J.N. Oliveira

PROGRAM DESIGN BY CALCULATION

(DRAFT / Last update: April 2017)

University of Minho

(in preparation)

Contents

List of Exercises

Preamble

This textbook in preparation has arisen from the author's research and teaching experience. Its main aim is to provide software practitioners with a calculational approach to the design of software artifacts ranging from simple algorithms and functions to the specification and realization of information systems. Put in other words, the book invites software designers to raise standards and adopt mature development techniques found in other engineering disciplines, which (as a rule) are rooted on a sound mathematical basis so as to enable algebraic reasoning.

It is interesting to note that while coining the phrase *software engineering* in the 1960s, our colleagues of the time were already promising such high quality standards. The terminology seems to date from the Garmisch NATO conference in 1968, from whose report [31] the following excerpt is quoted:

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

Provocative or not, the need for sound theoretical foundations has clearly been under concern since the very beginning of the discipline. However, how "scientific" do such foundations turn out to be, now that nearly five decades have since elapsed?

Thirty years later, Richard Bird and Oege de Moore published a textbook [6] in whose preface C.A.R. Hoare writes:

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

The formulæ and equations mentioned in this quotation are those of the discipline known as the *Algebra of Programming*. Many others have contributed to this body of knowledge, notably Roland Backhouse and his colleagues at Eindhoven and Nottingham, see eg. [1] and [2], Jeremy Gibbons and Ralf Hinze at Oxford see e.g. [14], among many others. Unfortunately, references[1, 2] are still unpublished.

When the author of this draft textbook decided to teach *Algebra of Programming* to 2nd year students of the Minho degrees in computer science, back to 1998, he found textbook [6] too difficult for the students to follow, mainly because of its explicit categorial (allegorical) flavour. So he decided to start writing slides and notes helping the students to read the book. Eventually, such notes became chapters 2 to 4 of the current version of the monograph. The same procedure was taken when teaching the relational approach of [6] to 4th and 5th year students (today at master level). This draft book is by and large incomplete, most chapters being still in *slide form*[1]. Such half-finished chapters are omitted from the current print-out.

Altogether, the idea is to show that software engineering and, in particular, computer progamming can adopt the *scientific method* as other branches of engineering do.

Braga, at University of Minho, April 2017

José N. Oliveira

[1] For the slides which eventually will lead to the second part of this book see technical report [33]. The third part will address a linear algebra of programming intended for quantitative reasoning about software. This is even less stable, but a number of papers exist about the topic, see the bibliography.

Chapter 1

Introduction

not given in the current version of this textbook

Part I

Calculating with Functions

Chapter 2

An Introduction to Pointfree Programming

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

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

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

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

Many technical issues are deliberately ignored and deferred to later chapters. Most programming examples will be provided in the HASKELL functional pro-

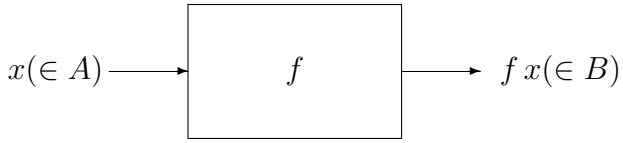
gramming language. Appendix A.1 includes the listings of some HASKELL modules which complement the HASKELL *Standard Prelude* and help to "animate" the main concepts introduced in this chapter.

2.1 Introducing functions and types

The definition of a function

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

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



From another viewpoint, f can be regarded as a kind of "contract": f *commits itself* to producing a B-value provided it is supplied with an A-value. How is such a value produced? In many situations one wishes to ignore it because one is just *using* function f. In others, however, one may want to inspect the internals of the "black box" in order to know the function's *computation rule*. For instance,

$$\begin{array}{rcl} succ & : & \mathbb{N} \to \mathbb{N} \\ succ\, n & \stackrel{\text{def}}{=} & n+1 \end{array}$$

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

In programming terminology one says that $succ$ and sq have the same "type". Types play a prominent rôle in functional programming (as they do in other programming paradigms). Informally, they provide the "glue", or interfacing material, for putting functions together to obtain more complex functions. Formally, a

"type checking" discipline can be expressed in terms of compositional rules which check for functional expression wellformedness.

It has become standard to use arrows to denote function signatures or function types, recall (2.1). In this book the following variants will be used interchangeably to denote the fact that function f accepts arguments of type A and produces results of type B: $f : B \leftarrow A$, $f : A \rightarrow B$, $B \xleftarrow{f} A$ or $A \xrightarrow{f} B$. This corresponds to writing $f :: a \rightarrow b$ in the HASKELL functional programming language, where type variables are denoted by lowercase letters. A will be referred to as the *domain* of f and B will be referred to as the *codomain* of f. Both A and B are symbols which denote sets of values, very often called *types*.

2.2 Functional application

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

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

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

2.3 Functional equality and composition

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

$$distance\, (2 \times t) = 2 \times (distance\, t) \tag{2.2}$$

holds. Later on, we could learn from her or him that the same property can be restated as $distance\, (twice\, t) = twice\, (distance\, t)$, by introducing function

$twice\, x \stackrel{\text{def}}{=} 2 \times x$. Or even simply as

$$distance \cdot twice = twice \cdot distance \tag{2.3}$$

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

$$\begin{array}{ccc} \mathbb{R} & \xleftarrow{twice} & \mathbb{R} \\ {\scriptstyle distance}\downarrow & & \downarrow{\scriptstyle distance} \\ \mathbb{R} & \xleftarrow[twice]{} & \mathbb{R} \end{array} \tag{2.4}$$

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

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

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

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

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

Throughout this book we will adopt both, depending upon the context. Our chosen programming medium — HASKELL — blends the pointwise and pointfree programming styles in a quite successful way. In order to switch from one to the

other, we need two "bridges": one lifting equality to the functional level and the other lifting function application.

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

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

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

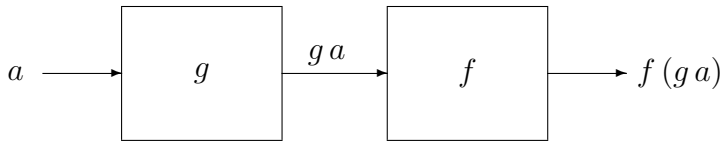
Concerning application, the pointfree style replaces it by the more generic concept of functional *composition* suggested by function-arrow chaining: wherever two functions are such that the target type of one of them, say $B \xleftarrow{g} A$ is the same as the source type of the other, say $C \xleftarrow{f}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

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

This situation is pictured by the following arrow-diagram

$$\begin{array}{ccc} B & \xleftarrow{\;g\;} & A \\ {\scriptstyle f}\downarrow & \swarrow_{f \cdot g} & \\ C & & \end{array} \tag{2.7}$$

or by block-diagram



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

$$\frac{C \xleftarrow{f} B \qquad B \xleftarrow{g} A}{C \xleftarrow{f \cdot g} A}$$

Composition is certainly the most basic of all functional combinators. It is the first kind of "glue" which comes to mind when programmers need to combine, or chain functions (or processes) to obtain more elaborate functions (or processes) [1]. This is because of one of its most relevant properties,

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

which shares the pattern of, for instance

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

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

2.4 Identity functions

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

$$f\, a = a$$

In this case $B = A$, of course, and f is said to be the *identity* function on A:

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

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

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

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

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

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

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

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

$$\begin{array}{ccc} A & \xleftarrow{id} & A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f} \\ B & \xleftarrow[id]{} & B \end{array} \tag{2.11}$$

Note the graphical analogy of diagrams (2.4) and (2.11). Diagrams of this kind are very common and express important (and rather 'natural') properties of functions, as we shall see further on.

2.5 Constant functions

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

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

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

The following property defines constant functions at pointfree level,

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

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

$$\begin{array}{ccc} C & \xleftarrow{\underline{c}} & A \\ {\scriptstyle id}\downarrow & & \downarrow{\scriptstyle f} \\ C & \xleftarrow[\underline{c}]{} & B \end{array} \tag{2.14}$$

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

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

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

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

Exercise 2.1. *The* HASKELL Prelude *provides for constant functions: you write* const c *for* $\underline{c}$*. Check that* HASKELL *assigns the same type to expressions* $f \cdot (\mathsf{const}\ \ c)$ *and* $\mathsf{const}\ \ (f\ \ c)$*, for every* f *and* c*. What else can you say about these functional expressions? Justify.*
□

2.6 Monics and epics

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

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

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

$$B \xleftarrow{f} A \underset{k}{\overset{h}{\leftleftarrows}} C$$

(we say that f is "cancellable on the left"). It is easy to check that "the" identity function is monic,

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

and that any constant function $\underline{c}$ is not monic:

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

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

Functions which do not ignore values of their codomain are called *epics* (or *surjective* functions) and obey the following property: $A \xleftarrow{f} B$ is *epic* if, for every pair of functions $C \xleftarrow{h,k} A$, if $h \cdot f = k \cdot f$ then $h = k$, *cf.* diagram

$$C \underset{h}{\overset{k}{\leftleftarrows}} A \xleftarrow[f]{} B$$

(we say that f is "cancellable on the right").

As expected, identity functions are epic:

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

Exercise 2.2. *Under what circumstances is a constant function epic? Justify.*
□

2.7 Isos

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

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

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

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

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

$$Weekday = \{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday\}$$

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

$$\begin{aligned} &f : Weekday \to 7 \\ &f\ Monday = 1 \\ &f\ Tuesday = 2 \\ &f\ Wednesday = 3 \\ &f\ Thursday = 4 \\ &f\ Friday = 5 \\ &f\ Saturday = 6 \\ &f\ Sunday = 7 \end{aligned}$$

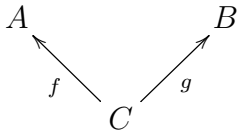
is iso (guess f°). Clearly, $f\,d = i$ means "d is the i-th day of the week". But note that function $g\,d \stackrel{\text{def}}{=} rem(f\,d, 7) + 1$ is also an iso between Weekday and 7. While f regards $Monday$ the first day of the week, g places $Sunday$ in that position. Both f and g are witnesses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

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

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

2.8 Gluing functions which do not compose — products

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



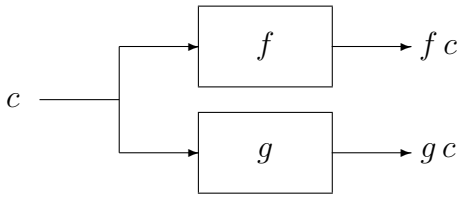
Because f and g share the same domain, their outputs can be paired, that is, we may write ordered pair $(f\,c, g\,c)$ for each $c \in C$. Such pairs belong to the Cartesian product of A and B, that is, to the set

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

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

$$\begin{array}{rcl} \langle f, g\rangle & : & C \to A \times B \\ \langle f, g\rangle\, c & \stackrel{\text{def}}{=} & (f\,c, g\,c) \end{array} \tag{2.18}$$

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



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

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B \tag{2.19}$$

defined by

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

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

$$\pi_1 \cdot \langle f, g\rangle = f \quad \text{and} \quad \pi_2 \cdot \langle f, g\rangle = g \tag{2.20}$$

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

$$\begin{array}{ccccc} A & \xleftarrow{\pi_1} & A \times B & \xrightarrow{\pi_2} & B \\ & {\scriptstyle f}\nwarrow & \uparrow{\scriptstyle \langle f,g\rangle} & \nearrow{\scriptstyle g} & \\ & & C & & \end{array} \tag{2.21}$$

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

$$\frac{A \xleftarrow{f} C \qquad B \xleftarrow{g} C}{A \times B \xleftarrow{\langle f,g\rangle} C}$$

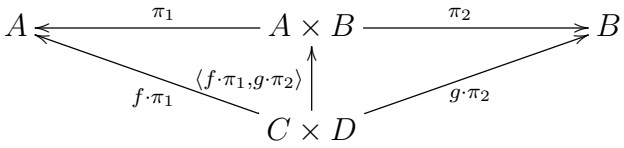
A *split* arises wherever two functions do not compose but share the same domain. What about gluing two functions which fail such a requisite, *e.g.*

$$\frac{A \xleftarrow{f} C \qquad B \xleftarrow{g} D}{\ldots ?}$$

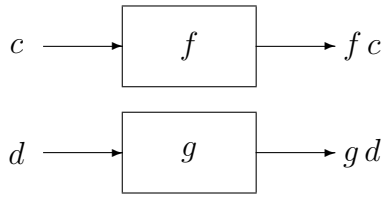
The $\langle f, g \rangle$ *split* combination does not work any more. Nevertheless, a way to "bridge" the domains of f and g, C and D respectively, is to regard them as targets of the projections π_1 and π_2 of $C \times D$:

$$\begin{array}{ccccc} A & \xleftarrow{\pi_1} & A \times B & \xrightarrow{\pi_2} & B \\ {\scriptstyle f}\uparrow & & & & \uparrow{\scriptstyle g} \\ C & \xleftarrow{\pi_1} & C \times D & \xrightarrow{\pi_2} & D \end{array}$$

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



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



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

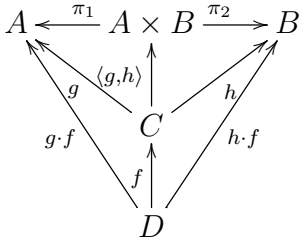
$$f \times g \stackrel{\text{def}}{=} \langle f \cdot \pi_1, g \cdot \pi_2 \rangle \tag{2.22}$$

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

$$\frac{\begin{array}{c} A \xleftarrow{f} C \\ B \xleftarrow{g} D \end{array}}{A \times B \xleftarrow{f \times g} C \times D} \tag{2.23}$$

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

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

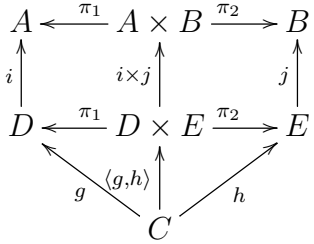


$$\langle g, h\rangle \cdot f = \langle g \cdot f, h \cdot f\rangle \qquad (2.24)$$

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

$$\begin{aligned}
& (i \times j) \cdot \langle g, h\rangle \\
= & \quad \{ \text{ by (2.22) } \} \\
& \langle i \cdot \pi_1, j \cdot \pi_2\rangle \cdot \langle g, h\rangle \\
= & \quad \{ \text{ by } \times\text{-fusion (2.24) } \} \\
& \langle (i \cdot \pi_1) \cdot \langle g, h\rangle, (j \cdot \pi_2) \cdot \langle g, h\rangle\rangle \\
= & \quad \{ \text{ by (2.8) } \} \\
& \langle i \cdot (\pi_1 \cdot \langle g, h\rangle), j \cdot (\pi_2 \cdot \langle g, h\rangle)\rangle \\
= & \quad \{ \text{ by } \times\text{-cancellation (2.20) } \} \\
& \langle i \cdot g, j \cdot h\rangle
\end{aligned}$$

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



$$(i \times j) \cdot \langle g, h\rangle = \langle i \cdot g, j \cdot h\rangle \qquad (2.25)$$

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

$$i \cdot \pi_1 = \pi_1 \cdot (i \times j) \tag{2.26}$$
$$j \cdot \pi_2 = \pi_2 \cdot (i \times j) \tag{2.27}$$

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

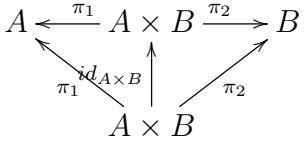
$$(g \cdot h) \times (i \cdot j) = (g \times i) \cdot (h \times j) \tag{2.28}$$

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

$$id_A \times id_B = id_{A \times B} \tag{2.29}$$

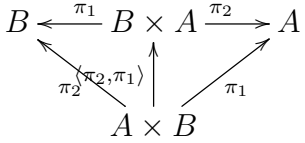
These two properties will be identified as the *functorial properties* of product. Once again, this choice of terminology will be explained later on.

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



$$\langle \pi_1, \pi_2 \rangle = id_{A \times B} \tag{2.30}$$

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



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

$$\begin{array}{ll} & swap \cdot swap \\ = & \quad \{ \text{ by definition } swap \stackrel{\text{def}}{=} \langle \pi_2, \pi_1 \rangle \ \} \end{array}$$

$$
\begin{array}{ll}
 & \langle \pi_2, \pi_1 \rangle \cdot swap \\
= & \quad \{ \text{ by } \times\text{-fusion (2.24) } \} \\
 & \langle \pi_2 \cdot swap, \pi_1 \cdot swap \rangle \\
= & \quad \{ \text{ definition of } swap \text{ twice } \} \\
 & \langle \pi_2 \cdot \langle \pi_2, \pi_1 \rangle, \pi_1 \cdot \langle \pi_2, \pi_1 \rangle \rangle \\
= & \quad \{ \text{ by } \times\text{-cancellation (2.20) } \} \\
 & \langle \pi_1, \pi_2 \rangle \\
= & \quad \{ \text{ by } \times\text{-reflexion (2.30) } \} \\
 & id
\end{array}
$$

Therefore, $swap$ is iso and establishes the following isomorphism

$$A \times B \;\; \cong \;\; B \times A \tag{2.31}$$

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

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

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

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

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

Isomorphism (2.31) can be re-interpreted in this context as a guarantee that *one does not lose (or gain) anything in swapping fields in record datatypes*. C or PASCAL programmers know also that record-field nesting has the same status,

that is to say that, for instance, datatype

```
record                                          record
    F: A;                                           F: record
    S: record                                               F: A;
           F: B;          is abstractly the same as         S: B
           S: C;                                       end;
       end                                          S: C;
end;                                            end;
```

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

$$A \times (B \times C) \;\cong\; (A \times B) \times C \tag{2.32}$$

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

$$assocr \;\stackrel{\text{def}}{=}\; \langle \pi_1 \cdot \pi_1, \langle \pi_2 \cdot \pi_1, \pi_2 \rangle \rangle \tag{2.33}$$

Section A.1 in the appendix lists an extension to the HASKELL *Standard Prelude* that makes isomorphisms such as $swap$ and $assocr$ available. In this module, the concrete syntax chosen for $\langle f, g \rangle$ is `split f g` and the one chosen for $f \times g$ is `f >< g`.

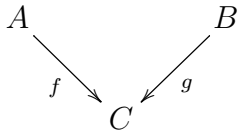
Exercise 2.3. *Show that $assocr$ is iso by conjecturing its inverse $assocl$ and proving that functional equality $assocr \cdot assocl = id$ holds.*
□

Exercise 2.4. *Rely on (2.22) to prove properties (2.28) and (2.29).*
□

2.9 Gluing functions which do not compose — co-products

The *split* functional combinator arose in the previous section as a kind of glue for combining two functions which do not compose but share the same domain. The

"dual" situation of two non-composable functions $f : C \leftarrow A$ and $g : C \leftarrow B$ which however share the same codomain is depicted in



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

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

$$A \xrightarrow{i_1} A + B \xleftarrow{i_2} B$$

The values of $A + B$ can be thought of as "copies" of A or B values which are "stamped" with different tags in order to guarantee that values which are simultaneously in A and B do not get mixed up. The tagging functions i_1 and i_2 are called *injections*:

$$i_1\, a = (t_1, a) \quad , \quad i_2\, b = (t_2, b) \tag{2.34}$$

Knowing the exact values of tags t_1 and t_2 is not essential to understanding the concept of a disjoint union. It suffices to know that i_1 and i_2 tag differently and consistently. For instance, the following realizations of $A + B$ in the C programming language,

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

or in PASCAL,

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

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

$$\mathbf{data}\ a + b = i_1\ a \mid i_2\ b$$

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

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

$$\begin{array}{rcl} [f,g] & : & A+B \longrightarrow C \\ [f,g]\,x & \stackrel{\text{def}}{=} & \left\{ \begin{array}{lcl} x = i_1\,a & \Rightarrow & f\,a \\ x = i_2\,b & \Rightarrow & g\,b \end{array} \right. \end{array} \tag{2.35}$$

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

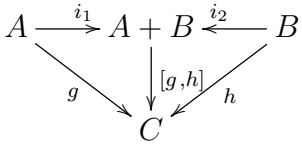
$$\begin{array}{ccccc} A & \xrightarrow{i_1} & A+B & \xleftarrow{i_2} & B \\ & {\scriptstyle f}\searrow & \downarrow{\scriptstyle [f,g]} & \swarrow{\scriptstyle g} & \\ & & C & & \end{array} \tag{2.36}$$

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

$$f + g \stackrel{\text{def}}{=} [i_1 \cdot f, i_2 \cdot g] \tag{2.37}$$

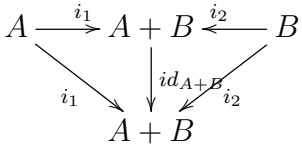
The following list of $+$-laws provides eloquent evidence of this duality:

$+$-cancellation :



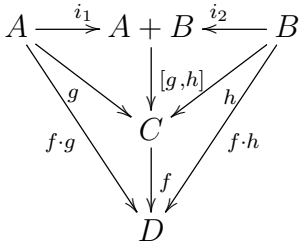
$$[g, h] \cdot i_1 = g \ , \ [g, h] \cdot i_2 = h \tag{2.38}$$

$+$-reflexion :



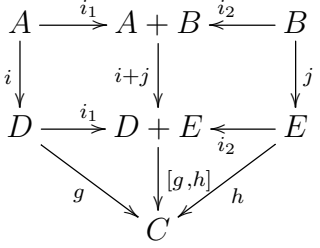
$$[i_1 , i_2] = id_{A+B} \tag{2.39}$$

$+$-fusion :



$$f \cdot [g, h] = [f \cdot g, f \cdot h] \tag{2.40}$$

$+$-absorption :



$$[g, h] \cdot (i + j) = [g \cdot i, h \cdot j] \tag{2.41}$$

$+$-functor :

$$(g \cdot h) + (i \cdot j) = (g + i) \cdot (h + j) \tag{2.42}$$

+-functor-id :

$$id_A + id_B = id_{A+B} \tag{2.43}$$

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

$$\frac{C \xleftarrow{f} A \quad C \xleftarrow{g} B}{C \xleftarrow{[f,g]} A + B} \qquad \frac{C \xleftarrow{f} A \quad D \xleftarrow{g} B}{C + D \xleftarrow{f+g} A + B} \tag{2.44}$$

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

$$A + B \quad \cong \quad B + A \tag{2.45}$$

holds.
□

Exercise 2.6. *Dualize (2.33), that is, write the iso which witnesses fact*

$$A + (B + C) \quad \cong \quad (A + B) + C \tag{2.46}$$

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

2.10 Mixing products and coproducts

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

a *split*. In the first case, both components of the *either* combinator are *splits*. In the latter, both components of the *split* combinator are *eithers*.

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

$$[\langle f, g\rangle, \langle h, k\rangle] \quad = \quad \langle [f, h], [g, k]\rangle \tag{2.47}$$

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

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

$$A \times (B + C) \xleftarrow{undistr} (A \times B) + (A \times C) \tag{2.48}$$

(pronounce $undistr$ as "un-distribute-right") which is defined by

$$undistr \quad \stackrel{\text{def}}{=} \quad [id \times i_1, id \times i_2] \tag{2.49}$$

and witnesses the fact that product distributes through coproduct:

$$A \times (B + C) \quad \cong \quad (A \times B) + (A \times C) \tag{2.50}$$

In this context, suppose that we know of three functions $D \xleftarrow{f} A$, $E \xleftarrow{g} B$ and $F \xleftarrow{h} C$. By (2.44) we infer $E + F \xleftarrow{g+h} B + C$. Then, by (2.23) we infer

$$D \times (E + F) \xleftarrow{f\times(g+h)} A \times (B + C) \tag{2.51}$$

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

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

— then we can write $\mathsf{F}(D, E, F) \xleftarrow{\mathsf{F}(f,g,h)} \mathsf{F}(A, B, C)$ for (2.51).

This kind of abstraction works for every combination of products and coproducts. For instance, if we now abstract the right-hand side of (2.48) via pattern

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

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

$$\begin{array}{ccc} \mathsf{F}(A,B,C) & \xleftarrow{undistr} & \mathsf{G}(A,B,C) \\ {\scriptstyle \mathsf{F}(f,g,h)}\downarrow & & \downarrow{\scriptstyle \mathsf{G}(f,g,h)} \\ \mathsf{F}(D,E,F) & & \mathsf{G}(D,E,F) \end{array}$$

which unfolds to

$$\begin{array}{ccc} A \times (B + C) & \xleftarrow{undistr} & (A \times B) + (A \times C) \\ {\scriptstyle f \times (g+h)}\downarrow & & \downarrow{\scriptstyle (f \times g)+(f \times h)} \\ D \times (E + F) & & (D \times E) + (D \times F) \end{array} \tag{2.52}$$

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

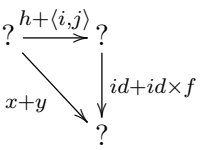
Exercise 2.7. *Apply the* exchange law *to* $undistr$.
□

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

$$\begin{array}{ccc} ? & & \\ {\scriptstyle [x\,,y]}\swarrow & \downarrow{\scriptstyle id+id\times f} & \\ ? & \xleftarrow[[k\,,g]]{} & ? \end{array}$$

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

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



□

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

2.11 Natural properties

Let us resume discussion about $undistr$ and the two other functions in diagram (2.52). What about using $undistr$ itself to close this diagram, at the bottom? Note that definition (2.49) works for D, E and F in the same way it does for A, B and C. (Indeed, the particular choice of symbols A, B and C in (2.48) was rather arbitrary.) Therefore, we get:

$$\begin{array}{ccc} A\times(B+C) & \xleftarrow{undistr} & (A\times B)+(A\times C) \\ {\scriptstyle f\times(g+h)}\downarrow & & \downarrow{\scriptstyle (f\times g)+(f\times h)} \\ D\times(E+F) & \xleftarrow[undistr]{} & (D\times E)+(D\times F) \end{array}$$

which expresses a very important property of $undistr$:

$$(f\times(g+h))\cdot undistr \;\;=\;\; undistr\cdot((f\times g)+(f\times h)) \qquad (2.53)$$

This is called the *natural* property of $undistr$. This kind of property (often called *free* instead of *natural*) is not a privilege of $undistr$. As a matter of fact, every function interfacing patterns such as F or G above will exhibit its own *natural* property. Furthermore, we have already quoted *natural* properties without mentioning it. Recall (2.10), for instance. This property (establishing id as the

unit of composition) is, after all, the *natural* property of id. In this case we have $\mathsf{F}\,\alpha = \mathsf{G}\,\alpha = \alpha$, as can be easily observed in diagram (2.11).

