\circ} \cdot \mathsf{F}\mathit{bagify} = g^{\circ} \cdot \mathit{geq} \cdot g\} \\
& \alpha \cdot ((g^{\circ} \cdot \mathit{geq} \cdot g) \cap (\alpha^{\circ} \cdot \mathit{cost}^{\circ} \cdot \mathit{geq} \cdot h)) \cdot \mathsf{F}\mathit{bagify}^{\circ} \\
= & \{\text{products}\} \\
& \alpha \cdot \langle g, \mathit{cost} \cdot \alpha \rangle^{\circ} \cdot \langle \mathit{geq} \cdot g, \mathit{geq} \cdot h \rangle \cdot \mathsf{F}\mathit{bagify}^{\circ}
\end{array}$$

The choice $Q = f^{\circ} \cdot \mathit{leq} \cdot f$, where $f = [\mathit{zero}, \mathit{penalty} \cdot (\mathit{bagify}^{\circ} \times \mathit{id})]$, satisfies the required specification. In words, a minimum under Q identifies a job with the least penalty.

To complete the proof it is sufficient to show

$$\alpha \cdot \langle g, \mathit{cost} \cdot \alpha \rangle^{\circ} \cdot \langle \mathit{geq} \cdot g, \mathit{geq} \cdot h \rangle \quad \subseteq \quad R^{\circ} \cdot \alpha.$$

Shunting cost° to the left-hand side, we reason:

$$\begin{array}{ll}
& \mathit{cost} \cdot \alpha \cdot \langle g, \mathit{cost} \cdot \alpha \rangle^{\circ} \cdot \langle \mathit{geq} \cdot g, \mathit{geq} \cdot h \rangle \\
\subseteq & \{\text{since } \mathit{cost} \cdot \alpha = \mathit{bmax} \cdot \langle g, \mathit{cost} \cdot \alpha \rangle \text{ and } \langle g, \mathit{cost} \cdot \alpha \rangle \text{ is simple}\} \\
& \mathit{bmax} \cdot \langle \mathit{geq} \cdot g, \mathit{geq} \cdot h \rangle \\
\subseteq & \{\text{monotonicity of } \mathit{bmax}\} \\
& \mathit{geq} \cdot \mathit{bmax} \cdot \langle g, h \rangle \\
= & \{(10.4)\} \\
& \mathit{geq} \cdot \mathit{cost} \cdot \alpha.
\end{array}$$

The greedy condition (10.3) is now established, so we can solve our problem by computing the least fixed point of

$$X \quad = \quad [\mathit{nil}, \mathit{snoc}] \cdot (\mathit{id} + (X \times \mathit{id})) \cdot \mathit{min}\ Q \cdot \Lambda[\mathit{nil}, \mathit{snag}]^{\circ}.$$

Appeal to Proposition 10.1 gives

$$X \quad = \quad (\mathit{null} \to \mathit{nil}, \mathit{snoc} \cdot (X \times \mathit{id}) \cdot \mathit{min}\ Q' \cdot \Lambda \mathit{snag}^{\circ}),$$

where $Q' = f^{\circ} \cdot \mathit{leq} \cdot f$ and $f = \mathit{penalty} \cdot (\mathit{bagify}^{\circ} \times \mathit{id})$.

Refining $min\ Q' \cdot \Lambda snag^{\circ}$ to a partial function $pick$, we obtain

$$schedule \quad = \quad (null \rightarrow nil, snoc \cdot (schedule \times id) \cdot pick).$$

The program

In the Gofer program we represent bags by lists and represent a list x by the pair $(x, sum\,(list\ ct\ x))$. The function $pick$ is implemented by choosing the first job in the list with minimum penalty.

```
> schedule = schedule' . pair (id, sum . list ct)

> schedule' (x,t) = [],                          if null x
>                 = schedule' (x',t') ++ [j], otherwise
>                   where x' = delete j x
>                         t' = t - ct j
>                         j  = pick (x,t)

> pick (x,t) = outl (minlist r [(j, (t - dt j) * wt j) | j <- x])
>              where r = leq . cross (outr, outr)

> delete j []    = []
> delete j (k:x) = x,                if j == k
>                = k : delete j x, otherwise
```

The running time of this program is quadratic in the number of jobs.

Exercises

10.5 Prove that $k \cdot \mathsf{F}(leq \times id) \subseteq leq \cdot k$, where

$$k \quad = \quad [zero, bmax \cdot (id \times (penalty \cdot (bagify^{\circ} \times id))) \cdot assocr].$$

10.6 Prove that $cost \cdot \alpha = bmax \cdot \langle g, h \rangle$.

10.7 To show that $add \subseteq R^{\circ} \cdot outl$ we can use a recursive characterisation of add:

$$add \quad = \quad (\mu X : snoc \cup (snoc \cdot (X \times id) \cdot exch \cdot (snoc^{\circ} \times id))),$$

where $exch : (A \times C) \times B \leftarrow (A \times B) \leftarrow C$.

Prove that $add \subseteq R^\circ \cdot outl$ using fixed-point induction (see Exercise 6.4) and the fact that

$$penalty \cdot (add \times id) \quad \subseteq \quad geq \cdot penalty \cdot (outl \times id).$$

10.8 Using the fact that $perm = bagify^\circ \cdot bagify = (\![nil, add]\!)$, prove that

$$bagify^\circ \cdot \beta \quad = \quad [nil, add] \cdot \mathsf{F} bagify^\circ .$$

10.9 Assuming all weights are the same, give an $O(n \log n)$ algorithm for computing the complete schedule.

10.10 The minimum *lateness* problem is similar to the minimum tardiness problem, except that the cost function is defined by

$$\begin{aligned} cost\,[\,] \quad &= \quad -\infty \\ cost\,(x +\!\!+ [j]) \quad &= \quad bmax\,(penalty\,(x, j), cost\,x). \end{aligned}$$

It follows that costs can be negative. How does this change affect the development?

10.11 Does the problem in which *cost* is defined by

$$\begin{aligned} cost\,[\,] \quad &= \quad 0 \\ cost\,(x +\!\!+ [j]) \quad &= \quad plus\,(penalty\,(x, j), cost\,x), \end{aligned}$$

have a greedy solution?

10.4 The TEX problem – part two

As a final example, let us solve the second of the TEX problems described in Chapter 3. Recall that the task is to convert between decimal fractions and integer multiples of 2^{-16}. The function *extern* has type $Decimal \leftarrow [0, 2^{16})$ and is specified by the property that *extern* n should be some shortest decimal whose internal representation is n:

$$\begin{aligned} extern \quad &\subseteq \quad min\,R \cdot \Lambda intern^\circ \\ R \quad &= \quad length^\circ \cdot leq \cdot length. \end{aligned}$$

The function *intern* is defined by the equations

$$\begin{aligned} intern \quad &= \quad round \cdot val \\ round\,r \quad &= \quad \lfloor 2^{16} r + 1/2 \rfloor \\ val \quad &= \quad (\![zero, shift]\!) \\ shift\,(d, r) \quad &= \quad (d + r)/10, \end{aligned}$$

in which *val* is a catamorphism on cons-lists. In Chapter 3 we showed how to compute *intern* using integer arithmetic only; this restriction also has to be maintained in the computation of *extern*.