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

$$\begin{array}{ccc} A & \mathsf{F}\,A \xleftarrow{\phi} \mathsf{G}\,A \\ {\scriptstyle f}\downarrow & {\scriptstyle \mathsf{F}\,f}\downarrow \qquad\qquad \downarrow{\scriptstyle \mathsf{G}\,f} \\ B & \mathsf{F}\,B \xleftarrow[\phi]{} \mathsf{G}\,B \end{array} \qquad (\mathsf{F}\,f)\cdot\phi = \phi\cdot(\mathsf{G}\,f) \tag{2.54}$$

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

Diagram (2.54) corresponds to unary patterns F and G. As we have seen with $undistr$, other functions (g,h *etc.*) come into play for multiary patterns. A very important rôle will be assigned throughout this book to these F, G, *etc.* "shapes" or patterns which are shared by pointfree functional expressions and by their domain and codomain expressions. From chapter 3 onwards we will refer to them by their proper name — "functor" — which is standard in mathematics and computer science. Then we will also explain the names assigned to properties such as, for instance, (2.28) or (2.42).

Exercise 2.11. *Show that (2.26) and (2.27) are* natural *properties. Dualize these properties.* ***Hint****: recall diagram (2.41).*
□

Exercise 2.12. *Establish the* natural *properties of the $swap$ (2.31) and $assocr$ (2.33) isomorphisms.*
□

Exercise 2.13. *Draw the natural property of function*

$$\phi = swap\cdot(id\times swap)$$

as a diagram, that is, identify F and G in (2.54) for this case.
□

Exercise 2.14. *Repeat the previous exercise for*

$$\phi = coswap \cdot (swap + swap)$$

where $coswap = [i_2 , i_1]$.
□

2.12 Universal properties

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

Such properties bear standard names such as *cancellation*, *reflexion*, *fusion*, *absorption etc.*. Where do these come from? As a rule, for each combinator to be defined one has to define suitable constructions at "interface"-level [2], *e.g.* $A \times B$ and $A + B$. These are not chosen or invented at random: each is defined in a way such that the associated combinator is uniquely defined. This is assured by a so-called *universal property* from which the others can derived.

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

$$k = \langle f, g\rangle \quad \Leftrightarrow \quad \begin{cases} \pi_1 \cdot k = f \\ \pi_2 \cdot k = g \end{cases} \tag{2.55}$$

holds — recall diagram (2.21) — for all $A \times B \xleftarrow{k} C$. This equivalence states that $\langle f, g\rangle$ is the *unique* arrow satisfying the property on the right. In fact, read (2.55) in the $\Rightarrow$ direction and let k be $\langle f, g\rangle$. Then $\pi_1 \cdot \langle f, g\rangle = f$ and $\pi_2 \cdot \langle f, g\rangle = g$

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

will hold, meaning that $\langle f, g\rangle$ effectively obeys the property on the right. In other words, we have derived $\times$-cancellation (2.20). Reading (2.55) in the $\Leftarrow$ direction we understand that, if some k satisfies such properties, then it "has to be" the same arrow as $\langle f, g\rangle$.

It is easy to see other properties of $\langle f, g\rangle$ arising from (2.55). For instance, for $k = id$ we get $\times$-reflexion (2.30),

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

and for $k = \langle i, j\rangle \cdot h$ we get $\times$-fusion (2.24):

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

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

$$
\langle i, j\rangle = \langle f, g\rangle \ \Leftrightarrow \ \left\{ \begin{array}{l} i = f \\ j = g \end{array} \right. \tag{2.56}
$$

from universal property (2.55) — just let $k = \langle i, j\rangle$.

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

$$k = [f\,,g] \;\;\Leftrightarrow\;\; \left\{ \begin{array}{l} k \cdot i_1 = f \\ k \cdot i_2 = g \end{array} \right. \tag{2.57}$$

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

$$[i\,,j] = [f\,,g] \;\;\Leftrightarrow\;\; \left\{ \begin{array}{l} i = f \\ j = g \end{array} \right. \tag{2.58}$$

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

Exercise 2.15. *Prove the equality:*

$$[\langle f\,,\underline{k}\rangle\,,\langle g\,,\underline{k}\rangle] \;\;=\;\; \langle [f\,,g]\,,\underline{k}\rangle \tag{2.59}$$

□

Exercise 2.16. *Derive $+$-cancellation (2.38), $+$-reflexion (2.39) and $+$-fusion (2.40) from universal property (2.57). Then derive the* exchange law *(2.47) from the universal property of product (2.55) or coproduct (2.57).*

□

Exercise 2.17. *Function $coassocr = [id + i_1\,,i_2 \cdot i_2]$ is a witness of isomorphism $(A + B) + C \cong A + (B + C)$, from left to right. Calculate its converse $coassocl$ by solving the equation*

$$\underbrace{[x\,,[y\,,z]]}_{coassocl} \cdot coassocr = id \tag{2.60}$$

for x, y and z.

□

Exercise 2.18. *Let δ be a function of which you know that $\pi_1 \cdot \delta = id$ e $\pi_2 \cdot \delta = id$ hold. Show that necessarily δ satisfies the natural property $(f \times f) \cdot \delta = \delta \cdot f$.*

□

2.13 Guards and McCarthy's conditional

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

$$if\ \ (p\,x)\ \ then\ \ (g\,x)\ \ else\ \ (h\,x)$$

meaning

$$\begin{cases} p\,x & \Rightarrow & g\,x \\ \neg(p\,x) & \Rightarrow & h\,x \end{cases}$$

for some given predicate $\mathsf{Bool} \xleftarrow{p} A$, some "then"-function $B \xleftarrow{g} A$ and some "else"-function $B \xleftarrow{h} A$. Bool is the primitive datatype containing truth values FALSE and TRUE.

Can such expressions be written in the pointfree style? They can, provided we introduce the so-called "McCarthy conditional" functional form

$$p \to g, h$$

which is defined by

$$p \to g, h \ \stackrel{\text{def}}{=}\ [g, h] \cdot p? \tag{2.61}$$

a definition we can understand provided we know the meaning of the "$p?$" construct. We call $A + A \xleftarrow{p?} A$ a *guard*, or better, the guard associated to a given predicate $\mathsf{Bool} \xleftarrow{p} A$. Every predicate p gives birth to its own guard $p?$ which, at point-level, is defined as follows:

$$(p?)a \ = \ \begin{cases} p\,a & \Rightarrow & i_1\,a \\ \neg(p\,a) & \Rightarrow & i_2\,a \end{cases} \tag{2.62}$$

In a sense, guard $p?$ is more "informative" than p alone: it provides information about the outcome of testing p on some input a, encoded in terms of the coproduct injections (i_1 for a *true* outcome and i_2 for a *false* outcome, respectively) without losing the input a itself.

The following fact, which we will refer to as *McCarthy's conditional fusion law*, is a consequence of $+$-fusion (2.40):

$$f \cdot (p \to g, h) \ = \ p \to f \cdot g, f \cdot h \tag{2.63}$$

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

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

Exercise 2.19. *Prove that the following equality between two conditional expressions*

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

holds by rewriting it in the pointfree style (using the McCarthy's conditional combinator) and applying the exchange law *(2.47), among others.*

□

Exercise 2.20. *Prove law (2.63).*

□

Exercise 2.21. *From (2.61) and property*

$$p? \cdot f \quad = \quad (f + f) \cdot (p \cdot f)? \tag{2.64}$$

infer

$$(p \to f, g) \cdot h \quad = \quad (p \cdot h) \to (f \cdot h), (g \cdot h) \tag{2.65}$$

□

Exercise 2.22. *Prove that property*

$$\langle f, (p \to q, h)\rangle \quad = \quad p \to \langle f, q\rangle, \langle f, h\rangle \tag{2.66}$$

and its corollary

$$(p \to g, h) \times f \quad = \quad p \cdot \pi_1 \to g \times f, h \times f \tag{2.67}$$

hold, assuming the basic fact:

$$p \to f, f \quad = \quad f \tag{2.68}$$

□

2.14 Gluing functions which do not compose — exponentials

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

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

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

$$\begin{aligned} f_c \quad &: \quad A \to B \\ f_c\, a \quad &\stackrel{\text{def}}{=} \quad f(c, a) \end{aligned}$$

from f which combines nicely with g via the *split* combinator: $\langle f_c, g\rangle$ is well-defined and bears type $B \times D \leftarrow A$. For instance, suppose that $C = A$ and f is

the equality predicate $=$ on A. Then $\mathsf{Bool} \xleftarrow{=_c} A$ is the "equal to c" predicate on A values:

$$=_c\ a \quad \stackrel{\text{def}}{=} \quad a = c \tag{2.69}$$

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

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

$$B^A \quad \stackrel{\text{def}}{=} \quad \{g \mid g : B \leftarrow A\} \tag{2.70}$$

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

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

$$\begin{aligned} &ap :\ B \xleftarrow{ap} B^A \times A \\ &ap(f, a) \stackrel{\text{def}}{=} f\,a \end{aligned} \tag{2.71}$$

which applies a function f to an argument a.

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

$$f(c, a) \quad = \quad (\overline{f}\,c)a \tag{2.72}$$

Given c and a, both expressions denote the same value. But, in a sense, $\overline{f}$ is more tolerant than f: while the latter is binary and requires *both* arguments (c, a) to

become available before application, the former is happy to be provided with c first and with a later on, if actually required by the evaluation process.

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

$$k = \overline{f} \quad \Leftrightarrow \quad f = ap \cdot (k \times id) \tag{2.73}$$

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

Exponentials cancellation :

$$\begin{array}{ccc} B^A & & \\ {\scriptstyle \overline{f}}\uparrow & & \\ C & & \end{array} \qquad \begin{array}{ccc} B^A \times A & \xrightarrow{ap} & B \\ {\scriptstyle \overline{f}\times id}\uparrow & \nearrow_{f} & \\ C \times A & & \end{array} \qquad f = ap \cdot (\overline{f} \times id) \tag{2.74}$$

Exponentials reflexion :

$$\begin{array}{ccc} B^A & & \\ {\scriptstyle id_{B^A}}\uparrow & & \\ B^A & & \end{array} \qquad \begin{array}{ccc} B^A \times A & \xrightarrow{ap} & B \\ {\scriptstyle id_{B^A}\times id_A}\uparrow & \nearrow_{ap} & \\ B^A \times A & & \end{array} \qquad \overline{ap} = id_{B^A} \tag{2.75}$$

Exponentials fusion :

$$\begin{array}{ccc} B^A & & \\ {\scriptstyle \overline{g}}\uparrow & & \\ C & & \\ {\scriptstyle f}\uparrow & & \\ D & & \end{array} \qquad \begin{array}{ccc} B^A \times A & \xrightarrow{ap} & B \\ {\scriptstyle \overline{g}\times id}\uparrow & \nearrow_{g} & \uparrow \\ C \times A & & \nearrow_{g\cdot(f\times id)} \\ {\scriptstyle f\times id}\uparrow & & \\ D \times A & & \end{array} \qquad \overline{g \cdot (f \times id)} = \overline{g} \cdot f \tag{2.76}$$

Note that the cancellation law is nothing but fact (2.72) written in the pointfree style.

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

$$\begin{array}{ccc} D^A \times A & \xrightarrow{ap} & D \\ {\scriptstyle \overline{f\cdot ap}\times id}\uparrow & & \uparrow{\scriptstyle f} \\ B^A \times A & \xrightarrow{ap} & B \end{array}$$

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

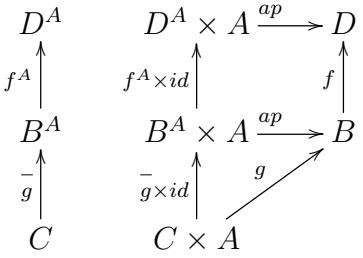
$$\frac{C \xleftarrow{f} B}{C^A \xleftarrow{f^A} B^A}$$

It can be shown that, once A and $C \xleftarrow{f} B$ are fixed, f^A is the function which accepts some input function $B \xleftarrow{g} A$ as argument and produces function $f \cdot g$ as result (see exercise 2.38). So f^A is the "compose with f" functional combinator:

$$(f^A)g \stackrel{\text{def}}{=} f \cdot g \tag{2.77}$$

Now we are ready to understand the laws which follow:

Exponentials absorption :



$$\overline{f \cdot g} = f^A \cdot \overline{g} \tag{2.78}$$

(Note how, from this, we also get $f^A = \overline{f \cdot ap}$.)

Exponentials-functor :

$$(g \cdot h)^A = g^A \cdot h^A \tag{2.79}$$

Exponentials-functor-id :

$$id^A = id \tag{2.80}$$

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

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

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

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

$$\begin{array}{ll} & \overbrace{(\underbrace{\mathsf{curry}\ f}_{\overline{f}}\ a)}^{g}\ b = f\ (a,b)\\ \equiv & \quad \{\ \text{introduce } g\ \}\\ & g\ b\ =\ f(a,b)\\ \equiv & \quad \{\ \text{since } g\ b = ap(g,b)\ (2.71)\ \}\\ & ap(g,b)\ =\ f(a,b)\\ \equiv & \quad \{\ g = \overline{f}\ a\ ;\ \text{natural-}id\ \}\\ & ap(\overline{f}\ a, id\ b)\ =\ f(a,b)\\ \equiv & \quad \{\ \text{product of functions: } (f\times g)(x,y) = (f\ x, g\ y)\ \}\\ & ap((\overline{f}\times id)(a,b))\ =\ f(a,b)\\ \equiv & \quad \{\ \text{composition}\ \}\\ & (ap\cdot(\overline{f}\times id))(a,b)\ =\ f(a,b)\\ \equiv & \quad \{\ \text{extensionality (2.5), i.e. removing points } a \text{ and } b\ \}\\ & ap\cdot(\overline{f}\times id)\ =\ f\end{array}$$

From the above we infer that the definition of curry is a re-statement of the cancellation law (2.74). That is,

$$\mathsf{curry}\ f\ \stackrel{\text{def}}{=}\ \overline{f} \tag{2.81}$$

and curry is transposition in HASKELL-speak.[3]

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

Next we do the same for the definition of uncurry :

$$
\begin{array}{ll}
& \underbrace{\mathsf{uncurry}\ f}_{k}\ (a,b) = f\ a\ b \\
\equiv & \quad \{\ \text{introduce } k \text{ ; lefthand side as calculated above}\ \} \\
& k\ (a,b) = (ap \cdot (f \times id))\ (a,b) \\
\equiv & \quad \{\ \text{extensionality (2.5)}\ \} \\
& k = ap \cdot (f \times id) \\
\equiv & \quad \{\ \text{universal property (2.73)}\ \} \\
& f = \overline{k} \\
\equiv & \quad \{\ \text{expand } k\ \} \\
& f = \overline{\mathsf{uncurry}\ f}
\end{array}
$$

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

$$
\begin{array}{ll}
& \overline{\widehat{f}} = ap \cdot (\overline{f} \times id) \\
\equiv & \quad \{\ \text{cancellation (2.74)}\ \} \\
& \overline{\widehat{f}} = f \\
\Box &
\end{array}
$$

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

$$g = \overline{f} \ \Leftrightarrow\ \hat{g} = f \tag{2.82}$$

leading to isomorphism

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

which can also be written as

$$(C^B)^A \underset{\text{curry}}{\overset{\text{uncurry}}{\cong}} C^{A \times B} \tag{2.83}$$

decorated with the corresponding witnesses.[4]

Isomorphism (2.83) is at the core of the theory and practice of functional programming. It clearly resembles a well known equality concerning numeric exponentials, $b^{c \times a} = (b^a)^c$. Moreover, other known facts about numeric exponentials, *e.g.* $a^{b+c} = a^b \times a^c$ or $(b \times c)^a = b^a \times c^a$ also find their counterpart in functional exponentials. The counterpart of the former,

$$A^{B+C} \quad \cong \quad A^B \times A^C \tag{2.84}$$

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

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

$$(B \times C)^A \cong B^A \times C^A \tag{2.85}$$

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

What about other facts valid for numeric exponentials such as $a^0 = 1$ and $1^a = 1$? We need to know what 0 and 1 mean as datatypes. Such elementary datatypes are presented in the section which follows.

Exercise 2.23. *Consider the witnesses of isomorphism (2.85)*

$$(B \times C)^A \underset{\text{pair}}{\overset{\text{unpair}}{\cong}} B^A \times C^A$$

defined by:

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

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

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

Exercise 2.24. *Show that the following equality*

$$\overline{f}\ a = f \cdot \langle \underline{a}, id \rangle \tag{2.86}$$

holds.
□

Exercise 2.25. *Consider function* $\alpha = [\overline{i_1}, \overline{i_2}]$.

- *Infer the principal (most general) type of* α *and depict it in a diagram.*
- $\widehat{\alpha}$ *is a well-known isomorphism — tell which by inferring its type.*

□

Exercise 2.26. *Prove the equality*

$$\underline{g} = \overline{g \cdot \pi_2}$$

knowing that

$$\overline{\pi_2} = \underline{id} \tag{2.87}$$

holds.
□

2.15 Elementary datatypes

So far we have talked mostly about arbitrary datatypes represented by capital letters A, B, *etc.* (lowercase `a`, `b`, *etc.* in the HASKELL illustrations). We also mentioned $\mathbb{R}$, Bool and $\mathbb{N}$ and, in particular, the fact that we can associate to each

natural number n its *initial segment* $\mathsf{n} = \{1, 2, \ldots, n\}$. We extend this to $\mathbb{N}_0$ by stating $\mathsf{0} = \{\}$ and, for $n > 0$, $\mathsf{n+1} = \{n+1\} \cup \mathsf{n}$.

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

Datatype 0

Among such enumerated types, $\mathsf{0}$ is the smallest because it is empty. This is the `Void` datatype in HASKELL, which has no constructor at all. Datatype $\mathsf{0}$ (which we tend to write simply as 0) may not seem very "useful" in practice but it is of theoretical interest. For instance, it is easy to check that the following "obvious" properties hold:

$$A + 0 \cong A \tag{2.88}$$

$$A \times 0 \cong 0 \tag{2.89}$$

Datatype 1

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

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

$$A \times 1 \cong A \tag{2.90}$$

$$A^0 \cong 1 \tag{2.91}$$

We can read (2.90) informally as follows: if the second component of a record ("struct") cannot change, then it is useless and can be ignored. Selector π_1 is, in this context, an iso mapping the left-hand side of (2.90) to its right-hand side. Its inverse is $\langle id, \underline{c} \rangle$ where c is a particular choice of inhabitant for datatype 1. Concerning (2.91), A^0 denotes the set of all functions from the empty set to some A. What does (2.91) mean? It simply tells us that there is only one function in

such a set — the empty function mapping "no" value at all. This fact confirms our choice of notation once again (compare with $a^0 = 1$ in a numeric context).

Next, we may wonder about facts

$$1^A \cong 1 \tag{2.92}$$

$$A^1 \cong A \tag{2.93}$$

which are the functional exponentiation counterparts of $1^a = 1$ and $a^1 = a$. Fact (2.92) is valid: it means that there is only one function mapping A to some singleton set $\{c\}$ — the constant function $\underline{c}$. There is no room for another function in 1^A because only c is available as output value. Fact (2.93) is also valid: all functions in A^1 are (single valued) constant functions and there are as many constant functions in such a set as there are elements in A.

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

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

Datatype 2

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

$$1 + 1 \cong 2$$

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

Exercise 2.27. *Relate* HASKELL *expressions*

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

and

```
\f->(f True, f False)
```

to the following isomorphisms involving generic elementary type 2*:*

$$2 \times A \cong A + A \tag{2.94}$$
$$A \times A \cong A^2 \tag{2.95}$$

Apply the exchange law *(2.47) to the first expression above.*
□

2.16 Finitary products and coproducts

In section 2.8 it was suggested that product could be regarded as the abstraction behind data-structuring primitives such as `struct` in C or `record` in PASCAL. Similarly, coproducts were suggested in section 2.9 as abstract counterparts of C unions or PASCAL variant records. For a finite A, exponential B^A could be realized as an *array* in any of these languages. These analogies are captured in table 2.1.

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

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

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

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

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

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

Datatype n

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

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

and

$$\underbrace{A \times \ldots \times A}_{n} \ \cong \ A^n \tag{2.96}$$

$$\underbrace{A + \ldots + A}_{n} \ \cong \ n \times A \tag{2.97}$$

hold.

Exercise 2.28. *On the basis of table 2.1, encode* $undistr$ *(2.49) in C or* PASCAL. *Compare your code with the* HASKELL *pointfree and pointwise equivalents.*
□

2.17 Initial and terminal datatypes

All properties studied for binary *splits* and binary *eithers* extend to the finitary case. For the particular situation $n = 1$, we will have $\langle f \rangle = [\ f\ ] = f$ and $\pi_1 = i_1 = id$, of course. For the particular situation $n = 0$, finitary products

Abstract notation	PASCAL	C/C++	**Description**

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

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

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

Table 2.1: Abstract notation versus programming language data-structures.

"degenerate" to 1 and finitary coproducts "degenerate" to 0. So diagrams (2.21) and (2.36) are reduced to

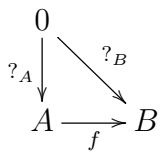
The standard notation for the empty *split* $\langle\rangle$ is $!_C$, where subscript C can be omitted if implicit in the context. By the way, this is precisely the only function in 1^C, recall (2.92). Dually, the standard notation for the empty *either* $[\ ]$ is $?_C$, where subscript C can also be omitted. By the way, this is precisely the only function in C^0, recall (2.91).

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

Initial datatype reflexion :

$$?_0 = id_0 \tag{2.98}$$

Initial datatype fusion :

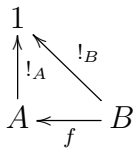


$$f \cdot ?_A = ?_B \tag{2.99}$$

Terminal datatype reflexion :

$$!_1 = id_1 \tag{2.100}$$

Terminal datatype fusion :



$$!_A \cdot f = !_B \tag{2.101}$$

Exercise 2.29. *Particularize the* exchange law *(2.47) to empty products and empty coproducts,* i.e. 1 *and* 0.
□

2.18 Sums and products in HASKELL

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

```
data CostumerId = P Int | CC Int
```

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

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

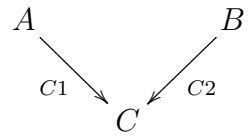
```
Main> :i CostumerId
-- type constructor
data CostumerId

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

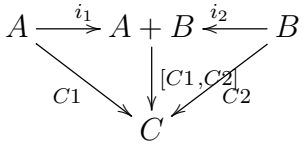
In general, let A and B be two known datatypes. Via declaration

```
data C = C1 A | C2 B
```
(2.102)

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

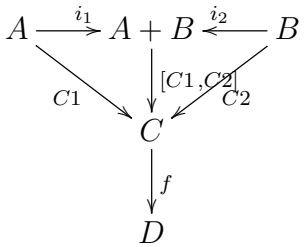


This diagram leads to an obvious instance of coproduct diagram (2.36),



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

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



This is an opportunity for $+$-*fusion* (2.40), whereby we obtain

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

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

$$\begin{aligned} & f : C \to D \\ & f(C1\, a) = \ldots \\ & f(C2\, b) = \ldots \end{aligned}$$

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

$$\begin{aligned} & inv \cdot [C1, C2] = id \\ \equiv & \quad \{ \text{ by } +\text{-}\textit{fusion} \text{ (2.40) } \} \\ & [inv \cdot C1, inv \cdot C2] = id \\ \equiv & \quad \{ \text{ by } +\text{-}\textit{reflexion} \text{ (2.39) } \} \\ & [inv \cdot C1, inv \cdot C2] = [i_1, i_2] \\ \equiv & \quad \{ \textit{either} \text{ structural equality (2.58) } \} \\ & inv \cdot C1 = i_1 \wedge inv \cdot C2 = i_2 \end{aligned}$$

Therefore:

$$\begin{aligned} & inv : C \to A + B \\ & inv(C1\, a) = i_1\, a \\ & inv(C2\, b) = i_2\, b \end{aligned}$$

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

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

$[C1, C2]$ is called the *algebra* of datatype C and its inverse inv is called the *coalgebra* of C. The algebra contains the constructors of $C1$ and $C2$ of type C, that

is, it is used to "build" C-values. In the opposite direction, co-algebra inv enables us to "destroy" or observe values of C:

$$C \underset{[C1,C2]}{\overset{inv}{\cong}} A + B$$

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

$$\begin{array}{ccccc} A & \xleftarrow{\pi_1} & A \times A & \xrightarrow{\pi_2} & A \\ & {\scriptstyle x}\nwarrow & \downarrow{\scriptstyle Point} & \nearrow{\scriptstyle y} & \\ & & Point & & \end{array} \tag{2.103}$$

This rises an algebra ($Point$) and a coalgebra ($\langle x, y\rangle$) for datatype $Point$:

$$Point \underset{Point}{\overset{\langle x,y\rangle}{\cong}} A \times A$$

In HASKELL one writes

```
data Point a = Point { x :: a, y :: a }
```

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

```
Point :: a -> a -> Point a
```

Finally, what is the "pointer"-equivalent in HASKELL? This corresponds to $A = 1$ in (2.102) and to the following HASKELL declaration:

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

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

```
data C = C1 | C2 B
```

The difference is that here `C1` denotes an inhabitant of C (and so a clause $f(C1\ a) = \ldots$ is rewritten to $f\ C1 = \ldots$) while above `C1` denotes a (constant) function $C \xleftarrow{C1} 1$. Isomorphism (2.93) helps in comparing these two alternative situations.

2.19 Exercises

Exercise 2.30. *Let A and B be two disjoint datatypes, that is, $A \cap B = \emptyset$ holds. Show that isomorphism*

$$A \cup B \quad \cong \quad A + B \tag{2.104}$$

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

Exercise 2.31. *Knowing that a given function* xr *satisfies property*

$$\mathsf{xr} \cdot \langle\langle f, g\rangle, h\rangle = \langle\langle f, h\rangle, g\rangle \tag{2.105}$$

for all f, g and h, derive from (2.105) the definition of xr*:*

$$\mathsf{xr} = \langle \pi_1 \times id, \pi_2 \cdot \pi_1 \rangle \tag{2.106}$$

□

Exercise 2.32. *Let $distr$ (read:* 'distribute right'*) be the bijection which witnesses isomorphism $A \times (B + C) \cong A \times B + A \times C$. Fill in the "..." in the diagram which follows so that it describes bijection $distl$ (red:* 'distribute left'*) which witnesses isomorphism $(B + C) \times A \cong B \times A + C \times A$:*

$$(B + C) \times A \xrightarrow{swap} \cdots \xrightarrow{distr} \cdots \xrightarrow{\cdots} B \times A + C \times A$$
$$\xrightarrow[distl]{}$$

□

Exercise 2.33. *In the context of exercise 2.32, prove*

$$[g, h] \times f \quad = \quad [g \times f, h \times f] \cdot distl \tag{2.107}$$

knowing that

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

holds.

□

Exercise 2.34. *The arithmetic law* $(a + b)(c + d) = (ac + ad) + (bc + bd)$ *corresponds to the isomorphism*

$$(A + B) \times (C + D) \qquad \cong \qquad (A \times C + A \times D) + (B \times C + B \times D)$$

$$h = [[i_1 \times i_1, i_1 \times i_2], [i_2 \times i_1, i_2 \times i_2]]$$

From universal property (2.57) infer the following definition of function h*, written in Haskell syntax:*

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

□

Exercise 2.35. *Every C programmer knows that a* `struct` *of pointers*

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

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

$$\begin{array}{rcl}
(A+1) \times (B+1) & \xleftarrow{\;i_1\;} & ((A+1) \times B) + ((A+1) \times 1) \\
 & & \uparrow i_2 \\
 & & (A \times B + 1 \times B) + (A \times 1 + 1 \times 1) \\
 & & \uparrow i_3 \\
 & & (A \times B + B) + (A + 1) \\
 & & \uparrow i_4 \\
(A \times B + (B + A)) + 1 & \xrightarrow[i_5]{} & A \times B + (B + (A + 1))
\end{array}$$

which witness the observation above.
□

Exercise 2.36. *Prove the following property of McCarthy conditionals:*

$$p \to f \cdot g, h \cdot k \quad = \quad [f, h] \cdot (p \to i_1 \cdot g, i_2 \cdot k) \tag{2.108}$$

□

Exercise 2.37. *Assuming the fact*

$$(p? + p?) \cdot p? \quad = \quad (i_1 + i_2) \cdot p? \tag{2.109}$$

show that nested conditionals can be simplified:

$$p \to (p \to f, g), (p \to h, k) \quad = \quad p \to f, k \tag{2.110}$$

□

Exercise 2.38. *Show that* $(\overline{f \cdot ap})\ g = f \cdot g$ *holds,* cf. *(2.77).*
□

Exercise 2.39. *Consider the higher-order isomorphism* $flip$ *defined as follows:*

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

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

Exercise 2.40. *Let* $C \xrightarrow{\mathsf{const}} C^A$ *be the function of exercise 2.1, that is,* $\mathsf{const}\, c = \underline{c}_A$. *Which fact is expressed by the following diagram featuring* const*?*

$$\begin{array}{ccc} C & \xrightarrow{\mathsf{const}} & C^A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f^A} \\ B & \xrightarrow[\mathsf{const}]{} & B^A \end{array}$$

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

Exercise 2.41. *Show that*

$$\overline{\pi_2} \cdot f = \overline{\pi_2}$$

holds for every f. *Thus* $\overline{\pi_2}$ *is a constant function — which one?*
□

Exercise 2.42. *Establish the difference between the following two declarations in* HASKELL,

```
data D = D1 A | D2 B C
data E = E1 A | E2 (B,C)
```

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

2.20 Bibliography notes

A few decades ago John Backus read, in his Turing Award Lecture, a revolutionary paper [3]. This paper proclaimed conventional command-oriented programming languages obsolete because of their inefficiency arising from retaining, at a high-level, the so-called "memory access bottleneck" of the underlying computation

model — the well-known *von Neumann* architecture. Alternatively, the (at the time already mature) *functional programming* style was put forward for two main reasons. Firstly, because of its potential for concurrent and parallel computation. Secondly — and Backus emphasis was really put on this —, because of its strong algebraic basis.

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

The impact of Backus first argument in the computing science and computer architecture communities was considerable, in particular if assessed in quality rather than quantity and in addition to the almost contemporary *structured programming* trend [5]. By contrast, his second argument for changing computer programming was by and large ignored, and only the so-called *algebra of programming* research minorities pursued in this direction. However, the advances in this area throughout the last two decades are impressive and can be fully appreciated by reading a textbook written relatively recently by Bird and de Moor [6]. A comprehensive review of the voluminous literature available in this area can also be found in this book.

Although the need for a pointfree algebra of programming was first identified by Backus, perhaps influenced by Iverson's APL growing popularity in the USA at that time, the idea of reasoning and using mathematics to transform programs is much older and can be traced to the times of McCarthy's work on the foundations of computer programming [27], of Floyd's work on program meaning [9] and of Paterson and Hewitt's *comparative schematology* [35]. Work of the so-called *program transformation* school was already very expressive in the mid 1970s, see for instance references [7].

The mathematics adequate for the effective integration of these related but independent lines of thought was provided by the categorial approach of Manes and Arbib compiled in a textbook [26] which has very strongly influenced the last decade of 20th century theoretical computer science.

A so-called MPC ("Mathematics of Program Construction") community has been among the most active in producing an integrated body of knowledge on the algebra of programming which has found in functional programming an eloquent and paradigmatic medium. Functional programming has a tradition of absorb-

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

ing fresh results from theoretical computer science, algebra and category theory. Languages such as HASKELL [5] have been competing to integrate the most recent developments and therefore are excellent *prototyping* vehicles in courses on program calculation, as happens with this book.

For fairly recent work on this topic see e.g. [12, 15, 16, 11].

Chapter 3

Recursion in the Pointfree Style

How useful from a programmer's point of view are the abstract concepts presented in the previous chapter? Recall that a table was presented — table 2.1 — which records an analogy between abstract type notation and the corresponding data-structures available in common, imperative languages.

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

3.1 Motivation

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

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

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

of multiplication:

$$\begin{cases} a \times 0 = 0 \\ a \times 1 = a \\ a \times (b + c) = a \times b + a \times c \end{cases} \tag{3.1}$$

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

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

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

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

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

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

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

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

Indeed, the two arguments a and b will have different types in general. Let us see why and how. Starting by looking at infix operators $(\times)$ and $(+)$ as *curried* operators — recall section 2.14 — we can resort to the corresponding *sections* and write:

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

It can be easily checked that

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

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

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

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

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

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

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

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

$$\begin{cases} (a\times) \cdot \underline{0} = \underline{0} \\ (a\times) \cdot (+1) = (+a) \cdot (a\times) \end{cases}$$

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

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

— thanks to the Eq-$+$ rule (2.58) — then single out the common factor $(a\times)$ in the left hand side,

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

— thanks to $+$-fusion (2.40) — and finally do a similar *fission* operation on the other side,

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

— thanks to $+$-absorption (2.41).

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

$$\begin{array}{ccc} \mathbb{N}_0 & \xleftarrow{[\underline{0}\ ,(+1)]} & A+\mathbb{N}_0 \\ {\scriptstyle (a\times)}\downarrow & & \downarrow{\scriptstyle id+(a\times)} \\ \mathbb{N}_0 & \xleftarrow[[\underline{0}\ ,(+a)]]{} & A+\mathbb{N}_0 \end{array}$$

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

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

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

$$\begin{array}{ccc} \mathbb{N}_0 & \overset{out=in^\circ}{\underset{in=[\underline{0}\ ,succ]}{\cong}} & 1+\mathbb{N}_0 \\ {\scriptstyle (a\times)}\downarrow & & \downarrow{\scriptstyle id+(a\times)} \\ \mathbb{N}_0 & \xleftarrow[[\underline{0}\ ,(+a)]]{} & 1+\mathbb{N}_0 \end{array}$$

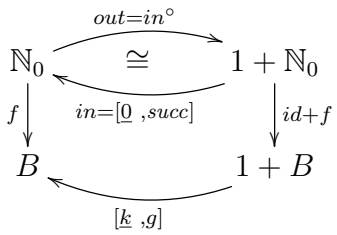
capturing both the isomorphism and the $(a\times)$ recursive function. By solving the isomorphism equation $out \cdot in = id$ we easily obtain the definition of out, the converse of in [1]:

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

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

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

corresponding generalization of $(a\times)$ is denoted by f below:



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

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

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

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

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

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

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

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

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

More knowledge about for-loops can be extracted from (3.7). Later on we will show that for-loops are special cases of a more general concept termed *catamorphism* (see eg. section 3.6). So it is perhaps wiser to study the (more general) theory of catamorphisms first and then instantiate it for for-loops. Then we will

understand how more interesting for-loops can be synthesized, for instance those handling more than one "global variable", thanks to catamorphism theory (for instance, the mutual recursion laws).

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

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

$$(a+) \quad = \quad \mathsf{for}\ succ\ a \tag{3.9}$$

□

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

$$h \cdot (\mathsf{for}\ g\ k) = \mathsf{for}\ j\ (h\ k) \quad \Leftarrow \quad h \cdot g = j \cdot h \tag{3.10}$$

can be derived from universal-property (3.7) [2]*. Since $(a+) \cdot id = (a+)$, provide an alternative derivation of (3.9) using the fusion-law above.*

□

Exercise 3.3. *From (3.4) and fusion-law (3.10) infer: $(a*) \cdot \mathsf{succ} = \mathsf{for}\ a\ (a+)$.*

□

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

□

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

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

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

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

3.2 From natural numbers to finite sequences

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

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

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

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

and so on.

What interests us in such "first year programming course" datatype declarations? Records and pointers have already been dealt with in table 2.1. So we can use this table to find the abstract version of datatype L, by replacing pointers by

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

$$\begin{cases} L &= 1 + N \\ N &= A \times (1 + N) \end{cases} \tag{3.11}$$

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

$$\begin{cases} L &= 1 + N \\ N &= A \times (1 + N) \end{cases}$$

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

$$\begin{cases} L &= 1 + N \\ N &= A \times L \end{cases}$$

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

$$\begin{cases} L &= 1 + A \times L \\ N &= A \times L \end{cases}$$

System (3.11) is thus equivalent to:

$$\begin{cases} L &= 1 + A \times L \\ N &= A \times (1 + N) \end{cases} \tag{3.12}$$

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

$$x = 1 + \frac{x}{2} \qquad ? \tag{3.13}$$

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

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

$$\equiv \quad \{ \text{ adding } -\tfrac{x}{2} \text{ to both sides of the equation } \}$$

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

$$
\begin{aligned}
&\equiv && \{\ -\tfrac{x}{2} \text{ cancels } \tfrac{x}{2}\ \}\\
&& x - \frac{x}{2} = 1\\
&\equiv && \{\ \text{multiplying both sides of the equation by } 2 \textit{ etc.}\ \}\\
&& 2 \times x - x = 2\\
&\equiv && \{\ \text{subtraction}\ \}\\
&& x = 2
\end{aligned}
$$

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

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

$$
\begin{array}{ll}
& 2 \times x - 2 \times \dfrac{x}{2} = 2 \\
\equiv & \quad \{ \text{ 2 cancels its inverse } \tfrac{1}{2} \ \} \\
& 2 \times x - 1 \times x = 2 \\
\equiv & \quad \{ \text{ multiplication distributes over addition } \} \\
& (2-1) \times x = 2 \\
\equiv & \quad \{ \ 2 - 1 = 1 \text{ and } 1 \text{ is the unit of multiplication } \} \\
& x = 2
\end{array}
$$

Back to (3.12), we would like to submit each of the equations, *e.g.*

$$L \ = \ 1 + A \times L \tag{3.14}$$

to a similar reasoning. Can we do it? The analogy which can be found between this equation and (3.13) goes beyond pattern similarity. From chapter 2 we know that many properties required in the reasoning above hold in the context of (3.14), provided the "=" sign is replaced by the "$\cong$" sign, that of set-theoretical isomorphism. Recall that, for instance, $+$ is associative (2.46), 0 is the unit of addition (2.88), 1 is the unit of multiplication (2.90), multiplication distributes over addition (2.50) *etc.* Moreover, the first step above assumed that addition is compatible (monotonic) with respect to equality,

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

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

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

— recall (2.44) for isos f and g.

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

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

which hold about numbers but do not hold about datatypes. In fact, neither products nor coproducts have arbitrary inverses [3], and so we cannot "calculate by cancellation". How do we circumvent this limitation?

Just think of how we would have gone about (3.13) in case we didn't know about the *cancellation properties*: we would be bound to the x by $1+\frac{x}{2}$ substitution plus the other properties. By performing such a substitution over and over again we would obtain...

$$
\begin{array}{ll}
& x = 1 + \dfrac{x}{2} \\
\equiv & \quad \{\ x \text{ by } 1 + \frac{x}{2} \text{ substitution followed by simplification } \} \\
& x = 1 + \dfrac{1 + \frac{x}{2}}{2} = 1 + \dfrac{1}{2} + \dfrac{x}{4} \\
\equiv & \quad \{\ \text{the same as above } \} \\
& x = 1 + \dfrac{1}{2} + \dfrac{1 + \frac{x}{2}}{4} = 1 + \dfrac{1}{2} + \dfrac{1}{4} + \dfrac{x}{8} \\
\equiv & \quad \{\ \text{over and over again, } n\text{-times } \} \\
& \cdots \\
\equiv & \quad \{\ \text{simplification } \} \\
& x = \displaystyle\sum_{i=0}^{n} \frac{1}{2^i} + \frac{x}{2^{n+1}} \\
\equiv & \quad \{\ \text{sum of } n \text{ first terms of a geometric progression } \} \\
& x = (2 - \dfrac{1}{2^n}) + \dfrac{x}{2^{n+1}} \\
\equiv & \quad \{\ \text{let } n \to \infty \ \} \\
& x = (2 - 0) + 0 \\
\equiv & \quad \{\ \text{simplification } \} \\
& x = 2
\end{array}
$$

Clearly, this is a much more complicated way of finding solution $x = 2$ for

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

equation (3.13). But we would have loved it in case it were the only known way, and this is precisely what happens with respect to (3.14). In this case we have:

$$
\begin{aligned}
& L = 1 + A \times L \\
\equiv & \quad \{ \text{ substitution of } 1 + A \times L \text{ for } L \} \\
& L = 1 + A \times (1 + A \times L) \\
\equiv & \quad \{ \text{ distributive property (2.50) } \} \\
& L \cong 1 + A \times 1 + A \times (A \times L) \\
\equiv & \quad \{ \text{ unit of product (2.90) and associativity of product (2.32) } \} \\
& L \cong 1 + A + (A \times A) \times L \\
\equiv & \quad \{ \text{ by (2.91), (2.93) and (2.96) } \} \\
& L \cong A^0 + A^1 + A^2 \times L \\
\equiv & \quad \{ \text{ another substitution as above and similar simplifications } \} \\
& L \cong A^0 + A^1 + A^2 + A^3 \times L \\
\equiv & \quad \{ \text{ after } (n+1)\text{-many similar steps } \} \\
& L \cong \sum_{i=0}^{n} A^i + A^{n+1} \times L
\end{aligned}
$$

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

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

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

denoted by $A^\star$ and defined as follows:

$$A^\star \stackrel{\text{def}}{=} \bigcup_{i \geqslant 0} A^i \tag{3.15}$$

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

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

Because $\sum_{i=0}^{\infty} A^i$ is isomorphic to $\bigcup_{i=0}^{\infty} A^i$ (see exercise 2.30), we finally have:

$$L \cong A^\star$$

All in all, we have obtained $A^\star$ as a solution to equation (3.14). In other words, datatype L is isomorphic to the datatype which contains all finite sequences of some predefined datatype A. This corresponds to the HASKELL `[a]` datatype, in general. Recall that we started from the "linked-list datatype" expressed in PASCAL or C. In fact, wherever the C programmer thinks of linked-lists, the HASKELL programmer will think of finite sequences.

But, what does equation (3.14) mean in fact? Is $A^\star$ the only solution to this equation? Back to the numeric field, we know of equations which have more than one solution — for instance $x = \frac{x^2+3}{4}$, which admits two solutions 1 and 3 —, which have no solution at all — for instance $x = x + 1$ —, or which admit an infinite number of — for instance $x = x$.

We will address these topics in the next section about *inductive* datatypes and in chapter 9, where the formal semantics of recursion will be made explicit. This is where the "limit" constructions used informally in this section will be shown to make sense.

3.3 Introducing inductive datatypes

Datatype L as defined by (3.14) is said to be *recursive* because L "recurs" in the definition of L itself [4]. From the discussion above, it is clear that set-theoretical

[4] By analogy, we may regard (3.13) as a "recursive definition" of number 2.

equality "=" in this equation should give place to set-theoretical isomorphism ("$\cong$"):

$$L \;\cong\; 1 + A \times L \tag{3.16}$$

Which isomorphism $L \xleftarrow{in} 1 + A \times L$ do we expect to witness (3.14)? This will depend on which particular solution to (3.14) we are thinking of. So far we have seen only one, $A^\star$. By recalling the notion of *algebra* of a datatype (section 2.18), so we may rephrase the question as: which algebra

$$A^\star \xleftarrow{in} 1 + A \times A^\star$$

do we expect to witness the tautology which arises from (3.14) by replacing unknown L with solution $A^\star$, that is

$$A^\star \;\cong\; 1 + A \times A^\star \qquad\qquad ?$$

It will have to be of the form $in = [in_1 , in_2]$ as depicted by the following diagram:

$$\begin{array}{ccccc} 1 & \xrightarrow{i_1} & 1 + A \times A^\star & \xleftarrow{i_2} & A \times A^\star \\ & {\scriptstyle in_1}\searrow & \downarrow{\scriptstyle in} & \swarrow{\scriptstyle in_2} & \\ & & A^\star & & \end{array} \tag{3.17}$$

Arrows in_1 and in_2 can be guessed rather intuitively: $in_1 = \underline{[\,]}$, which will express the "NIL pointer" by the empty sequence, at $A^\star$ level, and $in_2 = cons$, where $cons$ is the standard "left append" sequence constructor, which we for the moment introduce rather informally as follows:

$$\begin{array}{l} cons : A \times A^\star \to A^\star \\ cons(a, [a_1, \ldots, a_n]) = [a, a_1, \ldots, a_n] \end{array} \tag{3.18}$$

In a diagram:

$$\begin{array}{ccccc} 1 & \xrightarrow{i_1} & 1 + A \times A^\star & \xleftarrow{i_2} & A \times A^\star \\ & {\scriptstyle \underline{[\,]}}\searrow & \downarrow{\scriptstyle [\underline{[\,]}, cons]} & \swarrow{\scriptstyle cons} & \\ & & A^\star & & \end{array} \tag{3.19}$$

Of course, for in to be iso it needs to have an inverse, which is not hard to guess,

$$out \;\stackrel{\text{def}}{=}\; (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?) \tag{3.20}$$

where sequence operators hd (*head of a nonempty sequence*) and tl (*tail of a nonempty sequence*) are (again informally) described as follows:

$$\begin{array}{l} hd : A^\star \to A \\ hd\,[a_1, a_2, \ldots, a_n] = a_1 \end{array} \tag{3.21}$$

$$\begin{array}{l} tl : A^\star \to A^\star \\ tl\,[a_1, a_2, \ldots, a_n] = [a_2, \ldots, a_n] \end{array} \tag{3.22}$$

Showing that in and out are each other inverses is not a hard task either:

$$\begin{array}{ll} & in \cdot out = id \\ \equiv & \quad \{ \text{ definitions of } in \text{ and } out \} \\ & [[\underline{\ }], cons] \cdot (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?) = id \\ \equiv & \quad \{ +\text{-absorption (2.41) and (2.15)} \} \\ & [[\underline{\ }], cons \cdot \langle hd, tl \rangle] \cdot (=_{[\,]}?) = id \\ \equiv & \quad \{ \text{ property of sequences: } cons(hd\,s, tl\,s) = s \} \\ & [[\underline{\ }], id] \cdot (=_{[\,]}?) = id \\ \equiv & \quad \{ \text{ going pointwise (2.62) } \} \\ & \left\{ \begin{array}{lcl} =_{[\,]}\ a & \Rightarrow & [[\underline{\ }], id]\,(i_1\,a) \\ \neg(=_{[\,]}\ a) & \Rightarrow & [[\underline{\ }], id]\,(i_2\,a) \end{array} \right. = a \\ \equiv & \quad \{ \text{ } +\text{-cancellation (2.38) } \} \\ & \left\{ \begin{array}{lcl} =_{[\,]}\ a & \Rightarrow & \underline{[\,]}\,a \\ \neg(=_{[\,]}\ a) & \Rightarrow & \underline{id}\,a \end{array} \right. = a \\ \equiv & \quad \{ \text{ } a = [\,] \text{ in one case and identity function (2.9) in the other } \} \\ & \left\{ \begin{array}{lcl} a = [\,] & \Rightarrow & a \\ \neg(a = [\,]) & \Rightarrow & a \end{array} \right. = a \\ \equiv & \quad \{ \text{ property } (p \to f, f) = f \text{ holds } \} \\ & a = a \end{array}$$

A comment on the particular choice of terminology above: symbol in suggests that we are going inside, or constructing (synthesizing) values of $A^\star$; symbol out

suggests that we are going out, or destructing (analyzing) values of $A^\star$. We shall often resort to this duality in the sequel.

Are there more solutions to equation (3.16)? In trying to implement this equation, a HASKELL programmer could have written, after the declaration of type A, the following datatype declaration:

```
data L = Nil () | Cons (A,L)
```

which, as we have seen in section 2.18, can be written simply as

```
data L = Nil | Cons (A,L)
```
(3.23)

and generates diagram

$$\begin{array}{ccccc} 1 & \xrightarrow{i_1} & 1 + A \times L & \xleftarrow{i_2} & A \times L \\ & \underline{Nil}\searrow & \downarrow{\scriptstyle in'} & \swarrow Cons & \\ & & L & & \end{array} \tag{3.24}$$

leading to algebra $in' = [\underline{Nil}, Cons]$.

HASKELL seems to have generated another solution for the equation, which it calls L. To avoid the inevitable confusion between this symbol denoting the newly created datatype and symbol L in equation (3.16), which denotes a mathematical variable, let us use symbol T to denote the former (T stands for "type"). This can be coped with very simply by writing `T` instead of `L` in (3.23):

```
data T = Nil | Cons (A,T)
```
(3.25)

In order to make T more explicit, we will write in_{T} instead of in'.

Some questions are on demand at this point. First of all, what is datatype T? What are its inhabitants? Next, is $\mathsf{T} \xleftarrow{in_{\mathsf{T}}} 1 + A \times \mathsf{T}$ an iso or not?

HASKELL will help us to answer these questions. Suppose that A is a primitive numeric datatype, and that we add `deriving Show` to (3.25) so that we can "see" the inhabitants of the T datatype. The information associated to T is thus:

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

-- constructors:
```

```
Nil :: T
Cons :: (A,T) -> T

-- instances:
instance Show T
instance Eval T
```

By typing `Nil`

```
Main> Nil
Nil :: T
```

we confirm that Nil is itself an inhabitant of T, and by typing `Cons`

```
Main> Cons
<<function>> :: (A,T) -> T
```

we realize that $Cons$ is not so (as expected), but it can be used to build such inhabitants, for instance:

```
Main> Cons(1,Nil)
        Cons (1,Nil) :: T
```

or

```
Main> Cons(2,Cons(1,Nil))
        Cons (2,Cons (1,Nil)) :: T
```

etc. We conclude that *expressions* involving Nil and $Cons$ are inhabitants of type T. Are these the *only* ones? The answer is *yes* because, by design of the HASKELL language, the constructors of type T will remain fixed once its declaration is interpreted, that is, no further constructor can be added to T. Does in_{T} have an inverse? Yes, its inverse is coalgebra

$$\begin{aligned} &out_{\mathsf{T}} : \mathsf{T} \to 1 + A \times \mathsf{T} \\ &out_{\mathsf{T}}\, Nil = i_1\ \text{NIL} \\ &out_{\mathsf{T}}(Cons(a, l)) = i_2(a, l) \end{aligned} \tag{3.26}$$

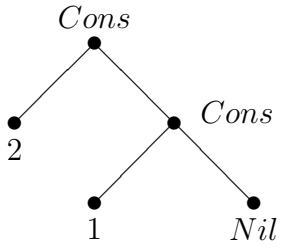
which can be straightforwardly encoded in HASKELL using the `Either` realization of $+$ (recall sections 2.9 and 2.18):

```
        outT :: T -> Either () (A,T)
        outT Nil = Left ()
outT (Cons(a,l)) = Right(a,l)
```

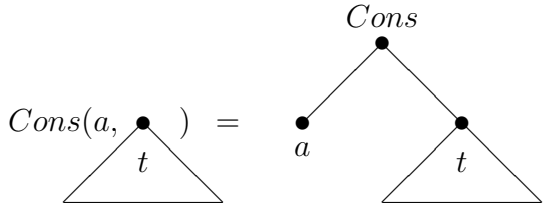
In summary, isomorphism

$$\mathsf{T} \underset{in_{\mathsf{T}}}{\overset{out_{\mathsf{T}}}{\cong}} 1 + A \times \mathsf{T} \tag{3.27}$$

holds, where datatype T is inhabited by symbolic expressions which we may visualize very conveniently as trees, for instance



picturing expression $Cons(2, Cons(1, Nil))$. Nil is the empty tree and $Cons$ may be regarded as the operation which adds a new root and a new branch, say a, to a tree t:



The choice of symbols `T`, `Nil` and `Cons` was rather arbitrary in (3.25). Therefore, an alternative declaration such as, for instance,

```
data U = Stop | Join (A,U)
```  (3.28)

would have been perfectly acceptable, generating another solution for the equation under algebra $[\underline{Stop}, Join]$. It is easy to check that (3.28) is but a renaming of

Nil to $Stop$ and of $Cons$ to $Join$. Therefore, both datatypes are isomorphic, or "abstractly the same".

Indeed, any other datatype X *inductively* defined by a constant and a binary constructor accepting A and X as parameters will be a solution to the equation. Because we are just renaming symbols in a consistent way, all such solutions are abstractly the same. All of them capture the abstract notion of a *list* of symbols.

We wrote "inductively" above because the set of all expressions (trees) which inhabit the type is defined by induction. Such types are called *inductive* and we shall have a lot more to say about them in chapter 9 .