The first job is to cast the problem of computing *extern* into the standard mould. Installing the definition of *intern*, we obtain

$$extern \subseteq min\, R \cdot \Lambda(val^\circ \cdot round^\circ).$$

Since $round^\circ$ is not a function we cannot simply take it out of the Λ expression. Instead, we use the fact that

$$n = \lfloor 2^{16} r + 1/2 \rfloor \quad \equiv \quad 2n - 1 \leq 2^{17} r < 2n + 1$$

to express $round^\circ$ in the form

$$round^\circ \quad = \quad inrange \cdot interval,$$

where

$$\begin{aligned} interval\, n \quad &= \quad ((2n-1)/2^{17}, (2n+1)/2^{17}) \\ r\, inrange\, (a, b) \quad &= \quad (a \leq r < b). \end{aligned}$$

Since *interval* is a function, we can rewrite the specification of *extern* to read

$$extern \subseteq min\, R \cdot \Lambda(val^\circ \cdot inrange) \cdot interval.$$

Finally, we appeal to fusion to show that $inrange^\circ \cdot val$ can be expressed as a catamorphism on cons-lists:

$$inrange^\circ \cdot val \quad = \quad (\!|arb, step|\!).$$

The conditions to be satisfied are

$$\begin{aligned} inrange^\circ \cdot zero \quad &= \quad arb \\ inrange^\circ \cdot shift \quad &= \quad step \cdot (id \times inrange^\circ). \end{aligned}$$

The first condition determines *arb* and to determine *step* we argue:

$$\begin{aligned} & (a, b)\, (inrange^\circ \cdot shift)\, (d, r) \\ \equiv \quad & \quad \{\text{definition of } inrange \text{ and } shift\} \\ & a \leq (d + r)/10 < b \\ \equiv \quad & \quad \{\text{arithmetic and definition of } inrange\} \\ & (10a - d, 10b - d)\, inrange^\circ\, r \\ \equiv \quad & \quad \{\text{arithmetic}\} \\ & (\exists a', b' : a = (d + a')/10 \ \wedge\ b = (d + b')/10 : (a', b')\, inrange^\circ\, r) \end{aligned}$$

$$\begin{array}{ll} \equiv & \{\text{introducing } \mathit{step}\,(d,(a',b')) = ((d+a')/10,(d+b')/10)\} \\ & (a,b)\,(\mathit{step}\cdot(\mathit{id}\times \mathit{inrange}^\circ))\,(d,r). \end{array}$$

Summarising, we now want to determine a function *extern* satisfying

$$\mathit{extern} \;\subseteq\; \mathit{min}\,R\cdot\Lambda(\![\mathit{arb},\mathit{step}]\!)^\circ\cdot \mathit{interval}.$$

So far we haven't considered the restriction on the problem, namely, that the argument to *extern* is an integer n in the range $0 \le n < 2^{16}$. For n in this range we have $\mathit{interval}\,n = (a,b)$, where a and b have the property that

$$0 < b < 1 \quad\text{and}\quad a < b. \qquad (10.9)$$

The important point is that if a' and b' satisfy (10.9), then so do a and b, where $(a,b) = \mathit{step}\,(d,(a',b'))$ and d is a digit. Furthermore, we can always restrict *arb* so that it returns an interval (a,b) satisfying (10.9). Hence, defining *Interval* to be the set of pairs (a,b) satisfying (10.9), we have

$$[\mathit{arb},\mathit{step}] : \mathit{Interval} \leftarrow 1 + (\mathit{Digit}\times \mathit{Interval}).$$

This type restriction is exploited in the derivation.

Derivation

It is easy to check that $\alpha = [\mathit{nil},\mathit{cons}]$ is monotonic under R, so this leaves the greedy condition. From above, it is sufficient to find a Q over the type $\mathsf{F}\mathit{Interval}$, where $\mathsf{F}A = 1 + (\mathit{Digit}\times A)$, satisfying

$$Q\cdot \mathsf{F}h\cdot\alpha^\circ \;\subseteq\; \mathsf{F}h\cdot\alpha^\circ\cdot R,$$

where $h = (\![\mathit{arb},\mathit{step}]\!)$.

For this problem a simple choice of Q suffices. To see why, consider the expression $\Lambda[\mathit{arb},\mathit{step}]^\circ$. Writing $*$ for the sole inhabitant of the terminal object, we have for (a,b) of type *Interval* that

$$\begin{array}{lll} (\Lambda \mathit{arb}^\circ)\,(a,b) & = & (a\le 0 \rightarrow \{*\},\{\,\}) \\ (\Lambda \mathit{step}^\circ)\,(a,b) & = & \{(d,(10a-d,10b-d))\,|\,0<10b-d<1\}. \end{array}$$

But for digits d_1 and d_2, bearing (10.9) in mind,

$$(0<10b-d_1<1)\wedge(0<10b-d_2<1) \;\Rightarrow\; d_1=d_2.$$

Hence $\mathit{step}^\circ : (\mathit{Digit}\times \mathit{Interval}) \leftarrow \mathit{Interval}$ is, in fact, a function

$$\mathit{step}^\circ\,(a,b) \;=\; (d,(10a-d,10b-d)),\quad \text{where } d = \lfloor 10b\rfloor.$$

It follows that

$$(\Lambda[arb, step]^{\circ})\,(a, b) = \begin{cases} \{inl\,(*), inr\,(d, (10a - d, 10b - d))\}, & \text{if } a \leq 0 \\ \{inr\,(d, (10a - d, 10b - d))\}, & \text{otherwise,} \end{cases}$$

and so Q need only choose between two alternatives. The appropriate definition of Q is

$$Q \quad = \quad (inl \cdot ! \cdot inr^{\circ}) \;\cup\; id,$$

where $! : 1 \leftarrow (Digit \times Interval)$. With this choice of Q the inhabitant of the terminal object is preferred whenever possible.

To establish the greedy condition, we argue:

$$\begin{array}{ll}
 & Q \cdot \mathsf{F}h \cdot \alpha^{\circ} \subseteq \mathsf{F}h \cdot \alpha^{\circ} \cdot R \\
\equiv & \quad \{\text{definition of } Q\} \\
 & ((inl \cdot ! \cdot inr^{\circ}) \;\cup\; id) \cdot \mathsf{F}h \cdot \alpha^{\circ} \subseteq \mathsf{F}h \cdot \alpha^{\circ} \cdot R \\
\equiv & \quad \{\text{since } R \text{ is reflexive}\} \\
 & inl \cdot ! \cdot inr^{\circ} \cdot \mathsf{F}h \cdot \alpha^{\circ} \subseteq \mathsf{F}h \cdot \alpha^{\circ} \cdot R \\
\equiv & \quad \{\text{definition of } \mathsf{F}\} \\
 & inl \cdot ! \cdot (id \times h) \cdot inr^{\circ} \cdot \alpha^{\circ} \subseteq \mathsf{F}h \cdot \alpha^{\circ} \cdot R \\
\equiv & \quad \{\text{universal property of } ! \text{ and } \alpha \cdot inr = cons\} \\
 & inl \cdot ! \cdot cons^{\circ} \subseteq \mathsf{F}h \cdot \alpha^{\circ} \cdot R \\
\equiv & \quad \{\text{shunting}\} \\
 & id \subseteq !^{\circ} \cdot inl^{\circ} \cdot \mathsf{F}h \cdot \alpha^{\circ} \cdot R \cdot cons \\
\equiv & \quad \{\text{definition of } \mathsf{F}\} \\
 & id \subseteq !^{\circ} \cdot inl^{\circ} \cdot \alpha^{\circ} \cdot R \cdot cons \\
= & \quad \{\text{since } \alpha \cdot inl = nil\} \\
 & id \subseteq !^{\circ} \cdot nil^{\circ} \cdot R \cdot cons \\
\equiv & \quad \{\text{shunting}\} \\
 & nil \cdot ! \subseteq R \cdot cons \\
\Leftarrow & \quad \{\text{since } length \cdot nil \cdot ! \subseteq leq \cdot length \cdot cons\} \\
 & true.
\end{array}$$

The greedy theorem is therefore applicable, so our problem is solved by computing the least solution of the recursion equation

$$X \quad = \quad \alpha \cdot \mathsf{F}X \cdot min\; Q \cdot \Lambda[arb, step]^{\circ}.$$

We know how to simplify $min\ Q \cdot \Lambda[arb, step]^\circ$ and the result is that

$$extern \quad = \quad f \cdot interval,$$

where

$$f(a,b) \quad = \quad \begin{cases} [\,], & \text{if } a \leq 0 \\ [d] \mathbin{+\!\!+} f(10a - d, 10b - d), & \text{otherwise} \\ \text{where } d = \lfloor 10b \rfloor. \end{cases}$$

The final step is to introduce the restriction that the computation should be performed using integer arithmetic only. This turns out to be easy: writing $w = 2^{17}$, every interval (a, b) computed during the algorithm satisfies $a = p/w$ and $b = q/w$ for some integers p and q. Initially, we have $interval\ n = ((2n-1)/w, (2n+1/w))$ and if $a = p/w$, then $10a - d = (10p - wd)/w$; similarly for b. Representing $(p/w, q/w)$ by (p, q), we therefore obtain that $extern\ n = f(2n-1, 2n+1)$, where

$$f(p,q) \quad = \quad \begin{cases} [\,], & \text{if } p \leq 0 \\ [d] \mathbin{+\!\!+} f(10p - wd, 10q - wd), & \text{otherwise} \\ \text{where } d = (10q) \text{ div } w. \end{cases}$$

The program

Here is the final program written in Gofer:

```
> extern       = f . interval
> f(p,q)       = [], if p <= 0
>              = [d] ++ f(10*p - w*d, 10*q - w*d), otherwise
>                where d = (10*q) `div` w
> interval n = (2*n - 1, 2*n + 1)
> w            = 131072
```

Exercises

10.12 Prove that $0 < 10b - d_1 < 1$ and $0 < 10b - d_2 < 1$ imply that $d_1 = d_2$.

10.13 The derivation of *extern* brought in integer arithmetic as a final step. Using the derived program for *intern*, give a derivation of *extern* that uses integer arithmetic from the outset.