Exercise 3.6. *Obviously,*

```
either (const []) (:)
```

does not work as a HASKELL *realization of the mediating arrow in diagram (3.19). What do you need to write instead?*
□

3.4 Observing an inductive datatype

Suppose that one is asked to express a particular *observation* of an inductive such as T (3.25), that is, a function of signature $B \xleftarrow{f} \mathsf{T}$ for some target type B. Suppose, for instance, that A is $\mathbb{N}_0$ (the set of all non-negative integers) and that we want to add all elements which occur in a T-list. Of course, we have to ensure that addition is available in $\mathbb{N}_0$,

$$\begin{aligned} &add : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \\ &add(x, y) \stackrel{\text{def}}{=} x + y \end{aligned}$$

and that $0 \in \mathbb{N}_0$ is a value denoting "the addition of nothing". So constant arrow $\mathbb{N}_0 \xleftarrow{\underline{0}} 1$ is available. Of course, $add(0, x) = add(x, 0) = x$ holds, for all $x \in \mathbb{N}_0$. This property means that $\mathbb{N}_0$, together with operator add and constant 0, forms a *monoid*, a very important algebraic structure in computing which will be exploited intensively later in this book. The following arrow "packaging" $\mathbb{N}_0$, add and $\underline{0}$,

$$\mathbb{N}_0 \xleftarrow{[\underline{0}, add]} 1 + \mathbb{N}_0 \times \mathbb{N}_0 \tag{3.29}$$

is a convenient way to express such a structure. Combining this arrow with the algebra

$$\mathsf{T} \xleftarrow{in_{\mathsf{T}}} 1 + \mathbb{N}_0 \times \mathsf{T} \tag{3.30}$$

which defines T, and the function f we want to define, the target of which is $B = \mathbb{N}_0$, we get the almost closed diagram which follows, in which only the dashed arrow is yet to be filled in:

$$\begin{array}{ccc} \mathsf{T} & \xleftarrow{in_{\mathsf{T}}} & 1 + \mathbb{N}_0 \times \mathsf{T} \\ {\scriptstyle f}\downarrow & & \vdots \\ \mathbb{N}_0 & \xleftarrow[[\underline{0},add]]{} & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \end{array} \tag{3.31}$$

We know that $in_{\mathsf{T}} = [\underline{Nil}, Cons]$. A pattern for the missing arrow is not difficult to guess: in the same way f bridges T and $\mathbb{N}_0$ on the lefthand side, it will do the same job on the righthand side. So pattern $\cdots + \cdots \times f$ comes to mind (recall section 2.10), where the "$\cdots$" are very naturally filled in by identity functions. All in all, we obtain diagram

$$\begin{array}{ccc} \mathsf{T} & \xleftarrow{[\underline{Nil},Cons]} & 1 + \mathbb{N}_0 \times \mathsf{T} \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle id+id\times f} \\ \mathbb{N}_0 & \xleftarrow[[\underline{0},add]]{} & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \end{array} \tag{3.32}$$

which pictures the following property of f

$$f \cdot [\underline{Nil}, Cons] \quad = \quad [\underline{0}, add] \cdot (id + id \times f) \tag{3.33}$$

and is easy to convert to pointwise notation:

$$\begin{array}{ll} & f \cdot [\underline{Nil}, Cons] = [\underline{0}, add] \cdot (id + id \times f) \\ \equiv & \quad \{ \text{ (2.40) on the lefthand side, (2.41) and identity } id \text{ on the righthand side } \} \\ & [f \cdot \underline{Nil}, f \cdot Cons] = [\underline{0}, add \cdot (id \times f)] \\ \equiv & \quad \{ \textit{either} \text{ structural equality (2.58) } \} \\ & \left\{ \begin{array}{l} f \cdot \underline{Nil} = \underline{0} \\ f \cdot Cons = add \cdot (id \times f) \end{array} \right. \end{array}$$

$$\equiv \quad \{ \text{ going pointwise } \}$$

$$\begin{cases} (f \cdot \underline{Nil})x = \underline{0}\, x \\ (f \cdot Cons)(a,x) = (add \cdot (id \times f))(a,x) \end{cases}$$

$$\equiv \quad \{ \text{ composition (2.6), constant (2.12), product (2.22) and definition of } add \ \}$$

$$\begin{cases} f\, Nil = 0 \\ f(Cons(a,x)) = a + f\, x \end{cases}$$

Note that we could have used out_{T} in diagram (3.31),

$$\begin{array}{ccc} \mathsf{T} & \xrightarrow{out_{\mathsf{T}}} & 1 + \mathbb{N}_0 \times \mathsf{T} \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle id+id\times f} \\ \mathbb{N}_0 & \xleftarrow[[\underline{0},add]]{} & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \end{array} \tag{3.34}$$

obtaining another version of the *definition* of f,

$$f \quad = \quad [\underline{0}, add] \cdot (id + id \times f) \cdot out_{\mathsf{T}} \tag{3.35}$$

which would lead to exactly the same pointwise recursive definition:

$$f = [\underline{0}, add] \cdot (id + id \times f) \cdot out_{\mathsf{T}}$$

$$\equiv \quad \{ \text{ (2.41) and identity } id \text{ on the righthand side } \}$$

$$f = [\underline{0}, add \cdot (id \times f)] \cdot out_{\mathsf{T}}$$

$$\equiv \quad \{ \text{ going pointwise on } out_{\mathsf{T}} \text{ (3.26) } \}$$

$$\begin{cases} f\, Nil = ([\underline{0}, add \cdot (id \times f)] \cdot out_{\mathsf{T}})Nil \\ f(Cons(a,x)) = ([\underline{0}, add \cdot (id \times f)] \cdot out_{\mathsf{T}})(a,x) \end{cases}$$

$$\equiv \quad \{ \text{ definition of } out_{\mathsf{T}} \text{ (3.26) } \}$$

$$\begin{cases} f\, Nil = ([\underline{0}, add \cdot (id \times f)] \cdot i_1)Nil \\ f(Cons(a,x)) = ([\underline{0}, add \cdot (id \times f)] \cdot i_2)(a,x) \end{cases}$$

$$\equiv \quad \{ \ +\text{-cancellation (2.38) } \}$$

$$\begin{cases} f\, Nil = \underline{0}\, Nil \\ f(Cons(a,x)) = (add \cdot (id \times f))\,(a,x) \end{cases}$$

$$\equiv \quad \{ \text{ simplification } \}$$

$$\begin{cases} f\, Nil = 0 \\ f(Cons(a,x)) = a + f\, x \end{cases}$$

Pointwise f mirrors the structure of type T in having has many definition clauses as constructors in T. Such functions are said to be defined *by induction on* the structure of their input type. If we repeat this calculation for $\mathbb{N}_0{}^\star$ instead of T, that is, for

$$out = (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?)$$

— recall (3.20) — taking place of out_{T}, we get a "more algorithmic" version of f:

$$\begin{array}{ll}
& f = [\underline{0}, add] \cdot (id + id \times f) \cdot (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?) \\
\equiv & \quad \{ \ + \text{-functor (2.42), identity and } \times \text{-absorption (2.25)} \ \} \\
& f = [\underline{0}, add] \cdot (! + \langle hd, f \cdot tl \rangle) \cdot (=_{[\,]}?) \\
\equiv & \quad \{ \ + \text{-absorption (2.41) and constant } \underline{0} \ \} \\
& f = [\underline{0}, add \cdot \langle hd, f \cdot tl \rangle] \cdot (=_{[\,]}?) \\
\equiv & \quad \{ \text{ going pointwise on guard } =_{[\,]}? \text{ (2.62) and simplifying } \} \\
& f\, l = \left\{ \begin{array}{lcl} l = [\,] & \Rightarrow & \underline{0}\, l \\ \neg(l = [\,]) & \Rightarrow & (add \cdot \langle hd, f \cdot tl \rangle)\, l \end{array} \right. \\
\equiv & \quad \{ \ \text{simplification} \ \} \\
& f\, l = \left\{ \begin{array}{lcl} l = [\,] & \Rightarrow & 0 \\ \neg(l = [\,]) & \Rightarrow & hd\, l + f(tl\, l) \end{array} \right.
\end{array}$$

The outcome of this calculation can be encoded in HASKELL syntax as

$$\begin{array}{l} f\ l \mid l \equiv [\,] = 0 \\ \quad \mid \mathsf{otherwise} = \mathsf{head}\ l + f\ (\mathsf{tail}\ l) \end{array}$$

or

$$f\ l = \mathbf{if}\ l \equiv [\,]\ \mathbf{then}\ 0\ \mathbf{else}\ \mathsf{head}\ l + f\ (\mathsf{tail}\ l)$$

both requiring the equality predicate $\equiv$ and destructors head and tail.

3.5 Synthesizing an inductive datatype

The issue which concerns us in this section dualizes what we have just dealt with: instead of analyzing or *observing* an inductive type such as T (3.25), we want to be

able to synthesize (generate) particular inhabitants of T. In other words, we want to be able to specify functions with signature $B \xrightarrow{f} \mathsf{T}$ for some given source type B. Let $B = \mathbb{N}_0$ and suppose we want f to generate, for a given natural number $n > 0$, the list containing all numbers less or equal to n in decreasing order

$$Cons(n, Cons(n-1, Cons(\ldots, Nil)))$$

or the empty list Nil, in case $n = 0$.

Let us try and draw a diagram similar to (3.34) applicable to the new situation. In trying to "re-use" this diagram, it is immediate that arrow f should be reversed. Bearing duality in mind, we may feel tempted to reverse all arrows just to see what happens. Identity functions are their own inverses, and in_{T} takes the place of out_{T}:

$$\begin{array}{ccc} \mathsf{T} & \xleftarrow{in_{\mathsf{T}}} & 1 + \mathbb{N}_0 \times \mathsf{T} \\ {\scriptstyle f}\uparrow & & \uparrow{\scriptstyle id+id\times f} \\ \mathbb{N}_0 & \dashrightarrow & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \end{array}$$

Interestingly enough, the bottom arrow is the one which is not obvious to reverse, meaning that we have to "invent" a particular destructor of $\mathbb{N}_0$, say

$$\mathbb{N}_0 \xrightarrow{g} 1 + \mathbb{N}_0 \times \mathbb{N}_0$$

fitting in the diagram and *generating* the particular computational effect we have in mind. Once we do this, a recursive definition for f will pop out immediately,

$$f \quad = \quad in_{\mathsf{T}} \cdot (id + id \times f) \cdot g \tag{3.36}$$

which is equivalent to:

$$f \quad = \quad [\underline{Nil}, Cons \cdot (id \times f)] \cdot g \tag{3.37}$$

Because we want $f\,0 = Nil$ to hold, g (the actual generator of the computation) should distinguish input 0 from all the others. One thus decomposes g as follows,

$$\mathbb{N}_0 \xrightarrow{=_0?} \mathbb{N}_0 + \mathbb{N}_0 \xrightarrow{!+h} 1 + \mathbb{N}_0 \times \mathbb{N}_0 \qquad (g)$$

leaving h to fill in. This will be a *split* providing, on the lefthand side, for the value to be $Cons$'ed to the output and, on the righthand side, for the "seed" to the

next recursive call. Since we want the output values to be produced contiguously and in decreasing order, we may define $h = \langle id, pred \rangle$ where, for $n > 0$,

$$pred\, n \stackrel{\text{def}}{=} n - 1 \tag{3.38}$$

computes the *predecessor* of n. Altogether, we have synthesized

$$g \;=\; (! + \langle id, pred \rangle) \cdot (=_0?) \tag{3.39}$$

Filling this in (3.37) we get

$$\begin{array}{ll} & f = [\underline{Nil}, Cons \cdot (id \times f)] \cdot (! + \langle id, pred \rangle) \cdot (=_0?) \\ \equiv & \quad \{ \text{ +-absorption (2.41) followed by } \times\text{-absorption (2.25) } \textit{etc.} \} \\ & f = [\underline{Nil}, Cons \cdot \langle id, f \cdot pred \rangle] \cdot (=_0?) \\ \equiv & \quad \{ \text{ going pointwise on guard } =_0? \text{ (2.62) and simplifying } \} \\ & f\, n = \left\{ \begin{array}{rcl} n = 0 & \Rightarrow & Nil \\ \neg(n = 0) & \Rightarrow & Cons(n, f\,(n-1)) \end{array} \right. \end{array}$$

which matches the function we had in mind:

$$\begin{array}{l} f\ n \\ \quad \mid n \equiv 0 = Nil \\ \quad \mid \mathsf{otherwise} = Cons\ (n, f\ (n-1)) \end{array}$$

We shall see briefly that the constructions of the f function adding up a list of numbers in the previous section and, in this section, of the f function generating a list of numbers are very standard in algorithm design and can be broadly generalized. Let us first introduce some standard terminology.

3.6 Introducing (list) catas, anas and hylos

Suppose that, back to section 3.4, we want to *multiply*, rather than add, the elements occurring in lists of type T (3.25). How much of the program synthesis effort presented there can be reused in the design of the new function?

It is intuitive that only the bottom arrow $\mathbb{N}_0 \xleftarrow{[\underline{0}, add]} 1 + \mathbb{N}_0 \times \mathbb{N}_0$ of diagram (3.34) needs to be replaced, because this is the only place where we can

specify that target datatype $\mathbb{N}_0$ is now regarded as the carrier of another (multiplicative rather than additive) monoidal structure,

$$\mathbb{N}_0 \xleftarrow{[\underline{1},mul]} 1 + \mathbb{N}_0 \times \mathbb{N}_0 \tag{3.40}$$

for $mul(x, y) \stackrel{\text{def}}{=} x\ y$. We are saying that the argument list is now to be reduced by the multiplication operator and that output value 1 is expected as the result of "nothing left to multiply".

Moreover, in the previous section we might have wanted our number-list generator to produce the list of even numbers smaller than a given number, in decreasing order (see exercise 3.9). Intuition will once again help us in deciding that only arrow g in (3.36) needs to be updated.

The following diagrams generalize both constructions by leaving such bottom arrows unspecified,

$$\begin{array}{ccc} \mathsf{T} & \xrightarrow{out_{\mathsf{T}}} & 1 + \mathbb{N}_0 \times \mathsf{T} \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle id+id\times f} \\ B & \xleftarrow[g]{} & 1 + \mathbb{N}_0 \times B \end{array} \qquad \begin{array}{ccc} \mathsf{T} & \xleftarrow{in_{\mathsf{T}}} & 1 + \mathbb{N}_0 \times \mathsf{T} \\ {\scriptstyle f}\uparrow & & \uparrow{\scriptstyle id+id\times f} \\ B & \xrightarrow[g]{} & 1 + \mathbb{N}_0 \times B \end{array} \tag{3.41}$$

and express their duality (*cf.* the directions of the arrows). It so happens that, for each of these diagrams, f is uniquely dependent on the g arrow, that is to say, each particular instantiation of g will determine the corresponding f. So both gs can be regarded as "seeds" or "genetic material" of the f functions they uniquely define [5].

Following the standard terminology, we express these facts by writing $f = (\!|g|\!)$ with respect to the lefthand side diagram and by writing $f = [\!(g)\!]$ with respect to the righthand side diagram. Read $(\!|g|\!)$ as "the T-*catamorphism* induced by g" and $[\!(g)\!]$ as "the T-*anamorphism* induced by g". This terminology is derived from the Greek words $\kappa\alpha\tau\alpha$ (cata) and $\alpha\nu\alpha$ (ana) meaning, respectively, "downwards" and "upwards" (compare with the direction of the f arrow in each diagram). The exchange of parentheses "( )" and "[ ]" in double parentheses "$(\!|\ |\!)$" and "$[\!(\ )\!]$" is aimed at expressing the duality of both concepts.

We shall have a lot to say about catamorphisms and anamorphisms of a given type such as T. For the moment, it suffices to say that

[5] The theory which supports the statements of this paragraph will not be dealt with until chapter 9 .

- the T-catamorphism induced by $B \xleftarrow{g} 1 + \mathbb{N}_0 \times B$ is the unique function $B \xleftarrow{(\!|g|\!)} \mathsf{T}$ which obeys to property (or is defined by)

$$(\!|g|\!) \quad = \quad g \cdot (id + id \times (\!|g|\!)) \cdot out_{\mathsf{T}} \tag{3.42}$$

  which is the same as

$$(\!|g|\!) \cdot in_{\mathsf{T}} \quad = \quad g \cdot (id + id \times (\!|g|\!)) \tag{3.43}$$

- given $B \xrightarrow{g} 1 + \mathbb{N}_0 \times B$ the T-anamorphism induced by g is the unique function $B \xrightarrow{[\!(g)\!]} \mathsf{T}$ which obeys to property (or is defined by)

$$[\!(g)\!] \quad = \quad in_{\mathsf{T}} \cdot (id + id \times [\!(g)\!]) \cdot g \tag{3.44}$$

From (3.41) it can be observed that T can act as a mediator between any T-anamorphism and any T-catamorphism, that is to say, $B \xleftarrow{(\!|g|\!)} \mathsf{T}$ composes with $\mathsf{T} \xleftarrow{[\!(h)\!]} C$, for some $C \xrightarrow{h} 1 + \mathbb{N}_0 \times C$. In other words, a T-catamorphism call always observe (consume) the output of a T-anamorphism. The latter produces a list of $\mathbb{N}_0$s which is consumed by the former. This is depicted in the diagram which follows:

$$\begin{array}{ccc} B & \xleftarrow{\quad g \quad} & 1 + \mathbb{N}_0 \times B \\ {\scriptstyle (\!|g|\!)}\uparrow & & \uparrow{\scriptstyle id+id\times(\!|g|\!)} \\ \mathsf{T} & \xleftarrow{\quad in_{\mathsf{T}} \quad} & 1 + \mathbb{N}_0 \times \mathsf{T} \\ {\scriptstyle [\!(h)\!]}\uparrow & & \uparrow{\scriptstyle id+id\times[\!(h)\!]} \\ C & \xrightarrow[\quad h \quad]{} & 1 + \mathbb{N}_0 \times C \end{array} \tag{3.45}$$

What can we say about the $(\!|g|\!) \cdot [\!(h)\!]$ composition? It is a function from C to B which resorts to T as an *intermediate* data-structure and can be subject to the following calculation (*cf.* outermost rectangle in (3.45)):

$$\begin{array}{ll} & (\!|g|\!) \cdot [\!(h)\!] = g \cdot (id + id \times (\!|g|\!)) \cdot (id + id \times [\!(h)\!]) \cdot h \\ \equiv & \quad \{ \ + \text{-functor (2.42)} \ \} \\ & (\!|g|\!) \cdot [\!(h)\!] = g \cdot ((id \cdot id) + (id \times (\!|g|\!)) \cdot (id \times [\!(h)\!])) \cdot h \\ \equiv & \quad \{ \ \text{identity and} \times \text{-functor (2.28)} \ \} \\ & (\!|g|\!) \cdot [\!(h)\!] = g \cdot (id + id \times (\!|g|\!) \cdot [\!(h)\!]) \cdot h \end{array}$$

This calculation shows how to define $C \xleftarrow{(\!|g|\!)\cdot[\!(h)\!]} B$ in one go, that is to say, doing without any intermediate data-structure:

$$\begin{array}{ccc} B & \xleftarrow{\quad g \quad} & 1 + \mathbb{N}_0 \times B \\ {\scriptstyle (\!|g|\!)\cdot[\!(h)\!]}\uparrow & & \uparrow{\scriptstyle id+id\times(\!|g|\!)\cdot[\!(h)\!]} \\ C & \xrightarrow[\quad h \quad]{} & 1 + \mathbb{N}_0 \times C \end{array} \tag{3.46}$$

As an example, let us see what comes out of $(\!|g|\!) \cdot [\!(h)\!]$ for h and g respectively given by (3.39) and (3.40):

$$\begin{array}{ll} & (\!|g|\!) \cdot [\!(h)\!] = g \cdot (id + id \times (\!|g|\!) \cdot [\!(h)\!]) \cdot h \\ \equiv & \quad \{ \ (\!|g|\!) \cdot [\!(h)\!] \text{ abbreviated to } f \text{ and instantiating } h \text{ and } g \ \} \\ & f = [\underline{1}, mul] \cdot (id + id \times f) \cdot (! + \langle id, pred \rangle) \cdot (=_0?) \\ \equiv & \quad \{ \ + \text{-functor (2.42) and identity} \ \} \\ & f = [\underline{1}, mul] \cdot (! + (id \times f) \cdot \langle id, pred \rangle) \cdot (=_0?) \\ \equiv & \quad \{ \ \times \text{-absorption (2.25) and identity} \ \} \\ & f = [\underline{1}, mul] \cdot (! + \langle id, f \cdot pred \rangle) \cdot (=_0?) \\ \equiv & \quad \{ \ + \text{-absorption (2.41) and constant } \underline{1} \text{ (2.15)} \ \} \\ & f = [\underline{1}, mul \cdot \langle id, f \cdot pred \rangle] \cdot (=_0?) \\ \equiv & \quad \{ \ \text{McCarthy conditional (2.61)} \ \} \\ & f = (=_0?) \rightarrow \underline{1}, mul \cdot \langle id, f \cdot pred \rangle \end{array}$$

Going pointwise, we get — via (2.61) —

$$\begin{array}{lll} f\,0 & = & [\underline{1}, mul \cdot \langle id, f \cdot pred \rangle](i_1\,0) \\ & = & \quad \{ \ + \text{-cancellation (2.38)} \ \} \\ & & \underline{1}\,0 \\ & = & \quad \{ \ \text{constant function (2.12)} \ \} \\ & & 1 \end{array}$$

and

$$f(n+1) \ = \ [\underline{1}, mul \cdot \langle id, f \cdot pred \rangle](i_2(n+1))$$

$$
\begin{array}{ll}
= & \{\ +\text{-cancellation (2.38)}\ \} \\
& mul \cdot \langle id, f \cdot pred \rangle (n+1) \\
= & \{\ \text{pointwise definitions of } \textit{split}\text{, identity, predecessor and } mul\ \} \\
& (n+1) \times f\, n
\end{array}
$$

In summary, f is but the well-known factorial function:

$$
\begin{cases} f\, 0 = 1 \\ f(n+1) = (n+1) \times f\, n \end{cases}
$$

This result comes to no surprise if we look at diagram (3.45) for the particular g and h we have considered above and recall a popular "definition" of factorial:

$$
n! \ = \ \underbrace{n \times (n-1) \times \ldots \times 1}_{n \text{ times}} \tag{3.47}
$$

In fact, $[\!(h)\!]\, n$ produces T-list

$$
Cons(n, Cons(n-1, \ldots Cons(1, Nil)))
$$

as an intermediate data-structure which is consumed by $(\!|g|\!)$, the effect of which is but the "replacement" of $Cons$ by $\times$ and Nil by 1, therefore accomplishing (3.47) and realizing the computation of factorial.

The moral of this example is that a function as simple as factorial can be *decomposed* into two components (producer/consumer functions) which share a common intermediate inductive datatype. The producer function is an anamorphism which "represents" or produces a "view" of its input argument as a value of the intermediate datatype. The consumer function is a catamorphism which reduces this intermediate data-structure and produces the final result. Like factorial, many functions can be handsomely expressed by a $(\!|g|\!) \cdot [\!(h)\!]$ composition for a suitable choice of the intermediate type, and of g and h. The intermediate data-structure is said to be *virtual* in the sense that it only exists as a means to induce the associated pattern of recursion and disappears by calculation.

The composition $(\!|g|\!) \cdot [\!(h)\!]$ of a T-catamorphism with a T-anamorphism is called a T-*hylomorphism* [6] and is denoted by $[\![g, h]\!]$. Because g and h fully determine the behaviour of the $[\![g, h]\!]$ function, they can be regarded as the "genes" of

[6] This terminology is derived from the Greek word $\upsilon\lambda o\sigma$ (hylos) meaning "matter".

the function they define. As we shall see, this analogy with biology will prove specially useful for algorithm analysis and classification.

Exercise 3.7. *A way of computing n^2, the square of a given natural number n, is to sum up the n first odd numbers. In fact, $1^2 = 1$, $2^2 = 1 + 3$, $3^2 = 1 + 3 + 5$,* etc.*, $n^2 = (2n - 1) + (n - 1)^2$. Following this hint, express function*

$$sq\, n \stackrel{\text{def}}{=} n^2 \tag{3.48}$$

as a T*-hylomorphism and encode it in* HASKELL.
□

Exercise 3.8. *Write function x^n as a* T*-hylomorphism and encode it in* HASKELL.
□

Exercise 3.9. *The following function in* HASKELL *computes the* T*-sequence of all even numbers less or equal to n:*

$$\begin{array}{l} f\ n = \\ \quad \textbf{if}\ n \leqslant 1\ \textbf{then}\ Nil\ \textbf{else}\ Cons\ (m, f\ (m-2)) \\ \quad\quad \textbf{where}\ m = \textbf{if}\ even\ n\ \textbf{then}\ n\ \textbf{else}\ n-1 \end{array}$$

Find its "genetic material", that is, function g such that $f = [\![g]\!]$ in

$$\begin{array}{ccc} \mathsf{T} & \xleftarrow{in_{\mathsf{T}}} & 1 + \mathbb{N}_0 \times \mathsf{T} \\ {\scriptstyle [\![g]\!]}\uparrow & & \uparrow{\scriptstyle id+id\times[\![g]\!]} \\ \mathbb{N}_0 & \xrightarrow[g]{} & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \end{array}$$

□

3.7 Inductive types more generally

So far we have focussed our attention exclusively to a particular inductive type T (3.30) — that of finite sequences of non-negative integers. This is, of course, of

a very limited scope. First, because one could think of finite sequences of other datatypes, *e.g.* Booleans or many others. Second, because other datatypes such as trees, hash-tables *etc.* exist which our notation and method should be able to take into account.

Although a generic theory of arbitrary datatypes requires a theoretical elaboration which cannot be explained at once, we can move a step further by taking the two observations above as starting points. We shall start from the latter in order to talk generically about inductive types. Then we introduce parameterization and functorial behaviour.

Suppose that, as a mere notational convention, we abbreviate every expression of the form "$1 + \mathbb{N}_0 \times \ldots$" occurring in the previous section by "$\mathsf{F}\ldots$", *e.g.* $1 + \mathbb{N}_0 \times B$ by $\mathsf{F}\,B$, *e.g.* $1 + \mathbb{N}_0 \times \mathsf{T}$ by $\mathsf{F}\,\mathsf{T}$

$$\mathsf{T} \underset{in_{\mathsf{T}}}{\overset{out_{\mathsf{T}}}{\cong}} \mathsf{F}\,\mathsf{T} \tag{3.49}$$

etc. This is the same as introducing a datatype-level operator

$$\mathsf{F}\,X \stackrel{\text{def}}{=} 1 + \mathbb{N}_0 \times X \tag{3.50}$$

which maps every datatype A into datatype $1 + \mathbb{N}_0 \times A$. Operator F captures the pattern of recursion which is associated to so-called "right" lists (of non-negative integers), that is, lists which grow to the right. The slightly different pattern $\mathsf{G}\,X \stackrel{\text{def}}{=} 1 + X \times \mathbb{N}_0$ will generate a different, although related, inductive type

$$X \quad \cong 1 + X \times \mathbb{N}_0 \tag{3.51}$$

— that of so-called "left" lists (of non-negative integers). And it is not difficult to think of the pattern which is merges both right and left lists and gives rise to bi-linear lists, better known as *binary trees*:

$$X \quad \cong 1 + X \times \mathbb{N}_0 \times X \tag{3.52}$$

One may think of many other expressions $\mathsf{F}\,X$ and guess the inductive datatype they generate, for instance $\mathsf{H}\,X \stackrel{\text{def}}{=} \mathbb{N}_0 + \mathbb{N}_0 \times X$ generating non-empty lists of non-negative integers ($\mathbb{N}_0^+$). The general rule is that, given an inductive datatype definition of the form

$$X \quad \cong \quad \mathsf{F}\,X \tag{3.53}$$

(also called a domain equation), its pattern of recursion is captured by a so-called *functor* F.

3.8 Functors

The concept of a functor F, borrowed from category theory, is a most generic and useful device in programming [7]. As we have seen, F can be regarded as a datatype constructor which, given datatype A, builds a more elaborate datatype $\mathsf{F}\ A$; given another datatype B, builds a similarly elaborate datatype $\mathsf{F}\ B$; and so on. But what is more important and has the most beneficial consequences is that, if F is regarded as a functor, then its data-structuring effect extends smoothly to functions in the following way: suppose that $B \xleftarrow{f} A$ is a function which observes A into B, which are parameters of $\mathsf{F}\ A$ and $\mathsf{F}\ B$, respectively. By definition, if F is a functor then $\mathsf{F}\ B \xleftarrow{\mathsf{F}\ f} \mathsf{F}\ A$ exists for every such f:

$$\begin{array}{ccc} A & \cdots\cdots & \mathsf{F}\ A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\ f} \\ B & \cdots\cdots & \mathsf{F}\ B \end{array}$$

$\mathsf{F}\ f$ extends f to F-structures and will, by definition, obey to two very basic properties: it commutes with identity

$$\mathsf{F}\ id_A \quad = \quad id_{(\mathsf{F}\ A)} \tag{3.54}$$

and with composition

$$\mathsf{F}(g \cdot h) \quad = \quad (\mathsf{F}\ g) \cdot (\mathsf{F}\ h) \tag{3.55}$$

Two simple examples of a functor follow:

- Identity functor: define $\mathsf{F}\ X = X$, for every datatype X, and $\mathsf{F}\ f = f$. Properties (3.54) and (3.55) hold trivially just by removing symbol F wherever it occurs.

[7] The category theory practitioner must be warned of the fact that the word *functor* is used here in a too restrictive way. A proper (generic) definition of a functor will be provided later in this book.

Data construction	Universal construct	Functor	Description
$A \times B$	$\langle f, g \rangle$	$f \times g$	Product
$A + B$	$[f, g]$	$f + g$	Coproduct
B^A	$\overline{f}$	f^A	Exponential

Table 3.1: Datatype constructions and associated operators.

- Constant functors: for a given C, define $\mathsf{F}\, X = C$ (for all datatypes X) and $\mathsf{F}\, f = id_C$, as expressed in the following diagram:

$$\begin{array}{ccc} A & \cdots\cdots & C \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle id_C} \\ B & \cdots\cdots & C \end{array}$$

  Properties (3.54) and (3.55) hold trivially again.

In the same way functions can be unary, binary, *etc.*, we can have functors with more than one argument. So we get binary functors (also called *bifunctors*), ternary functors *etc.*. Of course, properties (3.54) and (3.55) have to hold for every parameter of an n-ary functor. For a binary functor B, for instance, equation (3.54) becomes

$$\mathsf{B}\,(id_A, id_B) \quad = \quad id_{\mathsf{B}\,(A,B)} \tag{3.56}$$

and equation (3.55) becomes

$$\mathsf{B}\,(g \cdot h, i \cdot j) \quad = \quad \mathsf{B}\,(g, i) \cdot \mathsf{B}\,(h, j) \tag{3.57}$$

Product and coproduct are typical examples of bifunctors. In the former case one has $\mathsf{B}\,(A, B) = A \times B$ and $\mathsf{B}\,(f, g) = f \times g$ — recall (2.22). Properties (2.29) and (2.28) instantiate (3.56) and (3.57), respectively, and this explains why we called them the functorial properties of product. In the latter case, one has $\mathsf{B}\,(A, B) = A + B$ and $\mathsf{B}\,(f, g) = f + g$ — recall (2.37) — and functorial properties (2.43) and (2.42). Finally, exponentiation is a functorial construction too: assuming A, one has $\mathsf{F}\, X \stackrel{\text{def}}{=} X^A$ and $\mathsf{F}\, f \stackrel{\text{def}}{=} \overline{f \cdot ap}$ and functorial properties (2.79) and (2.80). All this is summarized in table 3.1.

Such as functions, functors may compose with each other in the obvious way: the composition of F and G, denoted $\mathsf{F} \cdot \mathsf{G}$, is defined by

$$(\mathsf{F} \cdot \mathsf{G})X \stackrel{\text{def}}{=} \mathsf{F}\,(\mathsf{G}\,X) \tag{3.58}$$

$$(\mathsf{F} \cdot \mathsf{G})f \stackrel{\text{def}}{=} \mathsf{F}\,(\mathsf{G}\,f) \tag{3.59}$$

3.9 Polynomial functors

We may put constant, product, coproduct and identity functors together to obtain so-called *polynomial functors*, which are described by polynomial expressions, for instance

$$\mathsf{F}\,X = 1 + A \times X$$

— recall (3.16). A polynomial functor is either

- a constant functor or the identity functor, or
- the (finitary) product or coproduct (sum) of other polynomial functors, or
- the composition of other polynomial functors.

So the effect on arrows of a polynomial functor is computed in an easy and structured way, for instance:

$$\begin{array}{ll}
\mathsf{F}\,f & = \quad (1 + A \times X)f \\
& = \qquad \{ \text{ sum of two functors where } A \text{ is a constant and } X \text{ is a variable } \} \\
& \quad (1)f + (A \times X)f \\
& = \qquad \{ \text{ constant functor and product of two functors } \} \\
& \quad id_1 + (A)f \times (X)f \\
& = \qquad \{ \text{ constant functor and identity functor } \} \\
& \quad id_1 + id_A \times f \\
& = \qquad \{ \text{ subscripts dropped for simplicity } \} \\
& \quad id + id \times f
\end{array}$$

So, $1 + A \times f$ denotes the same as $id_1 + id_A \times f$, or even the same as $id + id \times f$ if one drops the subscripts.

It should be clear at this point that what was referred to in section 2.10 as a "symbolic pattern" applicable to both datatypes and arrows is after all a functor in the mathematical sense. The fact that the same polynomial expression is used to denote both the data and the operators which structurally transform such data is of great conceptual economy and practical application. For instance, once polynomial functor (3.50) is assumed, the diagrams in (3.41) can be written as simply as

$$\begin{array}{ccc} \mathsf{T} & \xrightarrow{out_{\mathsf{T}}} & \mathsf{F}\,\mathsf{T} \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,f} \\ B & \xleftarrow[g]{} & \mathsf{F}\,B \end{array} \qquad \begin{array}{ccc} \mathsf{T} & \xleftarrow{in_{\mathsf{T}}} & \mathsf{F}\,\mathsf{T} \\ {\scriptstyle f}\uparrow & & \uparrow{\scriptstyle \mathsf{F}\,f} \\ B & \xrightarrow[g]{} & \mathsf{F}\,B \end{array} \tag{3.60}$$

It is useful to know that, thanks to the isomorphism laws studied in chapter 2, every polynomial functor F may be put into the canonical form,

$$\begin{array}{lcl} \mathsf{F}\,X & \cong & C_0 + (C_1 \times X) + (C_2 \times X^2) + \cdots + (C_n \times X^n) \\ & = & \sum_{i=0}^{n} C_i \times X^i \end{array} \tag{3.61}$$

and that *Newton's binomial formula*

$$(A+B)^n \;\cong\; \sum_{p=0}^{n} {}^nC_p \times A^{n-p} \times B^p \tag{3.62}$$

can be used in such conversions. These are performed up to isomorphism, that is to say, after the conversion one gets a different but isomorphic datatype. Consider, for instance, functor

$$\mathsf{F}\,X \stackrel{\text{def}}{=} A \times (1+X)^2$$

(where A is a constant datatype) and check the following reasoning:

$$\begin{array}{lcl} \mathsf{F}\,X & = & A \times (1+X)^2 \\ & \cong & \quad \{ \text{ law (2.96) } \} \\ & & A \times ((1+X) \times (1+X)) \\ & \cong & \quad \{ \text{ law (2.50) } \} \\ & & A \times ((1+X) \times 1 + (1+X) \times X)) \\ & \cong & \quad \{ \text{ laws (2.90), (2.31) and (2.50) } \} \end{array}$$

$$
\begin{array}{ll}
& A \times ((1 + X) + (1 \times X + X \times X)) \\
\cong & \quad \{ \text{ laws (2.90) and (2.96) } \} \\
& A \times ((1 + X) + (X + X^2)) \\
\cong & \quad \{ \text{ law (2.46) } \} \\
& A \times (1 + (X + X) + X^2) \\
\cong & \quad \{ \text{ canonical form obtained via laws (2.50) and (2.97) } \} \\
& \underbrace{A}_{C_0} + \underbrace{A \times 2}_{C_1} \times X + \underbrace{A}_{C_2} \times X^2
\end{array}
$$

Exercise 3.10. *Synthesize the isomorphism* $A + A \times 2 \times X + A \times X^2 \xleftarrow{\nu} A \times (1 + X^2)$ *implicit in the above reasoning.*
□

3.10 Polynomial inductive types

An inductive datatype is said to be *polynomial* wherever its pattern of recursion is described by a polynomial functor, that is to say, wherever F in equation (3.53) is polynomial. For instance, datatype T (3.30) is polynomial ($n = 1$) and its associated polynomial functor is canonically defined with coefficients $C_0 = 1$ and $C_1 = \mathbb{N}_0$. For reasons that will become apparent later on, we shall always impose $C_0 \neq 0$ to hold in a *polynomial* datatype expressed in canonical form.

Polynomial types are easy to encode in HASKELL wherever the associated functor is in canonical polynomial form, that is, wherever one has

$$\mathsf{T} \underbrace{\xleftarrow{\quad\cong\quad}}_{in_{\mathsf{T}}} \sum_{i=0}^{n} C_i \times \mathsf{T}^i \tag{3.63}$$

Then we have

$$in_{\mathsf{T}} \stackrel{\text{def}}{=} [\, f_1, \ldots, f_n \,]$$

where, for $i = 1, n$, f_i is an arrow of type $\mathsf{T} \leftarrow C_i \times \mathsf{T}^i$. Since n is finite, one may expand exponentials according to (2.96) and encode this in HASKELL as follows:

```
          data T = C0 ||
          C1 (C1,T) ||
          C2 (C2,(T,T)) ||
          ... ||
Cn (Cn,(T, ..., T))
```

Of course the choice of symbol `Ci` to realize each f_i is arbitrary [8]. Several instances of polynomial inductive types (in canonical form) will be mentioned in section 3.14. Section 3.18 will address the conversion between inductive datatypes induced by so-called *natural transformations*.

The concepts of catamorphism, anamorphism and hylomorphism introduced in section 3.6 can be extended to arbitrary polynomial types. We devote the following sections to explaining catamorphisms in the polynomial setting. Polynomial anamorphisms and hylomorphisms will not be dealt with until chapter 9.

3.11 F-algebras and F-homomorphisms

Our interest in polynomial types is basically due to the fact that, for polynomial F, equation (3.53) always has a particularly interesting solution which corresponds to our notion of a recursive datatype.

In order to explain this, we need two notions which are easy to understand: first, that of an F-*algebra*, which simply is any function α of signature $A \xleftarrow{\alpha} \mathsf{F}\, A$. A is called the *carrier* of F-algebra α and contains the values which α manipulates by computing new A-values out of existing ones, according to the F-pattern (the "type" of the algebra). As examples, consider $[\underline{0}, add]$ (3.29) and in_{T} (3.30), which are both algebras of type $\mathsf{F}\, X = 1 + \mathbb{N}_0 \times X$. The type of an algebra clearly determines its form. For instance, any algebra α of type $\mathsf{F}\, X = 1 + X \times X$ will be of form $[\alpha_1, \alpha_2]$, where α_1 is a constant and α_2 is a binary operator. So monoids are algebras of this type [9].

Secondly, we introduce the notion of an F-*homomorphism* which is but a func-

[8] A more traditional (but less close to (3.63)) encoding will be

```
data T = C0 | C1 C1 T | C2 C2 T T | ...  | Cn Cn T ...  T
```
(3.64)

delivering every constructor in curried form.

[9] But not every algebra of this type is a monoid, since the type of an algebra only fixes its syntax and does not impose any properties such as associativity, *etc.*

tion observing a particular F-algebra α into another F-algebra β:

$$\begin{array}{ccc} A & \xleftarrow{\alpha} & \mathsf{F}\,A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,f} \\ B & \xleftarrow[\beta]{} & \mathsf{F}\,B \end{array} \qquad f \cdot \alpha = \beta \cdot (\mathsf{F}\,f) \tag{3.65}$$

Clearly, f can be regarded as a structural translation between A and B, that is, A and B have a similar structure [10]. Note that — thanks to (3.54) — identity functions are always (trivial) F-homomorphisms and that — thanks to (3.55) — these homomorphisms compose, that is, the composition of two F-homomorphisms is an F-homomorphism.

3.12 F-catamorphisms

An F-algebra can be epic, monic or both, that is, iso. Iso F-algebras are particularly relevant to our discussion because they describe solutions to the $X \cong \mathsf{F}\,X$ equation (3.53). Moreover, for polynomial F a particular iso F-algebra always exists, which is denoted by $\mu\mathsf{F} \xleftarrow{in} \mathsf{F}\,\mu\mathsf{F}$ and has special properties. First, its carrier is the smallest among the carriers of other iso F-algebras, and this is why it is denoted by $\mu\mathsf{F}$ — μ for "minimal" [11]. Second, it is the so-called *initial* F-algebra. What does this mean?

It means that, for every F-algebra α there exists one and only one F-homomorphism between in and α. This unique arrow mediating in and α is therefore determined by α itself, and is called the F-*catamorphism* generated by α. This construct, which was introduced in 3.6, is in general denoted by $(\!|\alpha|\!)_{\mathsf{F}}$:

$$\begin{array}{ccc} \mu\mathsf{F} & \xleftarrow{in} & \mathsf{F}\,\mu\mathsf{F} \\ {\scriptstyle f=(\!|\alpha|\!)_{\mathsf{F}}}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,(\!|\alpha|\!)_{\mathsf{F}}} \\ A & \xleftarrow[\alpha]{} & \mathsf{F}\,A \end{array} \tag{3.66}$$

We will drop the F subscript in $(\!|\alpha|\!)_{\mathsf{F}}$ wherever deducible from the context, and often call α the "gene" of $(\!|\alpha|\!)_{\mathsf{F}}$.

[10] Cf. *homomorphism* = *homo* (the same) + *morphos* (structure, shape).

[11] $\mu\mathsf{F}$ means the least fixpoint solution of equation $X \cong \mathsf{F}\,X$, as will be described in chapter 9 .

As happens with *splits*, *eithers* and transposes, the uniqueness of the catamorphism construct is captured by a universal property established in the class of all F-homomorphisms:

$$k = (\!|\alpha|\!) \quad \Leftrightarrow \quad k \cdot in = \alpha \cdot \mathsf{F}\, k \tag{3.67}$$

According to the experience gathered from section 2.12 onwards, a few properties can be expected as consequences of (3.67). For instance, one may wonder about the "gene" of the identity catamorphism. Just let $k = id$ in (3.67) and see what happens:

$$\begin{aligned}
& id = (\!|\alpha|\!) \Leftrightarrow id \cdot in = \alpha \cdot \mathsf{F}\, id \\
= & \qquad \{ \text{ identity (2.10) and } \mathsf{F} \text{ is a functor (3.54) } \} \\
& id = (\!|\alpha|\!) \Leftrightarrow in = \alpha \cdot id \\
= & \qquad \{ \text{ identity (2.10) once again } \} \\
& id = (\!|\alpha|\!) \Leftrightarrow in = \alpha \\
= & \qquad \{ \ \alpha \text{ replaced by } in \text{ and simplifying } \} \\
& id = (\!|in|\!)
\end{aligned}$$

Thus one finds out that the genetic material of the identity catamorphism is the initial algebra in. Which is the same as establishing the *reflection property* of catamorphisms:

Cata-reflection :

$$\begin{array}{ccc}
\mu\mathsf{F} & \xleftarrow{in} & \mathsf{F}\,\mu\mathsf{F} \\
{\scriptstyle (\!|in|\!)}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,(\!|in|\!)} \\
\mu\mathsf{F} & \xleftarrow[in]{} & \mathsf{F}\,\mu\mathsf{F}
\end{array} \qquad\qquad (\!|in|\!) = id_{\mu\mathsf{F}} \tag{3.68}$$

In a more intuitive way, one might have observed that $(\!|in|\!)$ is, by definition of in, the unique arrow mediating $\mu\mathsf{F}$ and itself. But another arrow of the same type is already known: the identity $id_{\mu\mathsf{F}}$. So these two arrows must be the same.

Another property following immediately from (3.67), for $k = (\!|\alpha|\!)$, is

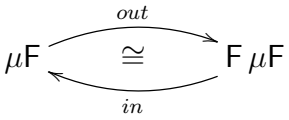
Cata-cancellation :

$$(\!|\alpha|\!) \cdot in = \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) \tag{3.69}$$

Because in is iso, this law can be rephrased as follows

$$(\!|\alpha|\!) = \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) \cdot out \tag{3.70}$$

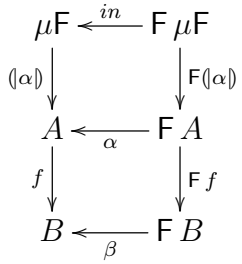
where out denotes the inverse of in:



Now, let f be F-homomorphism (3.65) between F-algebras α and β. How does it relate to $(\!|\alpha|\!)$ and $(\!|\beta|\!)$? Note that $f \cdot (\!|\alpha|\!)$ is an arrow mediating $\mu\mathsf{F}$ and B. But B is the carrier of β and $(\!|\beta|\!)$ is the unique arrow mediating $\mu\mathsf{F}$ and B. So the two arrows are the same:

Cata-fusion :

$$f \cdot (\!|\alpha|\!) = (\!|\beta|\!) \quad \text{if} \quad f \cdot \alpha = \beta \cdot \mathsf{F}\,f \tag{3.71}$$



Of course, this law is also a consequence of the universal property, for $k = f \cdot (\!|\alpha|\!)$:

$$\begin{array}{lll}
f \cdot (\!|\alpha|\!) = (\!|\beta|\!) & \Leftrightarrow & (f \cdot (\!|\alpha|\!)) \cdot in = \beta \cdot \mathsf{F}\,(f \cdot (\!|\alpha|\!)) \\
& \Leftrightarrow & \quad \{ \text{ composition is associative and F is a functor (3.55) } \} \\
& & f \cdot ((\!|\alpha|\!) \cdot in) = \beta \cdot (\mathsf{F}\,f) \cdot (\mathsf{F}\,(\!|\alpha|\!)) \\
& \Leftrightarrow & \quad \{ \text{ cata-cancellation (3.69) } \} \\
& & f \cdot \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) = \beta \cdot \mathsf{F}\,f \cdot \mathsf{F}\,(\!|\alpha|\!) \\
& \Leftarrow & \quad \{ \text{ require } f \text{ to be a F-homomorphism (3.65) } \} \\
& & f \cdot \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) = f \cdot \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) \wedge f \cdot \alpha = \beta \cdot \mathsf{F}\,f \\
& \Leftrightarrow & \quad \{ \text{ simplify } \} \\
& & f \cdot \alpha = \beta \cdot \mathsf{F}\,f
\end{array}$$

The presentation of the *absorption* property of catamorphisms entails the very important issue of parameterization and deserves to be treated in a separate section, as follows.

3.13 Parameterization and type functors

By analogy with what we have done about *splits* (product), *eithers* (coproduct) and transposes (exponential), we now look forward to identifying F-catamorphisms which exhibit functorial behaviour.

Suppose that one wishes to square all numbers which are members of lists of type T (3.30). It can be checked that

$$(\!|[\underline{Nil}, Cons \cdot (sq \times id)]|\!) \tag{3.72}$$

will do this for us, where $\mathbb{N}_0 \xleftarrow{sq} \mathbb{N}_0$ is given by (3.48). This catamorphism, which converted to pointwise notation is nothing but function h which follows

$$\begin{cases} h\, Nil = Nil \\ h(Cons(a, l)) = Cons(sq\, a, h\, l) \end{cases}$$

maps type T to itself. This is because sq maps $\mathbb{N}_0$ to $\mathbb{N}_0$. Now suppose that, instead of sq, one would like to apply a given function $B \xleftarrow{f} \mathbb{N}_0$ (for some B other than $\mathbb{N}_0$) to all elements of the argument list. It is easy to see that it suffices to replace f for sq in (3.72). However, the output type no longer is T, but rather type $\mathsf{T}' \cong 1 + B \times \mathsf{T}'$.

Types T and T' are very close to each other. They share the same "shape" (recursive pattern) and only differ with respect to the type of elements — $\mathbb{N}_0$ in T and B in T'. This suggests that these two types can be regarded as instances of a more generic list datatype List

$$\mathsf{List}\, X \underbrace{\xleftarrow{in=[\underline{Nil},Cons]}}_{} 1 + X \times \mathsf{List}\, X \qquad \cong \tag{3.73}$$

in which the type of elements X is allowed to vary. Thus one has $\mathsf{T} = \mathsf{List}\, \mathbb{N}_0$ and $\mathsf{T}' = \mathsf{List}\, B$.

By inspection, it can be checked that, for every $B \xleftarrow{f} A$,

$$(\!|[\underline{Nil}, Cons \cdot (f \times id)]|\!) \tag{3.74}$$

maps $\mathsf{List}\, A$ to $\mathsf{List}\, B$. Moreover, for $f = id$ one has:

$$\begin{aligned} & (\!|[\underline{Nil}, Cons \cdot (id \times id)]|\!) \\ = & \quad \{ \text{ by the } \times\text{-functor-id property (2.29) and identity } \} \end{aligned}$$

$$
\begin{array}{ll}
 & (\!|[\underline{Nil}, Cons]|\!) \\
= & \qquad \{ \text{ cata-reflection (3.68) } \} \\
 & id
\end{array}
$$

Therefore, by defining

$$\mathsf{List}\, f \stackrel{\text{def}}{=} (\!|[\underline{Nil}, Cons \cdot (f \times id)]|\!)$$

what we have just seen can be written thus:

$$\mathsf{List}\, id_A = id_{\mathsf{List}\, A}$$

This is nothing but law (3.54) for F replaced by List. Moreover, it will not be too difficult to check that

$$\mathsf{List}\, (g \cdot f) = \mathsf{List}\, g \cdot \mathsf{List}\, f$$

also holds — *cf.* (3.55). Altogether, this means that List can be regarded as a functor.

In programming terminology one says that $\mathsf{List}\, X$ (the "lists of Xs datatype") is *parametric* and that, by instantiating parameter X, one gets ground lists such as lists of integers, booleans, *etc.* The illustration above deepens one's understanding of parameterization by identifying the functorial behaviour of the parametric datatype along with its parameter instantiations.

All this can be broadly generalized and leads to what is commonly known by a *type functor*. First of all, it should be clear that the generic format

$$\mathsf{T} \cong \mathsf{F}\, \mathsf{T}$$

adopted so far for the definition of an inductive type is not sufficiently detailed because it does not provide a parametric view of T. For simplicity, let us suppose (for the momement) that only one parameter is identified in T. Then we may factor this out via *type variable* X and write (overloading symbol T)

$$\mathsf{T}\, X \cong \mathsf{B}(X, \mathsf{T}\, X)$$

where B is called the type's *base functor*. Binary functor $\mathsf{B}(X, Y)$ is given this name because it is the basis of the whole inductive type definition. By instantiation

of X one obtains F. In the example above, $\mathsf{B}\,(X, Y) = 1 + X \times Y$ and in fact $\mathsf{F}\,Y = \mathsf{B}\,(\mathbb{N}_0, Y) = 1 + \mathbb{N}_0 \times Y$, recall (3.50). Moreover, one has

$$\mathsf{F}\,f \quad = \quad \mathsf{B}\,(id, f) \tag{3.75}$$

and so every F-homomorphism can be written in terms of the base-functor of F, *e.g.*

$$f \cdot \alpha = \beta \cdot \mathsf{B}\,(id, f)$$

instead of (3.65).