10.14 Show that the only property of $w = 2^{17}$ assumed in the derivation is that $10q/w$ should not be an integer for any q with $0 < q < w$. How can this restriction be removed?

10.15 Actually, Knuth required a slightly more stringent condition on *extern*: among equally short decimals, *extern* n should produce the one which is as close as possible to $n/2^{16}$. Which decimal, precisely, does the given algorithm for *extern* produce? What modification ensures that *extern* n returns the shortest and closest decimal to $n/2^{16}$?

Bibliographical remarks

For general remarks about the literature on greedy algorithms, see Chapter 7. The approach of this chapter is arguably more general, and closer to the view of greedy algorithms in the literature. We originally published the idea that dynamic programming and greedy algorithms are closely related in (Bird and De Moor 1993a). A similar suggestion occurs in (Helman 1989b).

In this chapter we have only considered problems where the base functor F is linear: no tree-like structures were introduced. For non-linear F, the recursion would be more appropriately termed 'divide-and-conquer'. We have not investigated this in detail, but we hope that some of the applications of divide-and-conquer studied by Smith can be treated in this manner (Smith 1985, 1987).

Although the approach sketched here is applicable to a wide class of problems, it still admits of further, meaningful generalisation. In (Curtis 1996), it is shown how, by using a more general form of iteration, a wider class of algorithms can be treated. Essentially, catamorphisms (and their converse) are replaced by a general loop operator; this allows more flexibility in specifications and solutions.

Appendix

The following Gofer prelude file contains definitions of the standard functions necessary for running all the programs in this book. As a prelude for general functional programming it is incomplete.

```
------------------------------------------------------------------
-- Prelude for 'Algebra of Programming' --------------------
-- Created 14 Sept, 1995, by Richard Bird -------------------
------------------------------------------------------------------

-- Operator precedence table: ---------------------------------

infixr 9 .
infixl 7 *
infix  7 /, 'mod'
infixl 6 +, -
infixr 5 ++, :
infix  4 ==, /=, <, <=, >=, >
infixr 3 &&
infixr 2 ||

-- Standard combinators: -------------------------------------

(f . g) x = f (g x)
const k a = k
id a = a

outl (a,b)  = a
outr (a,b)  = b
swap (a,b)  = (b,a)

assocl (a,(b,c)) = ((a,b),c)
assocr ((a,b),c) = (a,(b,c))
```

```
dupl (a,(b,c)) = ((a,b),(a,c))
dupr ((a,b),c) = ((a,c),(b,c))

pair (f,g) a      = (f a, g a)
cross (f,g) (a,b) = (f a, g b)
cond p (f,g) a    = if (p a) then (f a) else (g a)

curry f a b     = f (a,b)
uncurry f (a,b) = f a b

-- Boolean functions: ----------------------------------------

false = const False
true  = const True

False && x = False
True  && x = x

False || x = x
True  || x = True

not True   = False
not False  = True

otherwise  = True

-- Relations: ------------------------------------------------

leq   = uncurry (<=)
less  = uncurry (<)
eql   = uncurry (==)
neq   = uncurry (/=)
gtr   = uncurry (>)
geq   = uncurry (>=)

meet (r,s) = cond r (s, false)
join (r,s) = cond r (true, s)
wok r      = r . swap

-- Numerical functions: --------------------------------------

zero = const 0
succ = (+1)
pred = (-1)
```

```
plus   = uncurry (+)
minus  = uncurry (-)
times  = uncurry (*)
divide = uncurry (/)

negative = (< 0)
positive = (> 0)

-- List-processing functions: --------------------------------

[] ++ y    = y
(a:x) ++ y = a : (x++y)

null []    = True
null (a:x) = False

nil    = const []
wrap   = cons . pair (id, nil)
cons   = uncurry (:)
cat    = uncurry (++)
concat = catalist ([], cat)
snoc   = cat . cross (id, wrap)

head (a:x) = a
tail (a:x) = x
split      = pair (head, tail)

last = cata1list (id, outr)
init = cata1list (nil, cons)

inits  = catalist ([[]], extend)
         where extend (a,xs) = [[]] ++ list (a:) xs
tails  = catalist ([[]], extend)
         where extend (a,x:xs) = (a : x) : x : xs
splits = zip . pair (inits, tails)

cpp (x,y) = [(a,b) | a <- x, b <- y]
cpl (x,b) = [(a,b) | a <- x]
cpr (a,y) = [(a,b) | b <- y]
cplist    = catalist ([[]], list cons . cpp)

minlist r    = cata1list (id, bmin r)
bmin r       = cond r (outl, outr)
```

```
maxlist r    = catallist (id, bmax r)
bmax r       = cond (r . swap) (outl, outr)

thinlist r   = catalist ([], bump r)
               where bump r (a,[])  = [a]
                     bump r (a,b:x) | r(a,b)    = a:x
                                    | r(b,a)    = b:x
                                    | otherwise = a:b:x

length   = catalist (0, succ . outr)
sum      = catalist (0, plus)
trans    = catallist (list wrap, list cons . zip)
list f   = catalist ([], cons . cross (f, id))
filter p = catalist ([], cond (p . outl) (cons, outr))

catalist (c,f) []     = c
catalist (c,f) (a:x)  = f (a, catalist (c,f) x)

catallist (f,g) [a]   = f a
catallist (f,g) (a:x) = g (a, catallist (f,g) x)

cata2list (f,g) [a,b] = f (a,b)
cata2list (f,g) (a:x) = g (a, cata2list (f,g) x)

loop f (a,[])  = a
loop f (a,b:x) = loop f (f (a,b), x)

merge r ([],y)    = y
merge r (x,[])    = x
merge r (a:x,b:y) | r (a,b)   = a : merge r (x,b:y)
                  | otherwise = b : merge r (a:x,y)

zip (x,[])    = []
zip ([],y)    = []
zip (a:x,b:y) = (a,b) : zip (x,y)

unzip = pair (list outl, list outr)

-- Word and line processing functions: ----------------------

words = filter (not.null) . catalist ([[]], cond ok (glue, new))
        where ok (a,xs)      = (a /= ' ' && a /= '\n')
              glue (a,x:xs)  = (a:x):xs
              new (a,xs)     = []:xs
```

```
lines = catalist ([[]], cond ok (glue, new))
        where ok (a,xs)     = (a /= '\n')
              glue (a,x:xs) = (a:x):xs
              new (a,xs)    = []:xs

unwords = cata1list (id, join)
          where join (x,y) = x ++ " " ++ y

unlines = cata1list (id, join)
          where join (x,y) = x ++ "\n" ++ y

-- Essentials and built-in primitives: -----------------------

primitive ord "primCharToInt" :: Char -> Int
primitive chr "primIntToChar" :: Int -> Char

primitive (==) "primGenericEq",
          (/=) "primGenericNe",
          (<=) "primGenericLe",
          (<)  "primGenericLt",
          (>=) "primGenericGe",
          (>)  "primGenericGt"   :: a -> a -> Bool

primitive (+)  "primPlusInt",
          (-)  "primMinusInt",
          (/)  "primDivInt",
          div  "primDivInt",
          mod  "primModInt",
          (*)  "primMulInt"      :: Int -> Int -> Int

primitive negate "primNegInt"    :: Int -> Int
primitive primPrint "primPrint"  :: Int -> a -> String -> String
primitive strict "primStrict"    :: (a -> b) -> a -> b
primitive error "primError"      :: String -> a

show      ::  a -> String
show x     =  primPrint 0 x []

flip f a b = f b a

-- End of Algebra of Programming prelude ---------------------
--------------------------------------------------------------
```

Bibliography

Aarts, C. J., Backhouse, R. C., Hoogendijk, P. F., Voermans, E., and Van der Woude, J. C. S. P. (1992). A relational theory of datatypes. Available from URL `http://www.win.tue.nl/win/cs/wp/papers/papers.html`.

Ahrens, J. H. and Finke, G. (1975). Merging and sorting applied to the 0-1 knapsack problem. *Operations Research*, *23*(6), 1099–1109.

Asperti, A. and Longo, G. (1991). *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist*. Foundations of Computing Series. MIT Press.

Augusteijn, A. (1992). An alternative derivation of a binary heap construction function. In Bird, R. S., Morgan, C. C., and Woodcock, J. C. P., editors, *Mathematics of Program Construction*, Volume 669 of *Lecture Notes in Computer Science*, pages 368–374. Springer-Verlag.

Backhouse, R. C. and Hoogendijk, P. F. (1993). Elements of a relational theory of datatypes. In Möller, B., Partsch, H., and Schuman, S., editors, *Formal Program Development*, Volume 755 of *Lecture Notes in Computer Science*, pages 7–42. Springer-Verlag.

Backhouse, R. C. and Van der Woude, J. C. S. P. (1993). Demonic operators and monotype factors. *Mathematical Structures in Computing Science*, *3*(4), 417–433.

Backhouse, R. C., De Bruin, P., Malcolm, G., Voermans, T. S., and Van der Woude, J. C. S. P. (1991). Relational catamorphisms. In Möller, B., editor, *Constructing Programs from Specifications*, pages 287–318. Elsevier Science Publishers.

Backhouse, R. C., De Bruin, P., Hoogendijk, P. F., Malcolm, G., Voermans, T. S., and Van der Woude, J. C. S. P. (1992). Polynomial relators. In Nivat, M., Rattray, C. S., Rus, T., and Scollo, G., editors, *Algebraic Methodology and Software Technology*, Workshops in Computing, pages 303–362. Springer-Verlag.

Backus, J. (1978). Can programming be liberated from the Von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, *21*, 613–641.

Backus, J. (1981). The algebra of functional programs: function level reasoning, linear equations and extended definitions. In Díaz, J. and Ramos, I., editors, *Formalization of Programming Concepts*, Volume 107 of *Lecture Notes in Computer Science*, pages 1–43. Springer-Verlag.

Backus, J. (1985). From function level semantics to program transformations and optimization. In Ehrig, H., Floyd, C., Nivat, M., and Thatcher, J., editors, *Mathematical Foundations of Software Development, Vol. 1*, Volume 185 of *Lecture Notes in Computer Science*, pages 60–91. Springer-Verlag.

Barr, M. and Wells, C. (1985). *Toposes, Triples and Theories*, Volume 278 of *Grundlehren der Mathematischen Wissenschaften*. Springer-Verlag.

Barr, M. and Wells, C. (1990). *Category Theory for Computing Science*. International Series in Computer Science. Prentice Hall.

Bauer, F. L., Berghammer, R., Broy, M., Dosch, W., Geiselbrechtinger, F., Gnatz, R., Hangel, E., Hesse, W., Krieg-Brückner, B., Laut, A., Matzner, T., Möller, B., Nickl, F., Partsch, H., Pepper, P., Samelson, K., Wirsing, M., and Wössner, H. (1985). *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, Volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag.

Bauer, F. L., Ehler, H., Horsch, A., Möller, B., Partsch, H., Paukner, O., and Pepper, P. (1987). *The Munich Project CIP. Volume II: The Program Transformation System CIP-S*, Volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag.

Bellman, R. E. and Dreyfus, S. E. (1962). *Applied Dynamic Programming*. Princeton University Press.

Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press.

Berghammer, R. and Von Karger, B. (1995). Formal derivation of CSP programs from temporal specifications. In *Mathematics of Program Construction*, Volume 947 of *Lecture Notes in Computer Science*, pages 180–196. Springer-Verlag.

Berghammer, R. and Zierer, H. (1986). Relational algebraic semantics of deterministic and non-deterministic programs. *Theoretical Computer Science*, *43*(2–3), 123–147.

Berghammer, R., Kempf, P., Schmidt, G., and Ströhlein, T. (1991). Relation algebra and logic of programs. In Andreka, H. and Monk, J. D., editors, *Algebraic Logic*, Volume 54 of *Colloquia Mathematica Societatis Janos Bolyai*, pages 37–58. North-Holland.

Bird, R. S. and De Moor, O. (1993a). From dynamic programming to greedy algorithms. In Möller, B., Partsch, H., and Schuman, S., editors, *Formal Program Development*, Volume 755 of *Lecture Notes in Computer Science*, pages 43–61. Springer-Verlag.

Bird, R. S. and De Moor, O. (1993b). List partitions. *Formal Aspects of Computing*, *5*(1), 61–78.

Bird, R. S. and De Moor, O. (1993c). Solving optimisation problems with catamorphisms. In Bird, R. S., Morgan, C. C., and Woodcock, J. C. P., editors, *Mathematics of Program Construction*, Volume 669 of *Lecture Notes in Computer Science*, pages 45–66. Springer-Verlag.

Bird, R. S. and De Moor, O. (1994). Relational program derivation and context-free language recognition. In Roscoe, A. W., editor, *A Classical Mind: Essays dedicated to C.A.R. Hoare*, pages 17–35. Prentice Hall.

Bird, R. S. and Meertens, L. (1987). Two exercises found in a book on algorithmics. In Meertens, L., editor, *Program Specification and Transformation*, pages 451–458. North-Holland.

Bird, R. S. and Wadler, P. (1988). *Introduction to Functional Programming*. International Series in Computer Science. Prentice Hall.

Bird, R. S., Gibbons, J., and Jones, G. (1989). Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, *12*(2), 93–104.

Bird, R. S., Hoogendijk, P. F., and De Moor, O. (1996). Generic programming with relations and functors. *Journal of Functional Programming*, *6*(1), 1–28.

Bird, R. S. (1980). Tabulation techniques for recursive programs. *Computing Surveys*, *12*(4), 403–417.

Bird, R. S. (1984). The promotion and accumulation strategies in functional programming. *ACM Transactions on Programming Languages and Systems*, *6*(4), 487–504.

Bird, R. S. (1986). Transformational programming and the paragraph problem. *Science of Computer Programming*, *6*(2), 159–189.

Bird, R. S. (1987). An introduction to the theory of lists. In Broy, M., editor, *Logic of Programming and Calculi of Discrete Design*, Volume 36 of *NATO ASI Series F*, pages 3–42. Springer-Verlag.

Bird, R. S. (1989a). Algebraic identities for program calculation. *Computer Journal*, *32*(2), 122–126.

Bird, R. S. (1989b). Lectures on constructive functional programming. In Broy, M., editor, *Constructive Methods in Computing Science*, Volume 55 of *NATO ASI Series F*, pages 151–216. Springer-Verlag.

Bird, R. S. (1990). A calculus of functions for program derivation. In Turner, D. A., editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 287–308. Addison-Wesley.

Bird, R. S. (1991). Knuth's problem. In Möller, B., editor, *Constructing Programs from Specifications*, pages 1–8. Elsevier Science Publishers.

Bird, R. S. (1992a). The smallest upravel. *Science of Computer Programming*, *18*(3), 281–292.

Bird, R. S. (1992b). Two greedy algorithms. *Journal of Functional Programming*, *2*(2), 237–244.

Bird, R. S. (1992c). Unravelling greedy algorithms. *Journal of Functional Programming*, *2*(3), 375–385.

Bleeker, A. M. (1994). The calculus of minimals. M.Sc. thesis INF/SCR-1994-01, Department of Computer Science, Utrecht University, The Netherlands. Available from URL `http://www.cwi.nl/~annette/Papers/calculus_minimals.ps`.

Boiten, E. A. (1992). Improving recursive functions by inverting the order of evaluation. *Science of Computer Programming*, *18*(2), 139–179.

Bonzon, P. (1970). Necessary and sufficient conditions for dynamic programming of combinatorial type. *Journal of the ACM*, *17*(4), 675–682.

Brink, C. and Schmidt, G., editors. (1996). *Relational Methods in Computer Science*. Springer-Verlag. Supplemental Volume of the Journal *Computing*, to appear.

Brinkmann, H. B. (1969). Relations for exact categories. *Journal of Algebra*, *13*, 465–480.

Brook, T. (1977). *Order and Recursion in Topoi*, Volume 9 of *Notes on Pure Mathematics*. Department of Mathematics, Australian National University, Canberra.

Broome, P. and Lipton, J. (1994). Combinatory logic programming: computing in relation calculi. In Bruynooghe, M., editor, *Logic Programming*. MIT Press.

Brown, C. and Hutton, G. (1994). Categories, allegories and circuit design. In *Logic in Computer Science*, pages 372–381. IEEE Computer Society Press.