$\mathsf{T}\,X$ will be referred to as the *type functor* generated by B:

$$\underbrace{\mathsf{T}X}_{\text{type functor}} \quad \cong \quad \underbrace{\mathsf{B}(X, \mathsf{T}X)}_{\text{base functor}}$$

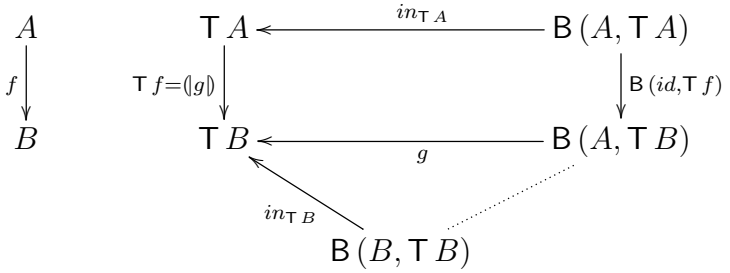
We proceed to the description of its functorial behaviour — $\mathsf{T}\,f$ — for a given $B \xleftarrow{f} A$. As far as typing rules are concerned, we shall have

$$\frac{B \xleftarrow{f} A}{\mathsf{T}\,B \xleftarrow{\mathsf{T}\,f} \mathsf{T}\,A}$$

So we should be able to express $\mathsf{T}\,f$ as a $\mathsf{B}\,(A, \_)$-catamorphism $(\!|g|\!)$:

$$\begin{array}{ccc} A & & \mathsf{T}\,A \xleftarrow{in_{\mathsf{T}\,A}} \mathsf{B}\,(A, \mathsf{T}\,A) \\ {\scriptstyle f}\downarrow & & {\scriptstyle \mathsf{T}\,f=(\!|g|\!)}\downarrow \qquad\qquad \downarrow{\scriptstyle \mathsf{B}\,(id, \mathsf{T}\,f)} \\ B & & \mathsf{T}\,B \xleftarrow[g]{} \mathsf{B}\,(A, \mathsf{T}\,B) \end{array}$$

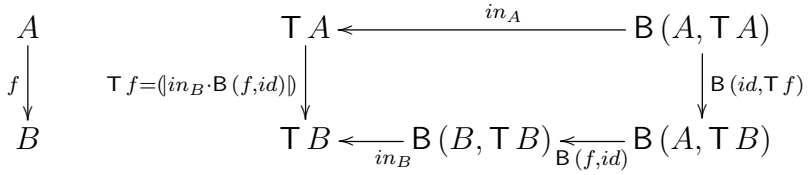
As we know that $in_{\mathsf{T}\,B}$ is the standard constructor of values of type $\mathsf{T}\,B$, we may put it into the diagram too:



The catamorphism's gene g will be synthesized by filling the dashed arrow in the diagram with the "obvious" $\mathsf{B}\,(f, id)$, whereby one gets

$$\mathsf{T}\,f \quad \stackrel{\text{def}}{=} \quad (\!|in_{\mathsf{T}\,B} \cdot \mathsf{B}\,(f, id)|\!) \tag{3.76}$$

and a final diagram, where $in_{\mathsf{T}\,A}$ is abbreviated by in_A (ibid. $in_{\mathsf{T}\,B}$ by in_B):



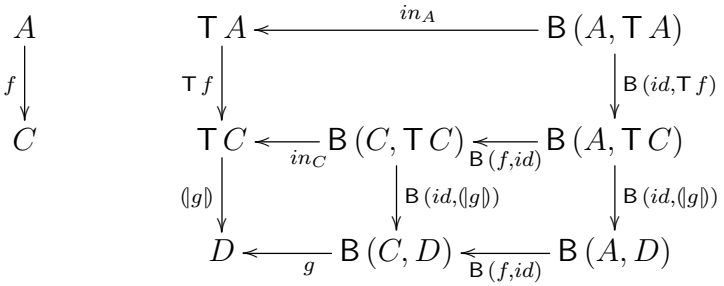
Next, we proceed to derive the useful law of *cata-absorption*

$$(\!|g|\!) \cdot \mathsf{T}\,f \quad = \quad (\!|g \cdot \mathsf{B}\,(f, id)|\!) \tag{3.77}$$

as consequence of the laws studied in section 3.12. Our target is to show that, for $k = (\!|g|\!) \cdot \mathsf{T}\,f$ in (3.67), one gets $\alpha = g \cdot \mathsf{B}\,(f, id)$:

$$\begin{array}{ll}
 & (\!|g|\!) \cdot \mathsf{T}\,f = (\!|\alpha|\!) \\
\Leftrightarrow & \quad \{ \text{ type-functor definition (3.76) } \} \\
 & (\!|g|\!) \cdot (\!|in_B \cdot \mathsf{B}\,(f, id)|\!) = (\!|\alpha|\!) \\
\Leftarrow & \quad \{ \text{ cata-fusion (3.71) } \} \\
 & (\!|g|\!) \cdot in_B \cdot \mathsf{B}\,(f, id) = \alpha \cdot \mathsf{B}\,(id, (\!|g|\!)) \\
\Leftrightarrow & \quad \{ \text{ cata-cancellation (3.69) } \} \\
 & g \cdot \mathsf{B}\,(id, (\!|g|\!)) \cdot \mathsf{B}\,(f, id) = \alpha \cdot \mathsf{B}\,(id, (\!|g|\!)) \\
\Leftrightarrow & \quad \{ \text{ B is a bi-functor (3.57) } \} \\
 & g \cdot \mathsf{B}\,(id \cdot f, (\!|g|\!) \cdot id) = \alpha \cdot \mathsf{B}\,(id, (\!|g|\!)) \\
\Leftrightarrow & \quad \{ \text{ } id \text{ is natural (2.11) } \} \\
 & g \cdot \mathsf{B}\,(f \cdot id, id \cdot (\!|g|\!)) = \alpha \cdot \mathsf{B}\,(id, (\!|g|\!)) \\
\Leftrightarrow & \quad \{ \text{ (3.57) again, this time from left to right } \} \\
 & g \cdot \mathsf{B}\,(f, id) \cdot \mathsf{B}\,(id, (\!|g|\!)) = \alpha \cdot \mathsf{B}\,(id, (\!|g|\!)) \\
\Leftarrow & \quad \{ \text{ Leibniz } \} \\
 & g \cdot \mathsf{B}\,(f, id) = \alpha
\end{array}$$

The following diagram pictures this property of catamorphisms:



It remains to show that (3.76) indeed defines a functor. This can be verified by checking properties (3.54) and (3.55) for $\mathsf{F} = \mathsf{T}$:

- Property **type-functor-id**, *cf.* (3.54):

$$
\begin{aligned}
& \mathsf{T}\, id \\
= & \quad \{ \text{ by definition (3.76) } \} \\
& (\!|in_B \cdot \mathsf{B}\,(id, id)|\!) \\
= & \quad \{ \text{ B is a bi-functor (3.56) } \} \\
& (\!|in_B \cdot id|\!) \\
= & \quad \{ \text{ identity and cata-reflection (3.68) } \} \\
& id
\end{aligned}
$$

- Property **type-functor**, *cf.* (3.55) :

$$
\begin{aligned}
& \mathsf{T}\,(f \cdot g) \\
= & \quad \{ \text{ by definition (3.76) } \} \\
& (\!|in_B \cdot \mathsf{B}\,(f \cdot g, id)|\!) \\
= & \quad \{ \ id \cdot id = id \text{ and B is a bi-functor (3.57) } \} \\
& (\!|in_B \cdot \mathsf{B}\,(f, id) \cdot \mathsf{B}\,(g, id)|\!) \\
= & \quad \{ \text{ cata-absorption (3.77) } \} \\
& (\!|in_B \cdot \mathsf{B}\,(f, id)|\!) \cdot \mathsf{T}\, g \\
= & \quad \{ \text{ again cata-absorption (3.77) } \}
\end{aligned}
$$

$$
\begin{aligned}
& (\!|in_B|\!) \cdot \mathsf{T}\, f \cdot \mathsf{T}\, g \\
= & \qquad \{ \text{ cata-reflection (3.68) followed by identity } \} \\
& \mathsf{T}\, f \cdot \mathsf{T}\, g
\end{aligned}
$$

Exercise 3.11. *Function* $\mathsf{length} = (\!|[\mathsf{zero}, \mathsf{succ} \cdot \pi_2]|\!)$ *counts the number of elements of a finite list. If the input list has at least one element it suffices to count the elements of its tail starting with count* 1 *instead of* 0*:*

$$\mathsf{length} \cdot (a{:}) = (\!|[\mathsf{one}, \mathsf{succ} \cdot \pi_2]|\!) \tag{3.78}$$

Prove (3.78) knowing that

$$\mathsf{length} \cdot (a{:}) = \mathsf{succ} \cdot \mathsf{length} \tag{3.79}$$

follows from the definition of length*. (**NB:** assume* $\mathsf{zero}\ \_ = 0$ *and* $\mathsf{one}\ \_ = 1$*.)*
□

Exercise 3.12. *Function* concat*, extracted from Haskell's* Prelude*, can be defined as list catamorphism,*

$$\mathsf{concat} = (\!|[\mathsf{nil}, \mathsf{conc}]|\!) \tag{3.80}$$

where $\mathsf{conc}\ (x, y) = x +\!\!+ y$*,* $\mathsf{nil}\ \_ = [\,]$*,* $\mathsf{B}\ (f, g) = id + f \times g$*,* $\mathsf{F}\ f = \mathsf{B}\ (id, f)$*, and* $\mathsf{T}\ f = \mathsf{map}\ f$*. Prove property*

$$\mathsf{length} \cdot \mathsf{concat} = \mathsf{sum} \cdot \mathsf{map}\ \mathsf{length} \tag{3.81}$$

resorting to cata-fusion and cata-absorption.
□

3.14 A catalogue of standard polynomial inductive types

The following table contains a collection of standard polynomial inductive types and associated base type bi-functors, which are in canonical form (3.63). The

table contains two extra columns which may be used as bookmarks for equations (3.75) and (3.76), respectively [12]:

Description	$\mathsf{T}\,X$	$\mathsf{B}\,(X,Y)$	$\mathsf{B}\,(id,f)$	$\mathsf{B}\,(f,id)$
"Right" Lists	$\mathsf{List}\,X$	$1+X\times Y$	$id+id\times f$	$id+f\times id$
"Left" Lists	$\mathsf{LList}\,X$	$1+Y\times X$	$id+f\times id$	$id+id\times f$
Non-empty Lists	$\mathsf{NList}\,X$	$X+X\times Y$	$id+id\times f$	$f+f\times id$
Binary Trees	$\mathsf{BTree}\,X$	$1+X\times Y^2$	$id+id\times f^2$	$id+f\times id$
"Leaf" Trees	$\mathsf{LTree}\,X$	$X+Y^2$	$id+f^2$	$f+id$

(3.82)

All type functors T in this table are unary. In general, one may think of inductive datatypes which exhibit more than one type parameter. Should n parameters be identified in T, then this will be based on an $n+1$-ary base functor B, *cf.*

$$\mathsf{T}(X_1,\ldots,X_n) \;\cong\; \mathsf{B}(X_1,\ldots,X_n,\mathsf{T}(X_1,\ldots,X_n))$$

So, every $n+1$-ary polynomial functor $\mathsf{B}(X_1,\ldots,X_n,X_{n+1})$ can be identified as the basis of an inductive n-ary type functor (the convention is to stick to the canonical form and reserve the last variable X_{n+1} for the "recursive call"). While type bi-functors ($n=2$) are often found in programming, the situation in which $n>2$ is relatively rare. For instance, the combination of leaf-trees with binary-trees in (3.82) leads to the so-called "full tree" type bi-functor

Description	$\mathsf{T}(X_1,X_2)$	$\mathsf{B}(X_1,X_2,Y)$	$\mathsf{B}(id,id,f)$	$\mathsf{B}(f,g,id)$
"Full" Trees	$\mathsf{FTree}(X_1,X_2)$	$X_1+X_2\times Y^2$	$id+id\times f^2$	$f+g\times id$

(3.83)

As we shall see later on, these types are widely used in programming. In the actual encoding of these types in HASKELL, exponentials are normally expanded to products according to (2.96), see for instance

$$\mathbf{data}\ \mathsf{BTree}\ a = \mathit{Empty} \mid \mathit{Node}\ (a,(\mathsf{BTree}\ a,\mathsf{BTree}\ a))$$

Moreover, one may chose to curry the type constructors as in, *e.g.*

$$\mathbf{data}\ \mathsf{BTree}\ a = \mathit{Empty} \mid \mathit{Node}\ a\ (\mathsf{BTree}\ a)\ (\mathsf{BTree}\ a)$$

Exercise 3.13. *Write as a catamorphisms*

[12] Since $(id_A)^2 = id_{(A^2)}$ one writes id^2 for id in this table.

- *the function which counts the number of elements of a non-empty list (type* NList *in (3.82)).*
- *the function which computes the maximum element of a binary-tree of natural numbers.*

□

Exercise 3.14. *Characterize the function which is defined by* $(\!|[\underline{[]}, h]|\!)$ *for each of the following definitions of* h*:*

$$h(x,(y_1,y_2)) = y_1 +\!\!+ [x] +\!\!+ y_2 \quad (3.84)$$

$$h = +\!\!+ \cdot (singl \times +\!\!+) \quad (3.85)$$

$$h = +\!\!+ \cdot (+\!\!+ \times singl) \cdot swap \quad (3.86)$$

assuming $singl\, a = [a]$*. Identify in (3.82) which datatypes are involved as base functors.*
□

Exercise 3.15. *Write as a catamorphism the function which computes the* frontier *of a tree of type* LTree *(3.82), listed from left to right.*
□

Exercise 3.16. *Function*

$$\mathsf{mirror}\ (Leaf\ a) = Leaf\ a$$
$$\mathsf{mirror}\ (Fork\ (x,y)) = Fork\ (\mathsf{mirror}\ y, \mathsf{mirror}\ x)$$

which mirrors binary trees of type $\mathsf{LTree}\ a = Leaf\ a \mid Fork\ (\mathsf{LTree}\ a, \mathsf{LTree}\ a)$ *can be defined both as a catamorphism*

$$\mathsf{mirror} = (\!|\mathsf{in} \cdot (id + \mathsf{swap})|\!) \quad (3.87)$$

and as an anamorphism

$$\mathsf{mirror} = [\!(\mathsf{(}id + \mathsf{swap}) \cdot \mathsf{out})]\!] \quad (3.88)$$

where out *is the converse of*

$$\mathsf{in} = [Leaf\,, Fork] \tag{3.89}$$

Show that both definitions are effectively the same, that is, complete the etc *steps of the rasoning:*

$$\begin{array}{ll} & \mathsf{mirror} = (\!|\mathsf{in} \cdot (id + \mathsf{swap})|\!) \\ \equiv & \quad \{\ ...\ etc\ ...\ \} \\ & \mathsf{mirror} = [\!(\,(id + \mathsf{swap}) \cdot \mathsf{out})]\!] \\ \Box & \end{array}$$

(***Hint***: *recall that* $\mathsf{F}\,f = id + f \times f$ *for this type and mind the natural property of* $id + \mathsf{swap}$.*)*
□

Exercise 3.17. *Let parametric type* T *be given with base* B*, that is, such that* $\mathsf{T}\ f = (\!|\ \mathbf{in} \cdot \mathsf{B}\ (f, id)|\!)$. *Define the so-called* triangular combinator *of* T*,* $tri\ f$*, as follows:*

$$tri\ f = (\!|\ \mathbf{in} \cdot \mathsf{B}\ (id, \mathsf{T}\ f)|\!) \tag{3.90}$$

Show that the instance of this combinator for type $\mathsf{LTree}\ a = Leaf\ a \mid Fork\ (\mathsf{LTree}\ a, \mathsf{LTree}\ a)$ *— such that* $\mathsf{in} = [Leaf\,, Fork]$ *and* $\mathsf{B}\ (f, g) = f + g \times g$ *— is the following function*

$$\begin{array}{l} tri :: (a \to a) \to \mathsf{LTree}\ a \to \mathsf{LTree}\ a \\ tri\ f\ (Leaf\ x) = Leaf\ x \\ tri\ f\ (Fork\ (t, t')) = Fork\ (fmap\ f\ (tri\ f\ t), fmap\ f\ (tri\ f\ t')) \end{array}$$

written in Haskell syntax.
□

3.15 Functors and type functors in HASKELL

The concept of a (unary) functor is provided in HASKELL in the form of a particular class exporting the `fmap` operator:

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}\\ \quad \mathsf{fmap}\ \cdot ::(a \to b) \to (f\ a \to f\ b)\end{array}$$

So `fmap g` encodes $\mathsf{F}\,g$ once we declare F as an instance of class `Functor`. The most popular use of `fmap` has to do with HASKELL lists, as allowed by declaration

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where}\\ \quad \mathsf{fmap}\ f\ [\,] = [\,]\\ \quad \mathsf{fmap}\ f\ (x : xs) = f\ x : \mathsf{fmap}\ f\ xs\end{array}$$

in language's *Standard Prelude*.

In order to encode the type functors we have seen so far we have to do the same concerning their declaration. For instance, should we write

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor}\ \mathsf{BTree}\\ \quad \textbf{where}\ \mathsf{fmap}\ f = \mathit{cataBTree}\ (\mathit{inBTree} \cdot (\mathit{id} + (f \times \mathit{id})))\end{array}$$

concerning the binary-tree datatype of (3.82) and assuming appropriate declarations of `cataBTree` and `inBTree`, then `fmap` is overloaded and used across such binary-trees.

Bi-functors can be added easily by writing

$$\begin{array}{l}\textbf{class}\ \mathit{BiFunctor}\ f\ \textbf{where}\\ \quad \mathit{bmap} :: (a \to b) \to (c \to d) \to (f\ a\ c \to f\ b\ d)\end{array}$$

Exercise 3.18. *Declare all datatypes in (3.82) in* HASKELL *notation and turn them into* HASKELL *type functors, that is, define* `fmap` *in each case.*
□

Exercise 3.19. *Declare datatype (3.83) in* HASKELL *notation and turn it into an instance of class* `BiFunctor`.
□

3.16 The mutual-recursion law

The theory developed so far for building (and reasoning about) recursive functions doesn't cope with mutual recursion. As a matter of fact, the pattern of recursion of a given cata(ana,hylo)morphism involves only the recursive function being defined, even though more than once, in general, as dictated by the relevant base functor.

It turns out that rules for handling mutual recursion are surprisingly simple to calculate. As motivation, recall section 2.10 where, by mixing products with coproducts, we obtained a result — the *exchange rule* (2.47) — which stemmed from putting together the two universal properties of product and coproduct, (2.55) and (2.57), respectively.

The question we want to address in this section is of the same brand: *what can one tell about catamorphisms which output pairs of values*? By (2.55), such catamorphisms are bound to be *splits*, as are the corresponding *genes*:

$$\begin{array}{ccc} \mu_{\mathsf{F}} & \xleftarrow{in} & \mathsf{F}\,\mu_{\mathsf{F}} \\ {\scriptstyle (\!|\langle h,k\rangle|\!)}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,(\!|\langle h,k\rangle|\!)} \\ A\times B & \xleftarrow[\langle h,k\rangle]{} & \mathsf{F}\,(A\times B) \end{array}$$

As we did for the exchange rule, we put (2.55) and the universal property of catamorphisms (3.67) against each other and calculate:

$$\begin{array}{ll} & \langle f,g\rangle = (\!|\langle h,k\rangle|\!) \\ \equiv & \quad \{\ \text{cata-universal (3.67)}\ \} \\ & \langle f,g\rangle \cdot in = \langle h,k\rangle \cdot \mathsf{F}\,\langle f,g\rangle \\ \equiv & \quad \{\ \times\text{-fusion (2.24) twice}\ \} \\ & \langle f\cdot in, g\cdot in\rangle = \langle h\cdot \mathsf{F}\,\langle f,g\rangle, k\cdot \mathsf{F}\,\langle f,g\rangle\rangle \\ \equiv & \quad \{\ (2.56)\ \} \\ & f\cdot in = h\cdot \mathsf{F}\,\langle f,g\rangle \quad \wedge \quad g\cdot in = k\cdot \mathsf{F}\,\langle f,g\rangle \end{array}$$

The rule thus obtained,

$$\left\{\begin{array}{l} f\cdot in = h\cdot \mathsf{F}\,\langle f,g\rangle \\ g\cdot in = k\cdot \mathsf{F}\,\langle f,g\rangle \end{array}\right. \quad \equiv \quad \langle f,g\rangle = (\!|\langle h,k\rangle|\!) \tag{3.91}$$

is referred to as the *mutual recursion law* (or as "Fokkinga's law") and is useful in combining two mutually recursive functions f and g

$$\begin{array}{ccc} \mu_{\mathsf{F}} & \xleftarrow{in} & \mathsf{F}\,\mu_{\mathsf{F}} \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,\langle f,g\rangle} \\ A & \xleftarrow[h]{} & \mathsf{F}\,(A\times B) \end{array} \qquad \begin{array}{ccc} \mu_{\mathsf{F}} & \xleftarrow{in} & \mathsf{F}\,\mu_{\mathsf{F}} \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,\langle f,g\rangle} \\ B & \xleftarrow[k]{} & \mathsf{F}\,(A\times B) \end{array}$$

into a single catamorphism.

When applied from left to right, law (3.91) is surprisingly useful in optimizing recursive functions in a way which saves redundant traversals of the input inductive type μ_{F}. Let us take the Fibonacci function as example:

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

It can be shown that fib is a hylomorphism of type LTree (3.82), $fib = [\![count, fibd]\!]$, for $count = [\underline{1}, add]$, $add(x,y) = x+y$ and $fibd\,n = if\ n<2\ then\ i_1 Nil\ else\ i_2(n-1, n-2)$. This hylo-factorization of fib tells its internal algorithmic structure: the *divide step* $[\!(fibd)\!]$ builds a tree whose number of leaves is a Fibonacci number; the *conquer step* $(\!|count|\!)$ just counts such leaves.

There is, of course, much re-calculation in this hylomorphism. Can we improve its performance? The clue is to regard the two instances of fib in the recursive branch as mutually recursive over the natural numbers. This clue is suggested not only by fib having two base cases (so, perhaps it hides two functions) but also by the lookahead $n+2$ in the recursive clause.

We start by defining a function which reduces such a lookahead by 1,

$$f\ n\ =\ fib(n+1)$$

Clearly, $f(n+1) = fib(n+2) = f\ n + fib\ n$ and $f\ 0 = fib\ 1 = 1$. Putting f and fib together,

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

we obtain two mutually recursive functions over the natural numbers ($\mathbb{N}_0$) which transform into pointfree equalities

$$\begin{aligned} f \cdot [\underline{0}, suc] &= [\underline{1}, add \cdot \langle f, fib \rangle] \\ fib \cdot [\underline{0}, suc] &= [\underline{1}, f] \end{aligned}$$

over

$$\mathbb{N}_0 \underset{\mathbf{in}=[\underline{0},suc]}{\overset{\cong}{\rightleftarrows}} \underbrace{1 + \mathbb{N}_0}_{\mathsf{F}\,\mathbb{N}_0} \tag{3.92}$$

Reverse $+$-absorption (2.41) will further enable us to rewrite the above into

$$\begin{aligned} f \cdot in &= [\underline{1}, add] \cdot \mathsf{F}\,\langle f, fib \rangle \\ fib \cdot in &= [\underline{1}, \pi_1] \cdot \mathsf{F}\,\langle f, fib \rangle \end{aligned}$$

thus bringing functor $\mathsf{F}\,f = id + f$ explicit and preparing for mutual recursion removal:

$$\begin{aligned} & f \cdot in = [\underline{1}, add] \cdot \mathsf{F}\,\langle f, fib \rangle \\ & fib \cdot in = [\underline{1}, \pi_1] \cdot \mathsf{F}\,\langle f, fib \rangle \\ \equiv & \quad \{\ (3.91)\ \} \\ & \langle f, fib \rangle = (\!|\langle [\underline{1}, add], [\underline{1}, \pi_1] \rangle|\!) \\ \equiv & \quad \{\ \text{exchange law (2.47)}\ \} \\ & \langle f, fib \rangle = (\!|[\langle \underline{1}, \underline{1} \rangle, \langle add, \pi_1 \rangle]|\!) \\ \equiv & \quad \{\ \text{going pointwise and denoting } \langle f, fib \rangle \text{ by } fib'\ \} \\ & \begin{cases} fib'\ 0 = (1, 1) \\ fib'\ (n+1) = (x+y, x)\ where\ (x, y) = fib'\ n \end{cases} \end{aligned}$$

Since $fib = \pi_2 \cdot fib'$ we easily recover fib from fib' and obtain the intended linear version of Fibonacci, below encoded in Haskell:

```
fib n = m where (_,m) = fib' n
                fib' 0 = (1,1)
                fib' (n+1) = (x+y,x)
                             where (x,y) = fib' n
```

This version of fib is actually the semantics of the "for-loop" — recall (3.7) — one would write in an imperative language which would initialize two global variables $x, y := 1, 1$, loop over assignment $x, y := x + y, x$ and yield the result in y. In the C programming language, one would write

```
int fib(int n)
{
int x=1; int y=1; int i;
for (i=1;i<=n;i++) {int a=x; x=x+y; y=a;}
return y;
};
```

where the extra variable a is required for ensuring that *simultaneous* assignment $x, y := x + y, x$ takes place in a sequential way.

Recall from section 3.1 that all $\mathbb{N}_0$ catamorphisms are of shape $(\!|[\underline{k}, g]|\!)$ and such that $(\!|[\underline{k}, g]|\!)n = g^n k$, where g^n is the n-th iteration of g, that is, $g^0 = id$ and $g^{n+1} = g \cdot g^n$. That is, g is the body of a "for-loop" which repeats itself n-times, starting with initial value k. Recall also that the for-loop combinator is nothing but the "fold combinator" (3.5) associated to the natural number data type.

In a sense, the mutual recursion law gives us a hint on how global variables "are born" in computer programs, out of the maths definitions themselves. Quite often more that two such variables are required in linearizing hylomorphisms by mutual recursion. Let us see an example. The question is: *how many squares can one draw on a $n \times n$-tiled wall?* The answer is given by function

$$ns\ n \stackrel{\text{def}}{=} \sum_{i=1,n} i^2$$

that is,

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

in Haskell. However, this hylomorphism is inefficient because each iteration involves another hylomorphism computing square numbers.

One way of improving ns is to introduce function $bnm\ n \stackrel{\text{def}}{=} (n+1)^2$ and express this over (3.92),

$$\begin{aligned} bnm\ 0 &= 1 \\ bnm(n+1) &= 2n + 3 + bnm\ n \end{aligned}$$

hoping to blend ns with bnm using the mutual recursion law. However, the same problem arises in bnm itself, which now depends on term $2n+3$. We invent $lin\ n \stackrel{\text{def}}{=} 2n+3$ and repeat the process, thus obtaining:

$$\begin{aligned} lin\ 0 &= 3 \\ lin(n+1) &= 2 + lin\ n \end{aligned}$$

By redefining

$$\begin{aligned} bnm'\ 0 &= 1 \\ bnm'(n+1) &= lin\ n + bnm'\ n \end{aligned}$$

$$\begin{aligned} ns'\ 0 &= 0 \\ ns'(n+1) &= bnm'\ n + ns'\ n \end{aligned}$$

we obtain three functions — ns', bnm' and lin — mutually recursive over the polynomial base $\mathsf{F}\ g = id + g$ of the natural numbers.

Exercise 3.22 below shows how to extend (3.91) to three mutually recursive functions (3.93). (From this it is easy to extend it further to the n-ary case.) It is routine work to show that, by application of (3.93) to the above three functions, one obtains the linear version of ns which follows:

$$\begin{aligned} &ns''\ n = a\ \mathbf{where} \\ &\quad (a, \_, \_) = aux\ n \\ &\quad aux\ 0 = (0, 1, 3) \\ &\quad aux\ (n+1) = \mathbf{let}\ (a, b, c) = aux\ n\ \mathbf{in}\ (a+b, b+c, 2+c) \end{aligned}$$

In retrospect, note that (in general) not every system of n mutually recursive functions

$$\begin{cases} f_1 = \phi_1(f_1, \ldots, f_n) \\ \vdots \\ f_n = \phi_n(f_1, \ldots, f_n) \end{cases}$$

involving n functions and n functional combinators $\phi_1, \ldots, \phi_n$ can be handled by a suitably extended version of (3.91). This only happens if all f_i have the same "shape", that is, if they share the same base functor F.

Exercise 3.20. *Use the mutual recursion law (3.91) to show that each of the two functions*

$$\begin{cases} odd\ 0 = False \\ odd(n+1) = even\ n \end{cases} \qquad \begin{cases} even\ 0 = True \\ even(n+1) = odd\ n \end{cases}$$

checking natural number parity can be expressed as a projection of

$$\mathsf{for}\ swap\ (False, True)$$

Encode this for-loop in C syntax.
□

Exercise 3.21. *The following Haskell function computes the list of the first n natural numbers in reverse order:*

$$\begin{aligned} &insg\ 0 = [\,] \\ &insg\ (n+1) = (n+1) : insg\ n \end{aligned}$$

1. *Show that $insg$ can also be defined as follows:*

$$\begin{aligned} &insg\ 0 = [\,] \\ &insg\ (n+1) = (fsuc\ n) : insg\ n \\ &fsuc\ 0 = 1 \\ &fsuc\ (n+1) = fsuc\ n + 1 \end{aligned}$$

2. *Based on the mutual recursion law derive from such a definition the following version of $insg$ encoded as a for-loop:*

$$\begin{aligned} &insg = \pi_2 \cdot insgfor \\ &insgfor = \mathsf{for}\ \langle (1+) \cdot \pi_1, \mathsf{cons} \rangle\ \underline{(1, [\,])} \end{aligned}$$

where $cons(n, m) = n : m$.