Burstall, R. M. and Darlington, J. (1977). A transformation system for developing recursive programs. *Journal of the ACM*, *24*(1), 44–67.

Burstall, R. M. and Landin, P. J. (1969). Programs and their proofs: an algebraic approach. In *Machine Intelligence*, Volume 4, pages 17–43. American Elsevier.

Carboni, A. and Street, R. (1986). Order ideals in categories. *Pacific Journal of Mathematics*, *124*(2), 275–288.

Carboni, A. and Walters, R. F. C. (1987). Cartesian bicategories I. *Journal of Pure and Applied Algebra*, *49*(1–2), 11–32.

Carboni, A., Kasangian, S., and Street, R. (1984). Bicategories of spans and relations. *Journal of Pure and Applied Algebra*, *33*(3), 259–267.

Carboni, A., Kelly, G. M., and Wood, R. J. (1991). A 2-categorical approach to geometric morphisms I. *Cahiers de Topologie et Geometrie Differentielle Categoriques*, *32*(1), 47–95.

Carboni, A., Lack, S., and Walters, R. F. C. (1993). Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra*, *84*(2), 145–158.

Chen, W. and Udding, J. T. (1990). Program inversion: more than fun!. *Science of Computer Programming*, *15*(1), 1–13.

Clark, K. L. and Darlington, J. (1980). Algorithm classification through synthesis. *Computer Journal*, *23*(1), 61–65.

Cockett, J. R. B. and Fukushima, T. (1991). About Charity. Technical Report 92/480/18, Department of Computer Science, University of Calgary, Canada. Available from URL
`http://www.cpsc.ucalgary.ca/projects/charity/home.html`.

Cockett, J. R. B. and Spencer, D. (1992). Strong categorical datatypes I. In Seely, R. A. G., editor, *Category Theory 1991*, Volume 13 of *CMS Conference Proceedings*, pages 141–169. Canadian Mathematical Society.

Cockett, J. R. B. (1990). List-arithmetic distributive categories: locoi. *Journal of Pure and Applied Algebra*, *66*(1), 1–29.

Cockett, J. R. B. (1991). Conditional control is not quite categorical control. In Birtwistle, G., editor, *Higher-order Workshop*, Workshops in Computing, pages 190–217. Springer-Verlag.

Cockett, J. R. B. (1993). Introduction to distributive categories. *Mathematical Structures in Computer Science*, *3*(3), 277–307.

Cohen, N. H. (1979). Characterization and elimination of redundancy in recursive programs. In *Principles of Programming Languages*, pages 143–157. Association for Computing Machinery.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms.* The MIT electrical engineering and computer science series. MIT Press.

Crochemore, M. (1986). Transducers and repetitions. *Theoretical Computer Science*, *45*(1), 63–86.

Curtis, S. and Lowe, G. (1995). A graphical calculus. In Möller, B., editor, *Mathematics of Program Construction*, Volume 947 of *Lecture Notes in Computer Science*, pages 214–231. Springer-Verlag.

Curtis, S. (1996). *A relational approach to optimization problems.* D.Phil. thesis, Computing Laboratory, Oxford, UK. Available from URL `http://www.comlab.ox.ac.uk/oucl/users/sharon.curtis/publications.html`.

Darlington, J. (1978). A synthesis of several sorting algorithms. *Acta Informatica*, *11*(1), 1–30.

Davey, B. A. and Priestley, H. A. (1990). *Introduction to Lattices and Order.* Cambridge Mathematical Textbooks. Cambridge University Press.

Davie, A. J. T. (1992). *Introduction to Functional Programming Systems using Haskell*, Volume 27 of *Computer Science Texts.* Cambridge University Press.

De Bakker, J. W. and De Roever, W. P. (1973). A calculus for recursive program schemes. In Nivat, M., editor, *Automata, Languages and Programming*, pages 167–196. North-Holland.

De Moor, O. (1992a). Categories, relations and dynamic programming. D.Phil. thesis. Technical Monograph PRG-98, Computing Laboratory, Oxford, UK.

De Moor, O. (1992b). Inductive data types for predicate transformers. *Information Processing Letters*, *43*(3), 113–118.

De Moor, O. (1994). Categories, relations and dynamic programming. *Mathematical Structures in Computing Science*, *4*, 33–69.

De Moor, O. (1995). A generic program for sequential decision processes. In Hermenegildo, M. and Swierstra, D. S., editors, *Programming Languages: Implementations, Logics, and Programs*, Volume 982 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag.

De Morgan, A. (1860). On the syllogism, no. IV, and on the logic of relations. *Transactions of the Cambridge Philosophical Society*, *10*, 331–358. Reprinted in: (De Morgan 1966).

De Morgan, A. (1966). *"On the syllogism" and other logical writings.* Yale University Press.

De Roever, W. P. (1972). A formalization of various parameter mechanisms as products of relations within a calculus of recursive program schemes. In *Théorie des Algorithmes, des Langages et de la Programmation*, pages 55–88. Séminaires IRIA.

De Roever, W. P. (1976). Recursive program schemes: semantics and proof theory. Mathematical Centre Tracts 70, Mathematisch Centrum, Amsterdam, The Netherlands.

Denardo, E. V. (1982). *Dynamic Programming – Models and Applications.* Prentice Hall.

Desharnais, J., Mili, A., and Mili, F. (1993). On the mathematics of sequential decompositions. *Science of Computer Programming*, *20*(3), 253–289.

Dijkstra, E. W. and Scholten, C. S. (1990). *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag.

Dijkstra, E. W. (1976). *A Discipline of Programming.* Series in Automatic Computation. Prentice Hall.

Dijkstra, E. W. (1979). Program inversion. In Bauer, F. L. and Broy, M., editors, *Program Construction*, Volume 69 of *Lecture Notes in Computer Science*, pages 54–57. Springer-Verlag.

Doornbos, H. and Backhouse, R. C. (1995). Induction and recursion on datatypes. In Möller, B., editor, *Mathematics of Program Construction*, Volume 947 of *Lecture Notes in Computer Science*, pages 242–256. Springer-Verlag.

Dreyfus, S. E. and Law, A. M. (1977). *The Art and Theory of Dynamic Programming*, Volume 130 of *Mathematics in Science and Engineering.* Academic Press.

Eilenberg, S. and Wright, J. B. (1967). Automata in general algebras. *Information and Control*, *11*(4), 452–470.

Enderton, H. B. (1977). *Elements of Set Theory*. Academic Press.

Eppstein, D., Galil, Z., Giancarlo, R., and Italiano, G. F. (1992). Sparse dynamic programming II: Convex and concave cost functions. *Journal of the ACM*, *39*(3), 546–567.

Feferman, S. (1969). Set-theoretical foundations of category theory. In *Reports of the Midwest Category Seminar III*, Volume 106 of *Lecture Notes in Mathematics*, pages 201–247. Springer-Verlag.

Fegaras, L., Sheard, T., and Stemple, D. (1992). Uniform traversal combinators: definition, use and properties. In Kapur, D., editor, *Automated Deduction*, Volume 607 of *Lecture Notes in Computer Science*, pages 148–162. Springer-Verlag.

Field, A. J. and Harrison, P. G. (1988). *Functional Programming*. International computer science series. Addison-Wesley.

Fokkinga, M. M. (1991). An exercise in transformational programming: backtracking and branch-and-bound. *Science of Computer Programming*, *16*(1), 19–48.

Fokkinga, M. M. (1992a). Calculate categorically!. *Formal Aspects of Computing*, *4*(4), 673–692.

Fokkinga, M. M. (1992b). A gentle introduction to category theory – the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, pages 1–72. University of Utrecht. Available from URL `http://hydra.cs.utwente.nl/~fokkinga/mmf92b.html`.

Fokkinga, M. M. (1992c). *Law and order in algorithmics*. Ph.D. thesis, Technical University Twente, The Netherlands. Available from URL `http://hydra.cs.utwente.nl/~fokkinga/mmfphd.html`.

Fokkinga, M. M. (1996). Datatype laws without signatures. *Mathematical Structures in Computer Science*, *6*, 1–32.

Freyd, P. J. and Šč edrov, A. (1990). *Categories, Allegories*, Volume 39 of *Mathematical Library*. North-Holland.

Galil, Z. and Giancarlo, R. (1989). Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, *64*, 107–118.

Gardiner, P. H. B., Martin, C. E., and De Moor, O. (1994). An algebraic construction of predicate transformers. *Science of Computer Programming*, *22*(1-2), 21–44.

Gibbons, J., Cai, W., and Skillicorn, D. B. (1994). Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, *23*, 1–18.

Gibbons, J. (1991). Algebras for tree algorithms. D.Phil. thesis. Technical Monograph PRG-94, Computing Laboratory, Oxford, UK.

Gibbons, J. (1993). Upwards and downwards accumulations on trees. In Bird, R. S., Morgan, C. C., and Woodcock, J. C. P., editors, *Mathematics of Program Construction*, Volume 669 of *Lecture Notes in Computer Science*, pages 122–138. Springer-Verlag.

Gibbons, J. (1995). An initial-algebra approach to directed acyclic graphs. In Möller, B., editor, *Mathematics of Program Construction*, Volume 947 of *Lecture Notes in Computer Science*, pages 282–303. Springer-Verlag.

Giegerich, R. and Kurtz, S. (1995). A comparison of purely functional suffix tree constructions. *Science of Computer Programming*, *25*, 187–218.

Gnesi, S., Montanari, U., and Martelli, A. (1981). Dynamic programming as graph searching: an algebraic approach. *Journal of the Association for Computing Machinery*, *28*(4), 737–751.

Goguen, J. A. and Meseguer, J. (1983). Correctness of recursive parallel nondeterministic flow programs. *Journal of Computer and System Sciences*, *27*(2), 268–290.

Goguen, J. A. (1980). How to prove inductive hypotheses without induction. In Bibel, W. and Kowalski, R., editors, *Automated Deduction*, Volume 87 of *Lecture Notes in Computer Science*, pages 356–373.

Goldblatt, R. (1986). *Topoi – The Categorial Analysis of Logic*, Volume 98 of *Studies in Logic and the Foundations of Mathematics*. North-Holland.

Gries, D. (1981). *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag.

Gries, D. (1984). A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, *2*, 207–214.

Gries, D. (1990a). Binary to decimal, one more time. In *Beauty is our Business: A Birthday Salute to Edsger W. Dijkstra*, pages 141–148. Springer-Verlag.

Gries, D. (1990b). The maximum-segment sum problem. In Dijkstra, E. W., editor, *Formal Development of Programs and Proofs*. Addison-Wesley.

Grillet, P. A. (1970). Regular categories. In Barr, M., Grillet, P. A., and Van Osdol, D. H., editors, *Exact Categories and Categories of Sheaves*, Volume 236 of *Lecture Notes in Mathematics*, pages 121–222. Springer-Verlag.

Hagino, T. (1987a). Category theoretic approach to data types. Ph.D. thesis. Technical Report ECS-LFCS-87-38, Laboratory for Foundations of Computer Science, University of Edinburgh, UK.

Hagino, T. (1987b). A typed lambda calculus with categorical type constructors. In Pitt, D. H., Poigne, A., and Rydeheard, D. E., editors, *Category Theory and Computer Science*, Volume 283 of *Lecture Notes in Computer Science*, pages 140–157. Springer-Verlag.

Hagino, T. (1989). Codatatypes in ML. *Journal of Symbolic Computation*, *8*, 629–650.

Hagino, T. (1993). A categorical programming language. In Takeichi, M., editor, *Advances in Software Science and Technology*, Volume 4, pages 111–135. Academic Press.

Harrison, P. G. and Khoshnevisan, H. (1988). Algebraic transformation techniques for functional languages. *Computer Journal*, *31*(3), 229–242.

Harrison, P. G. and Khoshnevisan, H. (1992). On the synthesis of function inverses. *Acta Informatica*, *29*(3), 211–239.

Harrison, P. G. (1988). Linearisation: an optimisation for nonlinear functional programs. *Science of Computer Programming*, *10*(3), 281–318.

Harrison, P. G. (1991). Towards the synthesis of static parallel algorithms. In Möller, B., editor, *Constructing Programs from Specifications*, pages 49–69. Elsevier Science Publishers.

Helman, P. and Rosenthal, A. (1985). A comprehensive model of dynamic programming. *SIAM Journal on Algebraic and Discrete Methods*, *6*(2), 319–334.

Helman, P., Moret, B. M. E., and Shapiro, H. D. (1993). An exact characterization of greedy structures. *SIAM Journal of Discrete Mathematics*, *6*(2), 274–283.

Helman, P. (1989a). A common schema for dynamic programming and branch-and-bound algorithms. *Journal of the ACM*, *36*(1), 97–128.

Helman, P. (1989b). A theory of greedy structures based on k-ary dominance relations. Technical report CS89-11, Department of Computer Science, The University of New Mexico, USA.

Henson, M. (1987). *Elements of Functional Languages*. Computer Science Texts. Blackwell Scientific Publications Ltd.

Hirschberg, D. S. and Larmore, L. L. (1987). The least weight subsequence problem. *SIAM Journal on Computing*, *16*(4), 628–638.

Hoare, C. A. R. and He, J. (1986a). The weakest prespecification, I. *Fundamenta Informaticae*, *9*(1), 51–84.

Hoare, C. A. R. and He, J. (1986b). The weakest prespecification, II. *Fundamenta Informaticae*, *9*(2), 217–251.

Hoare, C. A. R. and He, J. (1987). The weakest prespecification. *Information Processing Letters*, *24*(2), 127–132.

Hoare, C. A. R., He, J., and Sanders, J. W. (1987). Prespecification in data refinement. *Information Processing Letters*, *25*(2), 71–76.

Hoare, C. A. R. (1962). Quicksort. *Computer Journal*, *5*, 10–15.

Hochbaum, D. S. and Shamir, R. (1989). An $o(n \log^2 n)$ algorithm for the maximum weighted tardiness problem. *Information Processing Letters*, *31*, 215–219.

Hoogendijk, P. F. (1996). *A generic theory of datatypes*. Ph.D. thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands.

Hu, T. C. and Shing, M. T. (1982). Computation of matrix chain products, part I. *SIAM Journal on Computing*, *11*(2), 362–373.

Hu, T. C. and Shing, M. T. (1984). Computation of matrix chain products, part II. *SIAM Journal on Computing*, *13*(2), 228–251.

Hu, Z., Iwasaki, H., and Takeichi, M. (1996). Calculating accumulations. Technical Report METR 96-0-3, Department of Mathematical Engineering, University of Tokyo, Japan. Available from URL: `http://www.ipl.t.u-tokyo.ac.jp/~hu/pub/tech.html`.

Hutton, G. (1992). Between functions and relations in calculating programs. Ph.D. Thesis. Research report FP-93-5, Department of Computer Science, Glasgow University, UK. Available from URL `http://www.cs.nott.ac.uk/Department/Staff/gmh/`.

Jay, C. B. and Cockett, J. R. B. (1994). Shapely types and shape polymorphism. In Sannella, D., editor, *Programming Languages and Systems – ESOP '94*, Lecture Notes in Computer Science, pages 302–316. Springer-Verlag.

Jay, C. B. (1994). Matrices, monads and the fast fourier transform. In *Proceedings of the Massey Functional Programming Workshop 1994*, pages 71–80.

Jay, C. B. (1995). Polynomial polymorphism. In Kotagiri, R., editor, *Proceedings of the Eighteenth Australasian Computer Science Conference: Glenelg, South Australia 1–3 February, 1995*, Volume 17, pages 237–243. A. C. S. Communications.

Jeuring, J. T. (1989). Deriving algorithms on binary labelled trees. In Apers, P. M. G., Bosman, D., and van Leeuwen, J., editors, *Computing Science in the Netherlands*, pages 229–249. SION.

Jeuring, J. T. (1990). Algorithms from theorems. In Broy, M. and Jones, C. B., editors, *Programming Concepts and Methods*, pages 247–266. North-Holland.

Jeuring, J. T. (1991). The derivation of hierarchies of algorithms on matrices. In Möller, B., editor, *Constructing Programs from Specifications*, pages 9–32. Elsevier Science Publishers.

Jeuring, J. T. (1993). *Theories for algorithm calculation*. Ph.D. thesis, University of Utrecht, The Netherlands.

Jeuring, J. T. (1994). The derivation of on-line algorithms, with an application to finding palindromes. *Algorithmica*, *11*(2), 146–184.

Jeuring, J. T. (1995). Polytypic pattern matching. In Peyton-Jones, S., editor, *Functional Programming Languages and Computer Architecture*, pages 238–248. Association for Computing Machinery.

Jones, G. and Sheeran, M. (1990). Circuit design in Ruby. In Staunstrup, J., editor, *Formal Methods for VLSI Design*, pages 13–70. Elsevier Science Publications.