□

Exercise 3.22. *Show that law (3.91) generalizes to more than two mutually recursive functions, in this case three:*

$$\left\{\begin{array}{l} f \cdot in = h \cdot \mathsf{F}\,\langle f, \langle g, j\rangle\rangle \\ g \cdot in = k \cdot \mathsf{F}\,\langle f, \langle g, j\rangle\rangle \\ j \cdot in = l \cdot \mathsf{F}\,\langle f, \langle g, j\rangle\rangle \end{array}\right. \quad \equiv \quad \langle f, \langle g, j\rangle\rangle = (\!|\langle h, \langle k, l\rangle\rangle|\!) \tag{3.93}$$

□

Exercise 3.23. *The exponential function $e^x : \mathbb{R} \to \mathbb{R}$ (where "e" denotes Euler's number) can be defined in several ways, one being the calculation of Taylor series:*

$$e^x \quad = \quad \sum_{n=0}^{\infty} \frac{x^n}{n!} \tag{3.94}$$

The following function, in Haskell,

$$\begin{array}{l} \mathsf{exp} :: Double \to Integer \to Double \\ \mathsf{exp}\ x\ 0 = 1 \\ \mathsf{exp}\ x\ (n+1) = x \uparrow (n+1)\ /\ fac\ (n+1) + (\mathsf{exp}\ x\ n) \end{array}$$

computes an approximation of e^x, where the second parameter tells how many terms to compute. For instance, while $exp\ 1\ 1 = 2.0$, $exp\ 1\ 10$ yields 2.7182818011463845.

Function $exp\ x\ n$ performs badly for n larger and larger: while $exp\ 1\ 100$ runs instantaneously, $exp\ 1\ 1000$ takes around 9 seconds, $exp\ 1\ 2000$ takes circa 33 seconds, and so on.

Decompose exp into mutually recursive functions so as to apply (3.93) and obtain the following linear version:

$$\begin{array}{l} \mathsf{exp}\ x\ n = \mathbf{let}\ (e, b, c) = aux\ x\ n \\ \quad \mathbf{in}\ e\ \mathbf{where} \\ \quad\quad aux\ x\ 0 = (1, 2, x) \\ \quad\quad aux\ x\ (n+1) = \mathbf{let}\ (e, s, h) = aux\ x\ n \\ \quad\quad\quad \mathbf{in}\ (e + h, s + 1, (x\ /\ s) * h) \end{array}$$

□

Exercise 3.24. *Show that, for all $n \in \mathbb{N}_0$, $n = suc^n 0$. **Hint:** use cata-reflexion (3.68).* □

Mutual recursion over lists. As example of application of (3.91) for μ_{F} other than $\mathbb{N}_0$, consider the following recursive predicate which checks whether a (non-empty) list is ordered,

$$\begin{aligned} &ord : A^+ \to 2 \\ &ord\,[a] = \text{TRUE} \\ &ord\,(cons(a,l)) = a \geqslant (listMax\,l) \wedge (ord\,l) \end{aligned}$$

where $\geqslant$ is assumed to be a total order on datatype A and

$$listMax = (\!|[id, max]|\!) \tag{3.95}$$

computes the greatest element of a given list of As:

$$\begin{array}{ccc} A^+ & \xleftarrow{[singl,cons]} & A + A \times A^+ \\ {\scriptstyle listMax}\downarrow & & \downarrow{\scriptstyle id+id\times listMax} \\ A & \xleftarrow[[id,max]]{} & A + A \times A \end{array}$$

(In the diagram, $singl\,a = [a]$.)

Predicate ord is not a catamorphism because of the presence of $listMax\ l$ in the recursive branch. However, the following diagram depicting ord

$$\begin{array}{ccc} A^+ & \xleftarrow{[singl,cons]} & A + A \times A^+ \\ {\scriptstyle ord}\downarrow & & \downarrow{\scriptstyle id+id\times\langle listMax,ord\rangle} \\ 2 & \xleftarrow[[\underline{\text{TRUE}},\alpha]]{} & A + A \times (A \times 2) \end{array}$$

(where $\alpha(a,(m,b)) \stackrel{\text{def}}{=} a \geqslant m \wedge b$) suggests the possibility of using the mutual recursion law. One only has to find a way of letting $listMax$ depend also on ord, which isn't difficult: for any $A^+ \xrightarrow{g} B$, one has

$$\begin{array}{ccc} A^+ & \xleftarrow{[singl,cons]} & A + A \times A^+ \\ {\scriptstyle listMax}\downarrow & & \downarrow{\scriptstyle id+id\times\langle listMax,g\rangle} \\ A & \xleftarrow[[id,max\cdot(id\times\pi_1)]]{} & A + A \times (A \times B) \end{array}$$

where the extra presence of g is cancelled by projection π_1.

For $B = 2$ and $g = ord$ we are in position to apply Fokkinga's law and obtain:

$$\begin{aligned}\langle listMax, ord\rangle &= (\!|\langle [id, max \cdot (id \times \pi_1)], [\underline{\text{TRUE}}, \alpha]\rangle|\!) \\ &= \quad \{\ \textit{exchange law}\ (2.47)\ \} \\ &\quad (\!|[\langle id, \underline{\text{TRUE}}\rangle, \langle max \cdot (id \times \pi_1), \alpha\rangle]|\!)\end{aligned}$$

Of course, $ord = \pi_2 \cdot \langle listMax, ord\rangle$. By denoting the above synthesized catamorphism by aux, we end up with the following version of ord:

$$\begin{array}{lcll} ord\, l & = & let & (a, b) = aux\, l \\ & & in & b \end{array}$$

where

$$\begin{array}{l} aux : A^+ \to A \times 2 \\ aux\, [a] = (a, \text{TRUE}) \\ aux\, (cons(a, l)) = \begin{array}{ll} let & (m, b) = aux\, l \\ in & (max(a, m), (a > m \wedge b)) \end{array} \end{array}$$

Exercise 3.25. *What do the following Haskell functions do?*

$$\begin{array}{l} f_1\ [\,] = [\,] \\ f_1\ (h : t) = h : (f_2\ t) \\ f_2\ [\,] = [\,] \\ f_2\ (h : t) = f_1\ t \end{array}$$

Write $f = \langle f_1, f_2\rangle$ as a list catamorphism and encode f back into Haskell syntax.
□

3.17 "Banana-split": a corollary of the mutual-recursion law

Let $h = i \cdot \mathsf{F}\,\pi_1$ and $k = j \cdot \mathsf{F}\,\pi_2$ in (3.91). Then

$$f \cdot in = (i \cdot \mathsf{F}\,\pi_1) \cdot \mathsf{F}\,\langle f, g\rangle$$

$$
\begin{array}{ll}
\equiv & \{ \text{ composition is associative and } \mathsf{F} \text{ is a functor } \} \\
& f \cdot in = i \cdot \mathsf{F}\,(\pi_1 \cdot \langle f, g \rangle) \\
\equiv & \{ \text{ by } \times\text{-cancellation (2.20) } \} \\
& f \cdot in = i \cdot \mathsf{F}\, f \\
\equiv & \{ \text{ by cata-cancellation } \} \\
& f = (\!|i|\!)
\end{array}
$$

Similarly, from $k = j \cdot \mathsf{F}\,\pi_2$ we get

$$g = (\!|j|\!)$$

Then, from (3.91), we get

$$\langle (\!|i|\!), (\!|j|\!) \rangle = (\!|\langle i \cdot \mathsf{F}\,\pi_1, j \cdot \mathsf{F}\,\pi_2 \rangle|\!)$$

that is

$$\langle (\!|i|\!), (\!|j|\!) \rangle = (\!|(i \times j) \cdot \langle \mathsf{F}\,\pi_1, \mathsf{F}\,\pi_2 \rangle|\!) \tag{3.96}$$

by (reverse) $\times$-absorption (2.25).

This law provides us with a very useful tool for "parallel loop" inter-combination: "loops" $(\!|i|\!)$ and $(\!|j|\!)$ are fused together into a single "loop" $(\!|(i \times j) \cdot \langle \mathsf{F}\,\pi_1, \mathsf{F}\,\pi_2 \rangle|\!)$. The need for this kind of calculation arises very often. Consider, for instance, the function which computes the average of a non-empty list of natural numbers,

$$average \stackrel{\text{def}}{=} (/) \cdot \langle sum, length \rangle \tag{3.97}$$

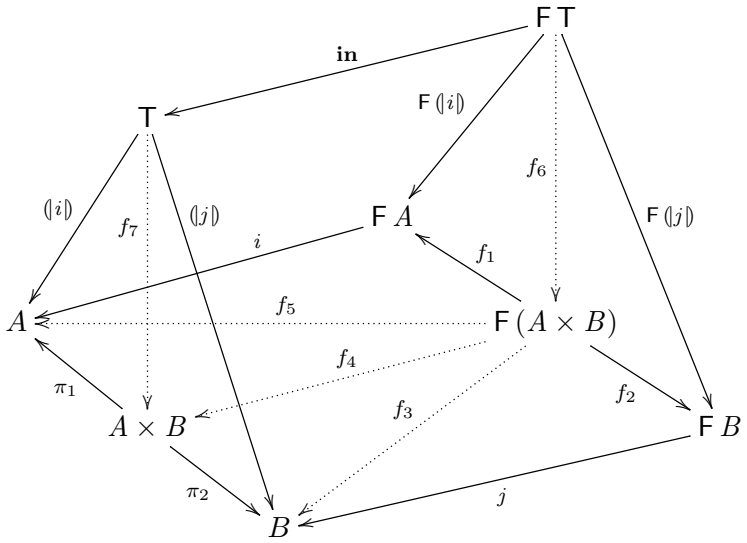
where sum and $length$ are the expected $\mathbb{N}^+$ catamorphisms:

$$
\begin{aligned}
sum &= (\!|[id, +]|\!) \\
length &= (\!|[\underline{1}, succ \cdot \pi_2]|\!)
\end{aligned}
$$

As defined by (3.97), function $average$ performs two independent traversals of the argument list before division $(/)$ takes place. Banana-split will fuse such two traversals into a single one (see function aux below), thus leading to a function which will run "twice as fast":

$$
\begin{array}{lll}
average\, l & = & x/y \\
& & where \quad (x, y) = aux\ l \\
& & \qquad\quad aux[a] = (a, 1) \\
& & \qquad\quad aux(cons(a, l)) = \ (a + x, y + 1) \\
& & \qquad\qquad\qquad\quad where \quad (x, y) = aux\ l
\end{array}
\tag{3.98}
$$

Exercise 3.26. *The following diagram depicts "banana-split" (3.96):*



Identify all functions f_1 to f_7.

□

Exercise 3.27. *Calculate (3.98) from (3.97). Which of these two versions of the same function is easier to understand?*

□

Exercise 3.28. *Show that the standard Haskell function*

$$\mathsf{unzip}\ xs = (\mathsf{map}\ \pi_1\ xs, \mathsf{map}\ \pi_2\ xs)$$

*can be defined as a catamorphism (*fold*) thanks to (3.96). Generalize this calculation to the* generic *unzip function over an inductive (polynomial) type* T*:*

$$unzip_{\mathsf{T}} = \langle \mathsf{T}\pi_1, \mathsf{T}\pi_2 \rangle$$

Suggestion: recall (3.76).

□

3.18 Inductive datatype isomorphism

not yet available

3.19 Bibliography notes

It is often the case that the expressive power of a particular programming language or paradigm is counter-productive in the sense that too much freedom is given to programmers. Sooner or later, these will end up writing unintelligible (authorship dependent) code which will become a burden to whom has to maintain it. Such has been the case of imperative programming in the past (inc. assembly code), where the unrestricted use of `goto` instructions eventually gave place to `if-then-else`, `while` and `repeat` *structured* programming constructs.

A similar trend has been observed over the last decades at a higher programming level: arbitrary recursion and/or (side) effects have been considered harmful in functional programming. Instead, programmers have been invited to structure their code around generic program devices such as eg. *fold/unfold* combinators, which bring discipline to recursion. One witnesses progress in the sense that the loss of freedom is balanced by the increase of formal semantics and the availability of program calculi.

Such disciplined programming combinators have been extended from list-processing to other inductive structures thanks to one of the most significant advances in programming theory over the last decade: the so-called *functorial* approach to datatypes which originated mainly from [26], was popularized by [25] and reached textbook format in [6]. A comfortable basis for exploiting *polymorphism* [37], the "datatypes as functors" moto has proved beneficial at a higher level of abstraction, giving birth to *polytypism* [21].

The literature on *anas*, *catas* and *hylos* is vast (see eg. [28], [20], [13]) and it is part of a broader discipline which has become known as the *mathematics of program construction* [2]. This chapter provides an introduction to such as discipline. Only the calculus of catamorphisms is presented. The corresponding theory of anamorphisms and hylomorphisms demands further mathematical machinery (functions generalized to binary relations) and won't be dealt with before chapters 10 and 9. The results on mutual recursion presented in this chapter, pionered by Maarten Fokkinga [10], have been extended towards probabilistic functions [30]. They have also shown to help in program understanding and reverse engineering [36]. Recently, the whole theory has undergone significant advances throught fur-

ther use of category theory notions such as adjunctions [13] and conjugate functors [15, 16].

[13] See chapter 4.

Chapter 4

Why Monads Matter

In this chapter we present a powerful device in state-of-the-art functional programming, that of a *monad*. The monad concept is nowadays of primary importance in computing science because it makes it possible to describe computational effects as disparate as input/output, comprehension notation, state variable updating, probabilistic behaviour, context dependence, partial behaviour *etc.* in an elegant and uniform way.

Our motivation to this concept will start from a well-known problem in functional programming (and computing as a whole) — that of coping with undefined computations.

4.1 Partial functions

Recall the function head which (respectively) yields the first element of a finite list. Clearly, head x is undefined for $x = [\,]$ because the empty list has no elements at all. As expected, the HASKELL output for head $[\,]$ is just "panic":

```
*Main> head []
*** Exception: Prelude.head: empty list
*Main>
```

Functions such as head are called *partial functions* because they cannot be applied to all of their (well-typed) inputs, *i.e.*, they diverge for some of such inputs. Partial functions are very common in mathematics or programming — for other examples think of *e.g.* tail, and so on.

Panic is very dangerous in programming. In order to avoid this kind of behaviour one has two alternatives, either (a) ensuring that every call to head x

is *protected* — *i.e.*, the contexts which wrap up such calls ensure *pre-condition* $x \neq [\,]$, or (b) *raising* exceptions, *i.e.* explicit error values, as above. In the former case, mathematical proofs need to be carried out in order to ensure *safety* (that is, *pre-condition* compliance). The overall effect is that of *restricting* the domain of the partial function. In the latter case one goes the other way round, by extending the co-domain (vulg. range) of the function so that it accommodates exceptional outputs. In this way one might define, in HASKELL:

$$\mathbf{data}\ ExtVal\ a = Ok\ a \mid Error$$

and then define the "extended" version of head:

$$\begin{aligned} &exthead :: [a] \to ExtVal\ a \\ &exthead\ [\,] = Error \\ &exthead\ x = Ok\ (\mathsf{head}\ x) \end{aligned}$$

Note that $ExtVal$ is a *parametric* type which extends an arbitrary data type a with its (polymorphic) exception (or error value). It turns out that, in HASKELL, $ExtVal$ is redundant because such a parametric type already exists and is called $Maybe$:

$$\mathbf{data}\ Maybe\ a = \mathsf{Nothing} \mid \mathsf{Just}\ a$$

Clearly, the isomorphisms hold:

$$\mathsf{ExtVal}\ A \cong \mathsf{Maybe}\ A \cong 1 + A$$

So, in abstract terms, one may regard as *partial* every function of type

$$1 + A \xleftarrow{g} B$$

for some A and B [1].

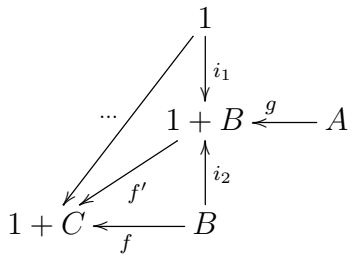
4.2 Putting partial functions together

Do partial functions compose? Their types won't match in general:

$$\begin{array}{ccc} & & 1 + B \xleftarrow{g} A \\ & & \vdots \\ 1 + C & \xleftarrow{f} & B \end{array}$$

[1] In conventional programming, every function delivering a *pointer* as result — as in *e.g.* the C programming language — can be regarded as one of these functions.

Clearly, we have to extend f — which is itself a partial function — to some f' able to accept arguments from $1 + B$:



The most "obvious" instance of the ellipsis (...) in the diagram above is i_1 and this corresponds to what is called *strict* composition: an exception produced by the *producer* function g is propagated to the output of the *consumer* function f. We define:

$$f \bullet g \quad \stackrel{\text{def}}{=} \quad [i_1 , f] \cdot g \tag{4.1}$$

Expressed in terms of $Maybe$, composite function $f \bullet g$ works as follows:

$$(f \bullet g)a \quad = \quad f'(g\, a)$$

where

$$f'\ \mathsf{Nothing} = \mathsf{Nothing}$$
$$f'\ (\mathsf{Just}\ b) = f\ b$$

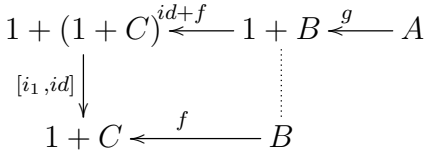
Altogether, we have the following Haskell pointwise expression for $f \bullet g$:

$$\lambda a \rightarrow f'\ (g\ a)\ \mathbf{where}$$
$$\quad f'\ \mathsf{Nothing} = \mathsf{Nothing}$$
$$\quad f'\ (\mathsf{Just}\ b) = f\ b$$

Note that the adopted extension of f can be decomposed — by reverse $+$-absorption (2.41) — into

$$f' \quad = \quad [i_1 , id] \cdot (id + f)$$

as displayed in diagram



All in all, we have the following version of (4.1):

$$f \bullet g \stackrel{\text{def}}{=} [i_1, id] \cdot (id + f) \cdot g$$

Does this functional composition scheme have a unit, that is, is there a function u such that

$$f \bullet u = f = u \bullet f \tag{4.2}$$

holds? Clearly, if it exists, it must bear type $1 + A \xleftarrow{u} A$. Let us *solve* (4.2) for u:

$$\begin{array}{ll} & f \bullet u = f = u \bullet f \\ \equiv & \quad \{ \text{ substitution } \} \\ & [i_1, f] \cdot u = f = [i_1, u] \cdot f \\ \Leftarrow & \quad \{ \text{ let } u = i_2 \} \\ & [i_1, f] \cdot i_2 = f = [i_1, i_2] \cdot f \wedge u = i_2 \\ \equiv & \quad \{ \text{ by } + \text{-cancellation (2.38) and } + \text{-reflection (2.39) } \} \\ & f = f = id \cdot f \wedge u = i_2 \\ \Leftarrow & \quad \{ \text{ identity } \} \\ & u = i_2 \end{array}$$

So $f \bullet u = f = u \bullet f$ for $u = i_2$.

Exercise 4.1. *Prove that property*

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

holds, for $f \bullet g$ defined by (4.1).
□

4.3 Lists

In contrast to partial functions, which may produce *no* output, let us now consider functions which may deliver *too many* outputs, for instance, lists of output values:

$$\begin{array}{ccc} & & B^\star \\ & \overset{g}{\nearrow} & \vdots \\ A & & \\ & \underset{f}{\searrow} & \\ & & C^\star \end{array} \qquad \begin{array}{ccc} & & B^\star \\ & \overset{g}{\nearrow} & \vdots \\ A & & \\ & \underset{f}{\searrow} & \\ & & C^\star \end{array}$$

Functions f and g do not compose but, once again, one can think of extending the consumer function (f) by mapping it along the output of the producer function (g):

$$\begin{array}{ccc} (C^\star)^\star & \overset{f^\star}{\longleftarrow} & B^\star \\ \vdots & & \vdots \\ C^\star & \underset{f}{\longleftarrow} & B \end{array}$$

To complete the process, one has to *flatten* the nested-sequence output in $(C^\star)^\star$ via the obvious list-catamorphism $C^\star \xleftarrow{concat} (C^\star)^\star$, where $concat \stackrel{\text{def}}{=} (\!|[[\,]\,, +\!\!+]|\!)$. In summary:

$$f \bullet g \stackrel{\text{def}}{=} concat \cdot f^\star \cdot g \tag{4.3}$$

as captured in the following diagram:

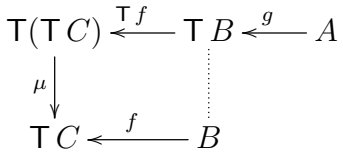
$$\begin{array}{ccccc} (C^\star)^\star & \overset{f^\star}{\longleftarrow} & B^\star & \overset{g}{\longleftarrow} & A \\ {\scriptstyle concat}\downarrow & & \vdots & & \\ C^\star & \underset{f}{\longleftarrow} & B & & \end{array}$$

Exercise 4.2. *Show that* $singl$ *(recall exercise 3.14) is the unit* u *of* $\bullet$ *as defined by (4.3).*
□

Exercise 4.3. *Encode in* HASKELL *a pointwise version of (4.3).* ***Hint****: start by applying (list) cata-absorption (3.77).*
□

4.4 Monads

Both function composition schemes (4.1) and (4.3) above share the same polytypic pattern: the output of the producer function g is "T-times" more elaborate than the input of the consumer function f, where T is some parametric datatype: $\mathsf{T}\,X = 1+X$ in case of (4.1), and $\mathsf{T}\,X = X^\star$ in case of (4.3). Then a composition scheme is devised for such functions, which is displayed in



and is given by

$$f \bullet g \stackrel{\text{def}}{=} \mu \cdot \mathsf{T}\, f \cdot g \tag{4.4}$$

where $\mathsf{T}\,A \xleftarrow{\mu} \mathsf{T}^2\,A$ is a suitable polymorphic function. (We have already seen $\mu = [i_1, id]$ in case (4.1), and $\mu = concat$ in case (4.3).)

Together with a unit function $\mathsf{T}\,A \xleftarrow{u} A$ and μ, that is

$$A \xrightarrow{u} \mathsf{T}\ A \xleftarrow{\mu} \mathsf{T}^2\ A$$

datatype T will form a so-called *monad* type, of which $(1+)$ and $(\_)^\star$ are the two examples seen above. Arrow $\mu \cdot \mathsf{T}\, f$ is called the *extension* of f. Functions μ and u are referred to as the monad's *multiplication* and *unit*, respectively. The monadic composition scheme (4.4) is referred to as *Kleisli composition*.

A *monadic arrow* $\mathsf{T}B \xleftarrow{f} A$ conveys the idea of a function which produces an output of "type" B "wrapped by T", where datatype T describes some kind of (computational) "effect". The monad's unit $\mathsf{T}B \xleftarrow{u} B$ is a primitive monadic arrow which injects (*i.e.* promotes, wraps) data *inside* such an effect.

The monad concept is nowadays of primary importance in computing science because it makes it possible to describe computational effects as disparate as input/output, state variable updating, context dependence, partial behaviour (seen above) *etc.* in an elegant, generic and uniform way. Moreover, the monad's operators exhibit notable properties which make it possible to *reason* about such computational effects.

The remainder of this section is devoted to such properties. First of all, the properties implicit in the following diagrams will be *required* for T to be regarded as a monad:

Multiplication :

$$\begin{array}{ccc} \mathsf{T}^2\,A & \xleftarrow{\mu} & \mathsf{T}^3\,A \\ {\scriptstyle\mu}\downarrow & & \downarrow{\scriptstyle\mathsf{T}\,\mu} \\ \mathsf{T}\,A & \xleftarrow[\mu]{} & \mathsf{T}^2\,A \end{array} \qquad \mu \cdot \mu = \mu \cdot \mathsf{T}\,\mu \tag{4.5}$$

Unit :

$$\begin{array}{ccc} \mathsf{T}^2\,A & \xleftarrow{u} & \mathsf{T}\,A \\ {\scriptstyle\mu}\downarrow & \swarrow_{id} & \downarrow{\scriptstyle\mathsf{T}\,u} \\ \mathsf{T}\,A & \xleftarrow[\mu]{} & \mathsf{T}^2\,A \end{array} \qquad \mu \cdot u = \mu \cdot \mathsf{T}\,u = id \tag{4.6}$$

The simple and beautiful symmetries apparent in these diagrams will make it easy to memorize their laws and check them for particular cases. For instance, for the $(1+)$ monad, law (4.6) will read as follows:

$$[i_1\,, id] \cdot i_2 = [i_1\,, id] \cdot (id + i_2) = id$$

These equalities are easy to check.

In laws (4.5) and (4.6), the different instances of μ and u are differently typed, as these are polymorphic and exhibit natural properties:

μ-natural :

$$\begin{array}{ccc} A & \mathsf{T}\,A & \xleftarrow{\mu} & \mathsf{T}^2\,A \\ {\scriptstyle f}\downarrow & {\scriptstyle\mathsf{T}\,f}\downarrow & & \downarrow{\scriptstyle\mathsf{T}^2\,f} \\ B & \mathsf{T}\,B & \xleftarrow[\mu]{} & \mathsf{T}^2\,B \end{array} \qquad \mathsf{T}\,f \cdot \mu = \mu \cdot \mathsf{T}^2\,f \tag{4.7}$$

u-natural :

$$\begin{array}{ccc} A & \mathsf{T}\,A & \xleftarrow{u} & A \\ {\scriptstyle f}\downarrow & {\scriptstyle\mathsf{T}\,f}\downarrow & & \downarrow{\scriptstyle f} \\ B & \mathsf{T}\,B & \xleftarrow[u]{} & B \end{array} \qquad \mathsf{T}\,f \cdot u = u \cdot f \tag{4.8}$$

The simplest of all monads is the *identity monad* $\mathsf{T}\,X \stackrel{\text{def}}{=} X$, which is such that $\mu = id$, $u = id$ and $f \bullet g = f \cdot g$. So — in a sense — the *whole functional discipline* studied thus far was already *monadic*, living inside the simplest of all monads: the identity one. Put in other words, such functional discipline can be framed into a wider discipline in which an arbitrary monad is present. Such is the main ain of this chapter.

4.4.1 Properties involving (Kleisli) composition

The following properties arise from the definitions and monadic properties presented above:

$$f \bullet (g \bullet h) = (f \bullet g) \bullet h \tag{4.9}$$
$$u \bullet f = f = f \bullet u \tag{4.10}$$
$$(f \bullet g) \cdot h = f \bullet (g \cdot h) \tag{4.11}$$
$$(f \cdot g) \bullet h = f \bullet (\mathsf{T}\,g \cdot h) \tag{4.12}$$
$$id \bullet id = \mu \tag{4.13}$$

Properties (4.9) and (4.10) are the monadic counterparts of, respectively, (2.8) and (2.10), meaning that monadic composition preserves the properties of normal functional composition. In fact, for the identity monad, these properties coincide with each other.