Jones, G. and Sheeran, M. (1993). Designing arithmetic circuits by refinement in Ruby. In Bird, R. S., Morgan, C. C., and Woodcock, J. C. P., editors, *Mathematics of Program Construction*, Volume 669 of *Lecture Notes in Computer Science*, pages 208–232. Springer-Verlag.

Jones, M. P. (1994). The implementation of the gofer functional programming system. Research report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA. Available from URL `http://www.cs.nott.ac.uk/Department/Staff/mpj/`.

Jones, M. P. (1995). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, *5*(1), 1–35.

Karp, R. M. and Held, M. (1967). Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, *15*(3), 693–718.

Kawahara, Y. (1973a). Notes on the universality of relational functors. *Memoirs of the Faculty of Science, Kyushu University, Series A, Mathematics*, *27*(3), 275–289.

Kawahara, Y. (1973b). Relations in categories with pullbacks. *Memoirs of the Faculty of Science, Kyushu University, Series A, Mathematics*, *27*(1), 149–173.

Kawahara, Y. (1990). Pushout-complements and basic concepts of grammars in toposes. *Theoretical Computer Science*, *77*(3), 267–289.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language* (Second edition). Software series. Prentice Hall.

Kieburtz, R. B. and Lewis, J. (1995). Programming with algebras. In Jeuring, J. T. and Meijer, E., editors, *Advanced Functional Programming*, Volume 925 of *Lecture Notes in Computer Science*, pages 267–307. Springer-Verlag.

Kleene, S. C. (1952). *Introduction to Metamathematics*, Volume 1 of *Bibliotheca Mathematica*. North-Holland.

Knapen, E. (1993). Relational programming, program inversion and the derivation of parsing algorithms. Computing science notes, Department of Mathematics and Computing Science, Eindhoven University of Technology. Available from URL `http://www.win.tue.nl/win/cs/wp/papers/papers.html`.

Knaster, B. (1928). Un théorème sur les fonctions d'ensembles. *Annales de la Societé Polonaise de Mathematique*, *6*, 133–134.

Knuth, D. E. and Plass, M. F. (1981). Breaking paragraphs into lines. *Software: Practice and Experience*, *11*, 1119–1184.

Knuth, D. E. (1990). A simple program whose proof isn't. In Feijen, W., Gries, D., and Van Gasteren, A. J. M., editors, *Beauty is Our Business – A Birthday Salute to Edsger W. Dijkstra*, pages 233–242. Springer-Verlag.

Kock, A. (1972). Strong functors and monoidal monads. *Archiv für Mathematik*, *23*, 113–120.

Korte, B., Lovasz, L., and Schrader, R. (1991). *Greedoids*, Volume 4 of *Algorithms and combinatorics*. Springer-Verlag.

Lambek, J. and Scott, P. J. (1986). *Introduction to Higher Order Categorical Logic*, Volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press.

Lambek, J. (1968). A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, *103*, 151–161.

Lawler, E. L. (1973). Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, *19*(5), 544–546.

Lawvere, F. W. (1966). The category of categories as a foundation for mathematics. In Eilenberg, S., Harrison, D. K., Mac Lane, S., and Röhrl, H., editors, *Categorical Algebra*, pages 1–20. Springer-Verlag.

Lehmann, D. J. and Smyth, M. B. (1981). Algebraic specification of data types: a synthetic approach. *Mathematical Systems Theory*, *14*(2), 97–139.

Mac Lane, S. and Moerdijk, I. (1992). *Sheaves in Geometry and Logic: A First Introduction to Topos Theory.* Universitext. Springer-Verlag.

Mac Lane, S. (1961). An algebra of additive relations. *Proceedings of the National Academy of Sciences*, *47*, 1043–1051.

Maddux, R. D. (1991). The origin of relation algebras in the development and axiomatization of the calculus of relations. *Studia Logica*, *50*(3–4), 421–455.

Malcolm, G. R. (1990a). *Algebraic data types and program transformation.* Ph.D. thesis, Department of Computing Science, Groningen University, The Netherlands.

Malcolm, G. R. (1990b). Data structures and program transformation. *Science of Computer Programming*, *14*(2–3), 255–279.

Manes, E. G. and Arbib, M. A. (1986). *Algebraic Approaches to Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag.

Manes, E. G. (1975). *Algebraic Theories*, Volume 26 of *Graduate Texts in Mathematics.* Springer-Verlag.

Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations.* Interscience Series in Discrete Mathematics and Optimization. Wiley.

Martin, U. and Nipkow, T. (1990). Automating Squiggol. In Broy, M. and Jones, C. B., editors, *Programming Concepts and Methods*, pages 223–236. North-Holland.

Martin, C. E. (1991). *Preordered categories and predicate transformers.* D.Phil. thesis, Computing Laboratory, Oxford, UK.

Mathematics of Program Construction Group. (1995). Fixed-point calculus. *Information Processing Letters*, *53*, 131–136.

McLarty, C. (1992). *Elementary Categories, Elementary Toposes*, Volume 21 of *Oxford Logic Guides.* Clarendon Press.

Meertens, L. (1987). Algorithmics – towards programming as a mathematical activity. In De Bakker, J. W., Hazewinkel, M., and Lenstra, J. K., editors, *Mathematics and Computer Science*, Volume 1 of *CWI Monographs*, pages 3–42. North-Holland.

Meertens, L. (1989). Constructing a calculus of programs. In Van de Snepscheut, J. L. A., editor, *Mathematics of Program Construction*, Volume 375 of *Lecture Notes in Computer Science*, pages 66–90. Springer-Verlag.

Meertens, L. (1992). Paramorphisms. *Formal Aspects of Computing*, *4*(5), 413–424.

Meijer, E. and Hutton, G. (1995). Bananas in space: extending fold and unfold to exponential types. In Peyton-Jones, S., editor, *Functional Programming Languages and Computer Architecture*, pages 324–333. Association for Computing Machinery.

Meijer, E., Fokkinga, M., and Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. In Hughes, J., editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, Volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag.

Meijer, E. (1992). *Calculating compilers.* Ph.D. thesis, University of Nijmegen, The Netherlands.

Mikkelsen, C. J. (1976). Lattice theoretic and logical aspects of elementary topoi. Various Publications Series 25, Matematisk Institut, Aarhus Universitet, Denmark.

Mili, A., Desharnais, J., and Mili, F. (1987). Relational heuristics for the design of deterministic programs. *Acta Informatica*, *24*(3), 239–276.

Mili, A., Desharnais, J., and Mili, F. (1994). *Computer Program Construction.* Oxford University Press.

Mili, A. (1983). A relational approach to the design of deterministic programs. *Acta Informatica*, *20*(4), 315–328.

Mitchell, J. C. and Ščedrov, A. (1993). Notes on sconing and relators. In Boerger, E., editor, *Computer Science Logic '92, Selected Papers*, Volume 702 of *Lecture Notes in Computer Science*, pages 352–378.

Mitten, L. G. (1964). Composition principles for synthesis of optimal multistage processes. *Operations Research*, *12*, 610–619.

Moggi, E. (1991). Notions of computation and monads. *Information and Computation*, *93*(1), 55–92.

Möller, B. and Russling, M. (1994). Shorter paths to graph algorithms. *Science of Computer Programming*, *22*(1–2), 157–180.

Möller, B. (1991). Relations as a program development language. In Möller, B., editor, *Constructing Programs from Specifications*, pages 373–397. North-Holland.

Möller, B. (1993). Derivation of graph and pointer algorithms. In Möller, B., Partsch, H., and Schuman, S., editors, *Formal Program Development*, Volume 755 of *Lecture Notes in Computer Science*, pages 123–160. Springer-Verlag.

Morgan, C. C. (1993). The cuppest capjunctive capping. In Roscoe, A. W., editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, International Series in Computer Science, pages 317–332. Prentice Hall.

Naumann, D. A. (1994). A recursion theorem for predicate transformers on inductive data types. *Information Processing Letters*, *50*(6), 329–336.

Ning, M. Z. (1997). *Functional programming and combinatorial optimisation.* Ph.D. thesis, Computing Laboratory, Oxford, UK. forthcoming.

Partsch, H. A. (1986). Transformational program development in a particular problem domain. *Science of Computer Programming*, *7*(2), 99–241.

Partsch, H. A. (1990). *Specification and Transformation of Programs – A Formal Approach to Software Development.* Texts and Monographs in Computer Science. Springer-Verlag.

Paterson, R. (1988). *Reasoning about functional programs.* Ph.D. thesis, University of Queensland, Brisbane.

Paulson, L. (1991). *ML for the working programmer.* Cambridge University Press.

Peirce, C. S. (1870). Description of a notation for the logic of relatives, resulting from an amplification of the conceptions of Boole's calculus of logic. *Memoirs of the American Academy of Sciences*, *9*, 317–378. Reprinted in (Peirce 1933).

Peirce, C. S. (1933). *Collected Papers.* Harvard University Press.

Pettorossi, A. and Burstall, R. M. (1983). Deriving very efficient algorithms for evaluating linear recurrence relations using the program transformation technique. *Acta Informatica*, *18*(2), 181–206.

Pettorossi, A. (1984). Methodologies for transformations and memoing in applicative languages. Ph.D. thesis CST-29-84, University of Edinburgh, Scotland.

Pettorossi, A. (1985). Towers of Hanoi problems: deriving iterative solutions by program transformations. *BIT*, *25*(2), 327–334.

Pierce, B. C. (1991). *Basic category theory for computer scientists.* Foundations of Computing Series. MIT Press.

Pratt, V. R. (1992). Origins of the calculus of binary relations. In *Logic in Computer Science*, pages 248–254. IEEE Computer Society Press.

Puppe, D. (1962). Korrespondenzen in abelschen kategorien. *Mathematische Annalen*, *148*, 1–30.

Reade, C. (1988). *Elements of Functional Programming.* International computer science series. Addison-Wesley.

Reingold, E. M., Nievergelt, J., and Deo, N. (1977). *Combinatorial Algorithms: Theory and Practice.* Prentice Hall.

Rietman, F. J. (1995). *A relational calculus for the design of distributed algorithms.* Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands.

Riguet, J. (1948). Relations binaires, fermeture, correspondances de Galois. *Bulletin de la Société Mathématique de France*, *76*, 114–155.

Russling, M. (1995). A general scheme for breadth-first graph traversal. In Möller, B., editor, *Mathematics of Program Construction*, Volume 947 of *Lecture Notes in Computer Science*, pages 380–398. Springer-Verlag.

Rydeheard, D. E. and Burstall, R. M. (1988). *Computational Category Theory.* International Series in Computer Science. Prentice Hall.

Sanderson, J. G. (1980). *A Relational Theory of Computing*, Volume 82 of *Lecture Notes in Computer Science.* Springer-Verlag.

Schmidt, G. W. and Ströhlein, T. (1993). *Relations and Graphs: Discrete Mathematics for Computer Scientists.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag.

Schmidt, G. W., Berghammer, R., and Zierer, H. (1989). Symmetric quotients and domain construction. *Information Processing Letters*, *33*(3), 163–168.

Schoenmakers, B. (1992). Inorder traversal of a binary heap and its inversion in optimal time and space. In Bird, R. S., Morgan, C. C., and Woodcock, J. C. P., editors, *Mathematics of Program Construction*, Volume 669 of *Lecture Notes in Computer Science*, pages 291–301. Springer-Verlag.

Schröder, E. (1895). *Vorlesungen über die Algebra der Logik (Exakte Logik). Dritter Band: Algebra und Logik der Relative.* Teubner, Leipzig.

Sheard, T. and Fegaras, L. (1993). A fold for all seasons. In *Functional Programming Languages and Computer Architecture*, pages 233–242. Association for Computing Machinery.

Sheeran, M. (1987). Relations + higher-order functions = hardware descriptions. In Proebster, W. E. and Reiner, E., editors, *VLSI and Computers*, pages 303–306. IEEE.

Sheeran, M. (1990). Categories for the working hardware designer. In Leeser, M. and Brown, G., editors, *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects. Cornell University 1989*, Volume 408 of *Lecture Notes in Computer Science*, pages 380–402. Springer-Verlag.

Skillicorn, D. B. (1995). *Foundations of Parallel Programming*, Volume 6 of *Cambridge International Series on Parallel Computation*. Cambridge University Press.

Smith, D. R. and Lowry, M. R. (1990). Algorithm theories and design tactics. *Science of Computer Programming*, *14*(2–3), 305–321.

Smith, D. R. (1985). Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, *27*(1), 43–96.

Smith, D. R. (1987). Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, *18*, 213–229.

Smith, D. R. (1990). KIDS: a semiautomatic program development system. *IEEE Transactions on Software Engineering*, *16*(9), 1024–1043.

Smith, D. R. (1991). Structure and design of problem reduction generators. In Möller, B., editor, *Constructing Programs from Specifications*, pages 91–124. North-Holland.

Smith, D. R. (1993). Constructing specification morphisms. *Journal of Symbolic Computation*, *15*, 571–606.

Sniedovich, M. (1986). A new look at Bellman's principle of optimality. *Journal of Optimization Theory and Applications*, *49*(1), 161–176.

Spivey, M. (1989). A categorical approach to the theory of lists. In Van de Snepscheut, J. L. A., editor, *Mathematics of Program Construction*, Volume 375 of *Lecture Notes in Computer Science*, pages 399–408. Springer-Verlag.

Takano, A. and Meijer, E. (1995). Shortcut deforestation in calculational form. In Peyton-Jones, S., editor, *Functional Programming Languages and Computer Architecture*, pages 306–313. Association for Computing Machinery.

Tarski, A. (1941). On the calculus of relations. *Journal of Symbolic Logic*, *6*(3), 73–89.

Tarski, A. (1955). A lattice–theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, *5*, 285–309.

Taylor, P. (1994). Commutative diagrams in TEX(version 4). Available from URL `http://theory.doc.ic.ac.uk/tex/contrib/Taylor/diagrams`.

Von Karger, B. and Hoare, C. A. R. (1995). Sequential calculus. *Information Processing Letters*, *53*(3), 123–130.

Wadler, P. (1987). Views: a way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*, pages 307–313. Association for Computing Machinery.

Wadler, P. (1989). Theorems for free!. In *Functional Programming Languages and Computer Architecture*, pages 347–359. Association for Computing Machinery.

Walters, R. F. C. (1989). Data types in distributive categories. *Bulletin of the Australian Mathematical Society*, *40*(1), 79–82.

Walters, R. F. C. (1992a). *Categories and Computer Science*, Volume 28 of *Cambridge Computer Science Texts*. Cambridge University Press.

Walters, R. F. C. (1992b). An imperative language based on distributive categories. *Mathematical Structures in Computer Science*, *2*(3), 249–256.

Wickström, A. (1987). *Functional Programming Using Standard ML*. International Series in Computer Science. Prentice Hall.

Williams, J. H. (1982). On the development of the algebra of functional programs. *ACM Transactions on Programming Languages and Systems*, *4*(4), 733–757.

Yao, F. F. (1980). Efficient dynamic programming using quadrangle inequalities. In *Theory of Computing*, pages 429–435. Association for Computing Machinery.

Yao, F. F. (1982). Speed–up in dynamic programming. *SIAM Journal on Algebraic and Discrete Methods*, *3*(4), 532–540.

Index

PEYTON JONES, S. and LESTER, D., *Implementing Functional Languages*
POTTER, B., SINCLAIR, J. and TILL, D., *An Introduction to Formal Specification and Z* (2nd edn)
RABHI, F.A., and LAPALME, G., *Designing Algorithms with Functional Languages*
ROSCOE, A.W. (ed.), *A Classical Mind: Essays in honour of C.A.R. Hoare*
ROZENBERG, G., and SALOMAA, A., *Cornerstones of Undecidability*
RYDEHEARD, D.E. and BURSTALL, R.M., *Computational Category Theory*
SHARP, R., *Principles of Protocol Design*
SLOMAN, M. and KRAMER, J., *Distributed Systems and Computer Networks*
SPIVEY, J.M., *An Introduction to Logic Programming through Prolog*
SPIVEY, J.M., *The Z. Notation: A reference manual (2nd edn)*
TENNENT, R.D., *Semantics of Programming Languages*
WATT, D.A., *Programming Language Concepts and Paradigms*
WATT, D.A., *Programming Language Processors*
WATT, D.A., *Programming Language Syntax and Semantics*
WATT, D.A., WICHMANN, B.A. and FINDLAY, W., *ADA: Language and methodology*
WELSH, J. and ELDER, J., *Introduction to Modula-2*
WELSH, J. and ELDER, J., *Introduction to Pascal (3rd edn)*
WIKSTRÖM, Å., *Functional Programming Using Standard ML*
WOODCOCK, J. and DAVIES, J., *Using Z: Specification, refinement, and proof*