Above we have shown that property (4.10) holds for the list monad, recall (4.2). A general proof can be produced similarly. We select property (4.9) as an illustration of the rôle of the monadic properties:

$$\begin{aligned}
& f \bullet (g \bullet h) \\
= & \quad \{ \text{ definition (4.4) twice } \} \\
& \mu \cdot \mathsf{T}\,f \cdot (\mu \cdot \mathsf{T}\,g \cdot h) \\
= & \quad \{ \ \mu \text{ is natural (4.7) } \} \\
& \mu \cdot \mu \cdot \mathsf{T}^2\,f \cdot \mathsf{T}\,g \cdot h \\
= & \quad \{ \text{ monad property (4.5) } \} \\
& \mu \cdot \mathsf{T}\,\mu \cdot \mathsf{T}^2\,f \cdot \mathsf{T}\,g \cdot h \\
= & \quad \{ \text{ functor T (3.55) } \}
\end{aligned}$$

$$\begin{array}{ll} & \mu \cdot \mathsf{T}\ (\mu \cdot \mathsf{T}\ f \cdot g) \cdot h \\ = & \qquad \{\ \text{definition (4.4) twice}\ \} \\ & (f \bullet g) \bullet h \end{array}$$

Clearly, this calculation generalizes that of exercise 4.1 to any monad T.

Exercise 4.4. *Prove the other laws above and the following two,*

$$(\mathsf{T}\ f) \cdot (h \bullet k) = (\mathsf{T}\ f \cdot h) \bullet k \tag{4.14}$$
$$(f \cdot g) \bullet h = f \bullet (\mathsf{T}\ g \cdot h) \tag{4.15}$$

where Kleilsi composition trades with normal composition.
□

4.5 Monadic application (binding)

The monadic counterpart of functional application ap (2.73) is another operator ap' which is intended to be "tolerant" in face of any "T-inflated" argument x:[2]

$$\begin{array}{c} (\mathsf{T}\ B)^A \times \mathsf{T}\ A \xrightarrow{ap'} \mathsf{T}\ B \\ ap'(f, x) = f'\ x = (\mu \cdot \mathsf{T}\ f)x \end{array} \tag{4.16}$$

If in curry/flipped format, monadic application is called *binding* and denoted by the symbol $\gg\!=$, looking very much like postfix functional application,

$$((\mathsf{T}\ B)^A)^{\mathsf{T}\ A} \xrightarrow{\gg\!=} \mathsf{T}\ B \tag{4.17}$$

that is:

$$x \gg\!= f \quad \stackrel{\text{def}}{=} \quad (\mu \cdot \mathsf{T}\ f)x \tag{4.18}$$

This operator will exhibit properties arising from its definition and the basic monadic properties, *e.g.*

$$x \gg\!= u$$

[2] Recall that notation A^B expresses the set of all functions $f : A \to B$.

$$
\begin{array}{ll}
\equiv & \{\ \text{definition (4.18)}\ \} \\
& (\mu \cdot \mathsf{T}\, u)x \\
\equiv & \{\ \text{law (4.6)}\ \} \\
& (id)x \\
\equiv & \{\ \text{identity function}\ \} \\
& x
\end{array}
$$

At pointwise level, one may chain monadic compositions from left to right, *e.g.*

$$(((x \gg\!= f_1) \gg\!= f_2) \gg\!= \ldots f_{n-1}) \gg\!= f_n$$

for functions $A \xrightarrow{f_1} \mathsf{T}\, B_1$, $B_1 \xrightarrow{f_2} \mathsf{T}\, B_2$, $\ldots$ $B_{n-1} \xrightarrow{f_n} \mathsf{T}\, B_n$.

4.6 Sequencing and the do-notation

Given two monadic values x and y, it becomes possible to "sequence" them, thus obtaining another of such value, by defining the following operator

$$x \gg y \ \stackrel{\text{def}}{=}\ x \gg\!= \underline{y} \tag{4.19}$$

of type $(\gg) : \mathsf{T}\ A \to \mathsf{T}\ B \to \mathsf{T}\ B$. For instance, within the finite-list monad, one has

$$[1,2] \gg [3,4] = (concat \cdot \underline{[3,4]}^{\star})[1,2] = concat[[3,4],[3,4]] = [3,4,3,4]$$

Because this operator is associative (prove this as an exercise), one may iterate it to more than two arguments and write, for instance,

$$x_1 \gg x_2 \gg \ldots \gg x_n$$

This leads to the popular "**do**-notation", which is another piece of (pointwise) notation which makes sense in a monadic context:

$$\mathbf{do}\ \{x_1; x_2; \ldots; x_n\} \ \stackrel{\text{def}}{=}\ x_1 \gg \mathbf{do}\ \{x_2; \ldots; x_n\}$$

for $n \geqslant 1$. For $n = 1$ one trivially has

$$\mathbf{do}\ x_1 \ \stackrel{\text{def}}{=}\ x_1$$

4.7 Generators and comprehensions

The **do**-notation accepts a variant in which the arguments of the $\gg$ operator are "generators" of the form

$$a \leftarrow x \tag{4.20}$$

where, for a of type A, x is an inhabitant of monadic type $\mathsf{T}\,A$. One may regard $a \leftarrow x$ as meaning "let a be taken from x". Then the **do**-notation unfolds as follows:

$$\mathbf{do}\ a \leftarrow x_1; x_2; \ldots; x_n \ \stackrel{\text{def}}{=}\ x_1 \gg\!= \lambda a \cdot (\mathbf{do}\ x_2; \ldots; x_n) \tag{4.21}$$

Of course, we should now allow for the x_i to range over terms involving variable a. For instance, by writing (again in the list-monad)

$$\mathbf{do}\ a \leftarrow [1,2,3]; [\,a^2\,] \tag{4.22}$$

we mean

$$\begin{array}{rl} & [1,2,3] \gg\!= \lambda a.[a^2] \\ = & \mathit{concat}((\lambda a.[a^2])^\star [1,2,3]) \\ = & \mathit{concat}[[1],[4],[9]] \\ = & [1,4,9] \end{array}$$

The analogy with classical set-theoretic ZF-notation, whereby one might write $\{a^2 \mid a \in \{1,2,3\}\}$ to describe the set of the first three perfect squares, calls for the following notation,

$$[\ a^2 \mid a \leftarrow [1,2,3]\ ] \tag{4.23}$$

as a "shorthand" of (4.22). This is an instance of the so-called *comprehension* notation, which can be defined in general as follows:

$$[\ e \mid a_1 \leftarrow x_1, \ldots, a_n \leftarrow x_n\ ] \ = \ \mathbf{do}\ \{a_1 \leftarrow x_1; \ldots; a_n \leftarrow x_n; u\ (e)\} \tag{4.24}$$

where u is the monad's unit (4.6,4.8).

Alternatively, comprehensions can be defined as follows, where p, q stand for arbitrary generators:

$$[t] \ = \ u\,t \tag{4.25}$$

$$[\ f\,x \mid x \leftarrow l\ ] \ = \ (\mathsf{T}\,f)l \tag{4.26}$$

$$[\ t \mid p, q\ ] \ = \ \mu[\ [\ t \mid q\ ] \mid p\ ] \tag{4.27}$$

Note, however, that comprehensions are not restricted to lists or sets — they can be defined for any monad T thanks to the **do**-notation.

Exercise 4.5. *Show that*

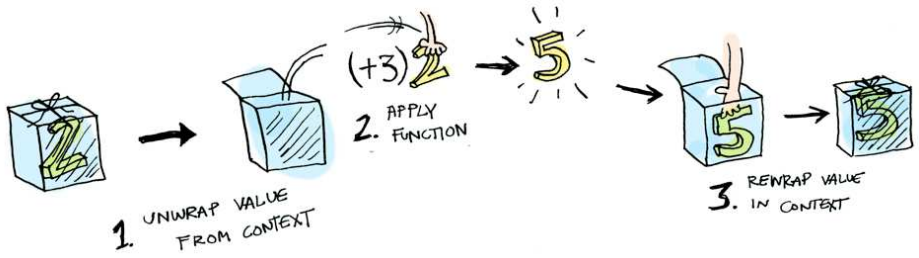
$$(f \bullet g)\ a = \mathbf{do}\ \{b \leftarrow g\ a; f\ b\} \quad (4.28)$$

$$\mathsf{T}\ f\ x = \mathbf{do}\ \{a \leftarrow x; u\ (f\ x)\} \quad (4.29)$$

Note that the second **do** *expression is equivalent to* $x \gg\!= (u \cdot f)$.
□

Exercise 4.6. *Show that* $x \gg\!= f = f \bullet id\ x = \mathbf{do}\ \{a \leftarrow x; f\ a\}$ *and then that* $(x \gg\!= g) \gg\!= f$ *is the same as* $x \gg\!= f \bullet g$.
□

Fact (4.29) is illustrated in the cartoon[3]



for the computation of $\mathsf{T}\ (+3)\ x$, where $x = u\ 2$ is the T-monadic object containing number 2.

4.8 Monads in HASKELL

In the *Standard Prelude* for HASKELL, one finds the following minimal definition of the $Monad$ class,

[3]Credits: see this and other helpful, artistic illustrations in
`http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.h`

$$\begin{array}{l}\textbf{class}\ Monad\ m\ \textbf{where}\\ \quad \mathsf{return} :: a \to m\ a\\ \quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b\end{array}$$

where return refers to the unit of m, on top of which the "sequence" operator

$$\begin{array}{l}(\gg) :: m\ a \to m\ b \to m\ b\\ fail\ :: String \to m\ a\end{array}$$

is defined by

$$p \gg q = p \gg\!= \lambda\_ \to q$$

as expected. This class is instantiated for finite sequences ($[\,]$), $Maybe$ and IO, among others.

The μ multiplication operator is function join in module `Monad.hs`:

$$\begin{array}{l}\mathsf{join} :: (Monad\ m) \Rightarrow m\ (m\ a) \to m\ a\\ \mathsf{join}\ x = x \gg\!= id\end{array}$$

This is easily justified:

$$\begin{array}{lcl} join\, x & = & x \gg\!= id \\ & = & \quad \{\ \text{definition (4.18)}\ \} \\ & & (\mu \cdot \mathsf{T}\, id)x \\ & = & \quad \{\ \text{functors commute with identity (3.54)}\ \} \\ & & (\mu \cdot id)x \\ & = & \quad \{\ \text{law (2.10)}\ \} \\ & & \mu\, x \end{array} \tag{4.30}$$

The following infix notation for (Kleisli) monadic composition in HASKELL uses the binding operator:

$$\begin{array}{l}(\bullet) :: Monad\ t \Rightarrow (b \to t\ c) \to (d \to t\ b) \to d \to t\ c\\ (f \bullet g)\ a = (g\ a) \gg\!= f\end{array}$$

4.8.1 Monadic I/O

IO, a parametric datatype whose inhabitants are special values called *actions* or *commands*, is a most relevant monad. Actions perform the interconnection between HASKELL and the environment (file system, operating system). For instance, $getLine :: \mathsf{IO}\ String$ is a particular action. Parameter `String` refers to the fact that this action "delivers" — or extracts — a string from the environent. This meaning is clearly conveyed by the type `String` assigned to symbol l in

$$\texttt{do}\ \ l \leftarrow getLine; \ldots l \ldots$$

which is consistent with typing rule for generators (4.20). Sequencing corresponds to the ";" syntax in most programming languages (*e.g.* C) and the `do`-notation is particulary intuitive in the IO-context.

Examples of functions delivering actions are

$$FilePath \xrightarrow{readFile} \mathsf{IO}\ String$$

and

$$Char \xrightarrow{putChar} \mathsf{IO}()$$

— both produce I/O commands as result.

As is to be expected, the implementation of the IO monad in HASKELL — available from the *Standard Prelude* — is not totally visible, for it is bound to deal with the intrincacies of the underlying machine:

$$\begin{array}{l}\textbf{instance}\ Monad\ \mathsf{IO}\ \textbf{where}\\ \quad (\gg\!=) = primbindIO\\ \quad \mathsf{return} = primretIO\end{array}$$

Rather interesting is the way IO is regarded as a functor:

$$\mathsf{fmap}\ f\ x = x \gg\!= (\mathsf{return} \cdot f)$$

This goes the other way round, the monadic structure "helping" in defining the functor structure, everything consistent with the underlying theory:

$$\begin{array}{lcl} x \gg\!= (u \cdot f) & = & (\mu \cdot \mathsf{IO}(u \cdot f))x \\ & = & \qquad \{\ \text{functors commute with composition}\ \} \\ & & (\mu \cdot \mathsf{IO}\, u \cdot \mathsf{IO}\, f)x \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ law (4.6) for } \mathsf{T} = \mathsf{IO} \ \} \\
& (\mathsf{IO}\ f)x \\
= & \{ \text{ definition of } fmap \ \} \\
& (fmap\ f)x
\end{array}
$$

For enjoyable reading on monadic input/output in HASKELL see [17], chapter 18.

Exercise 4.7. *Extend the* Maybe *monad to the following "error message" exception handling datatype:*

$$\mathbf{data}\ Error\ a = Err\ String \mid Ok\ a\ \mathbf{deriving}\ Show$$

In case of several error messages issued in a `do` *sequence, how many turn up on the screen? Which ones?*
□

Exercise 4.8. *Recalling section 3.13, show that any inductive type with base functor*

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

where F *is an arbitrary functor, forms a monad for*

$$\mu = (\!|[id, in \cdot i_2]|\!)$$
$$u = in \cdot i_1.$$

Identify F *for known monads such as eg.* Maybe, LTree *and (non-empty) lists.*
□

4.9 The state monad

The so-called *state monad* is a monad whose inhabitants are state-transitions encoding a particular brand of state-based automata known as *Mealy machines*.

Given a set A (input alphabet), a set B (output alphabet) and a set of states S, a deterministic Mealy machine (DMM) is specified by a transition function of type

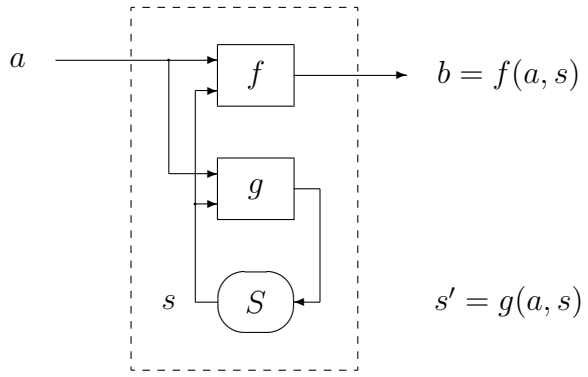
$$A \times S \xrightarrow{\delta} B \times S \tag{4.31}$$

Wherever $(b, s') = \delta(a, s)$, we say that the machine has transition

$$s \xrightarrow{a|b} s'$$

and refer to s as the **before** state, and to s' as the **after** state.

It is clear from (4.31) that δ can be expressed as the *split* of two functions f and g, $\delta = \langle f, g \rangle$, as depicted in the following diagram:



(4.32)

The information recorded in the state of a DMM is either meaningless to the user of the machine (as in eg. the case of states represented by numbers) or too complex to be perceived and handled explicitly (as is the case of eg. the data kept in a large database). So, it is convenient to *abstract* from it. Such an abstraction leads to the *state monad* in the following way: taking (4.31) and recalling (2.81), we simply transpose (ie. *curry*) δ and obtain

$$A \xrightarrow{\overline{\delta}} \underbrace{(B \times S)^S}_{(\mathsf{St}\ S)\ B} \tag{4.33}$$

thus "shifting" the input state to the output. In this way, $\overline{\delta}\, a$ is a function capturing all state-transitions (and corresponding outputs) for input a. For instance, the function which *appends* a new element to the back of a queue,

$$enq(a, s) \stackrel{\text{def}}{=} s +\!\!+ [a]$$

can be converted into a DMM by adding to it a dummy output of type 1 and then transposing:

$$\begin{array}{rcl} enqueue & : & A \to (1 \times S)^S \\ enqueue\ a & \stackrel{\text{def}}{=} & \langle !, (+\!\!+[a]) \rangle \end{array} \tag{4.34}$$

Action $enqueue$ performs enq on the state while acknowledging it by issuing an output of type 1.

Unit and multiplication. Let us show that

$$(\mathsf{St}\ S)\ A \ \cong \ (A \times S)^S \tag{4.35}$$

forms a monad. As we shall see, the fact that the *values* of this monad are functions brings the theory of exponentiation to the forefront. Thus, a review of section 2.14 is recommended at this point.

Notation $\widehat{f}$ will be used to abbreviate $uncurry\ f$, enabling the following variant of universal law (2.73),

$$\widehat{k} = f \ \Leftrightarrow \ f = ap \cdot (k \times id) \tag{4.36}$$

whose cancellation

$$\widehat{k} = ap \cdot (k \times id) \tag{4.37}$$

is written pointwise as follows:

$$\widehat{k}(c, a) \ = \ (k\ c)a \tag{4.38}$$

First of all, what is the functor behind (4.35)? Fixing the state space S, we obtain

$$\mathsf{T}X \ \stackrel{\text{def}}{=} \ (X \times S)^S \tag{4.39}$$

on objects and

$$\mathsf{T}f \ \stackrel{\text{def}}{=} \ (f \times id)^S \tag{4.40}$$

on functions, where $(\_)^S$ is the exponential functor (2.77).

The unit of this monad is the transpose of the simplest of all Mealy machines — the identity:

$$\begin{array}{lcl} u & : & A \to (A \times S)^S \\ u & = & \overline{id} \end{array} \tag{4.41}$$

Let us see what this means:

$$\begin{array}{ll} & u = \overline{id} \\ \equiv & \quad \{\ (2.73)\ \} \\ & ap \cdot (u \times id) = id \\ \equiv & \quad \{\ \text{introducing variables}\ \} \\ & ap(u\ a, s) = (a, s) \\ \equiv & \quad \{\ \text{definition of } ap\ \} \\ & (u\ a)s = (a, s) \end{array}$$

So, action $u\ a$ performed on state s keeps s unchanged and outputs a.

From the type of μ, for this monad,

$$((A \times S)^S \times S)^S \xrightarrow{\mu} (A \times S)^S$$

one figures out $\mu = x^S$ (recalling the exponential functor as defined by (2.77)) for x of type $((A \times S)^S \times S) \xrightarrow{x} (A \times S)$. This, on its turn, is easily recognized as an instance of the ap polymorphic function (2.73), which is such that $ap = \widehat{id}$, recall (2.75). Altogether, we define

$$\mu \quad = \quad ap^S \tag{4.42}$$

Let us inspect the behaviour of μ by checking the meaning of applying it to an action expressed as in diagram (2.81):

$$\begin{array}{ll} & \mu\langle f, g\rangle = ap^S\langle f, g\rangle \\ \equiv & \quad \{\ (2.77)\ \} \\ & \mu\langle f, g\rangle = ap \cdot \langle f, g\rangle \\ \equiv & \quad \{\ \text{extensional equality (2.5)}\ \} \\ & \mu\langle f, g\rangle s = ap(f\ s, g\ s) \\ \equiv & \quad \{\ \text{definition of } ap\ \} \\ & \mu\langle f, g\rangle s = (f\ s)(g\ s) \end{array}$$

We find out that μ "unnests" the action inside f by applying it to the state delivered by g.

Checking the monadic laws. The calculation of (4.6) is made in two parts, checking $\mu \cdot u = id$ first,

$$\begin{array}{ll} & \mu \cdot u \\ = & \quad \{ \text{ definitions } \} \\ & ap^S \cdot \overline{id} \\ = & \quad \{ \text{ exponentials absorption (2.78) } \} \\ & \overline{ap \cdot id} \\ = & \quad \{ \text{ reflection (2.75) } \} \\ & id \end{array}$$

and then checking $\mu \cdot (\mathsf{T}u) = id$:

$$\begin{array}{ll} & \mu \cdot (\mathsf{T}u) \\ = & \quad \{ \text{ (4.42,4.40) } \} \\ & ap^S \cdot (\overline{id} \times id)^S \\ = & \quad \{ \text{ functor } \} \\ & (ap \cdot (\overline{id} \times id))^S \\ = & \quad \{ \text{ cancellation (2.74) } \} \\ & id^S \\ = & \quad \{ \text{ functor } \} \\ & id \end{array}$$

The proof of (4.5) is also not difficult once supported by the laws of exponentials.

Kleisli composition. Let us calculate $f \bullet g$ for this monad:

$$\begin{array}{ll} & f \bullet g \\ = & \quad \{ \text{ (4.4) } \} \end{array}$$

$$
\begin{array}{ll}
 & \mu \cdot \mathsf{T}\, f \cdot g \\
= & \quad \{ \text{ (4.42) ; (4.40) } \} \\
 & ap^S \cdot (f \times id)^S \cdot g \\
= & \quad \{ \ (\_)^S \text{ is a functor } \} \\
 & (ap \cdot (f \times id))^S \cdot g \\
= & \quad \{ \text{ (4.36) } \} \\
 & \widehat{f}^S \cdot g \\
= & \quad \{ \text{ cancellation } \} \\
 & \widehat{f}^S \cdot \overline{\widehat{g}} \\
= & \quad \{ \text{ absorption (2.78) } \} \\
 & \overline{\widehat{f} \cdot \widehat{g}}
\end{array}
$$

In summary, we have:

$$f \bullet g \ = \ \overline{\widehat{f} \cdot \widehat{g}} \tag{4.43}$$

which can be written alternatively as

$$\widehat{f \bullet g} \ = \ \widehat{f} \cdot \widehat{g}$$

Let us use this in calculating law

$$pop \bullet push \ = \ u \tag{4.44}$$

where $push$ and pop are such that

$$
\begin{array}{lcl}
push & : & A \to (1 \times S)^S \\
\widehat{push} & \stackrel{\text{def}}{=} & \langle !, \widehat{(:)} \rangle
\end{array}
\tag{4.45}
$$

$$
\begin{array}{lcl}
pop & : & 1 \to (A \times S)^S \\
\widehat{pop} & \stackrel{\text{def}}{=} & \langle head, tail \rangle \cdot \pi_2
\end{array}
\tag{4.46}
$$

for S the datatype of finite lists. We reason:

$$pop \bullet push$$

$$
\begin{array}{ll}
= & \quad \{\ (4.43)\ \} \\
& \overline{\widehat{pop} \cdot \widehat{push}} \\
= & \quad \{\ (4.45, 4.46)\ \} \\
& \overline{\langle head, tail\rangle \cdot \pi_2 \cdot \langle !, \widehat{(:)}\rangle} \\
= & \quad \{\ (2.20, 2.24)\ \} \\
& \overline{\langle head, tail\rangle \cdot \widehat{(:)}} \\
= & \quad \{\ out \cdot in = id \text{ (lists)}\ \} \\
& \overline{id} \\
= & \quad \{\ (4.41)\ \} \\
& u
\end{array}
$$

Bind. The effect of binding a state transition x to a state-monadic function h is calculated in a similar way:

$$
\begin{array}{ll}
& x \gg\!= h \\
= & \quad \{\ (4.18)\ \} \\
& (\mu \cdot \mathsf{T}h)x \\
= & \quad \{\ (4.42) \text{ and } (4.40)\ \} \\
& (ap^S \cdot (h \times id)^S)x \\
= & \quad \{\ (\_)^S \text{ is a functor}\ \} \\
& (ap \cdot (h \times id))^S x \\
= & \quad \{\ \text{cancellation } (4.37)\ \} \\
& \widehat{h}^S x \\
= & \quad \{\ \text{exponential functor } (2.77)\ \} \\
& \widehat{h} \cdot x
\end{array}
$$

Let us unfold $\widehat{h} \cdot x$ by splitting x into its components two components f and g:

$$\langle f, g\rangle \gg\!= h \quad = \quad \widehat{h} \cdot \langle f, g\rangle$$

$$
\begin{array}{ll}
\equiv & \{ \text{ go pointwise } \} \\
& (\langle f, g\rangle \gg\!= h)s \;=\; \widehat{h}(\langle f, g\rangle s) \\
\equiv & \{ \text{ (2.18) } \} \\
& (\langle f, g\rangle \gg\!= h)s \;=\; \widehat{h}(f\ s, g\ s) \\
\equiv & \{ \text{ (4.38) } \} \\
& (\langle f, g\rangle \gg\!= h)s \;=\; h(f\ s)(g\ s)
\end{array}
$$

In summary, for a given "before state" s, $g\ s$ is the intermediate state upon which $f\ s$ runs and yields the output and (final) "after state".

Two prototypical inhabitants of the state monad: get and put. These generic actions are defined as follows, in the PF-style:

$$get \stackrel{\text{def}}{=} \langle id, id\rangle \tag{4.47}$$

$$put \stackrel{\text{def}}{=} \overline{\langle !, \pi_1\rangle} \tag{4.48}$$

Action g retrieves the data stored in the state without changing it, while put — which can also be written

$$put\ s \;=\; \langle !, \underline{s}\rangle \tag{4.49}$$

or even as

$$put\ s \;=\; modify\ \underline{s} \tag{4.50}$$

where

$$modify\ f \stackrel{\text{def}}{=} \langle !, f\rangle \tag{4.51}$$

updates the state via state-to-state function f — stores a particular value in the state.

The following is an example, in Haskell, of the standard use of get/put in managing context data, in this case a counter. The function decorates each node of a $BTree$ (recall this datatype from page 106) with its position in the tree:

$$
\begin{array}{l}
decBTree\ Empty = \mathsf{return}\ Empty \\
decBTree\ (Node\ (a, (t1, t2))) = \mathbf{do}\ \{
\end{array}
$$

$n \leftarrow get;$
$put\ (n+1);$
$x \leftarrow decBTree\ t1;$
$y \leftarrow decBTree\ t2;$
return $(Node\ ((a,n),(x,y)))$
$\}$

To close the chapter, we will present a strategy for deriving this kind of monadic functions.

4.10 'Monadification' of Haskell code made easy

There is an easy roadmap for "monadification" of Haskell code. What do we mean by *monadification*? Well, in a sense — as we shall soon see — every piece of code is monadic: we don't notice this because the underlying monad is *invisible* (the *identity* monad). We are going to see how to make it visible taking advantage of monadic `do` notation and leaving it open for instantiation. This will bridge the gap between monads' theory and its application to handling particular effects in concrete programming situations.

Let us take as starting point the pointwise version of sum, the list catamorphism which adds all numbers found in its input:

$$\begin{aligned}&\mathsf{sum}\ [\,] = 0\\ &\mathsf{sum}\ (h:t) = h + \mathsf{sum}\ t\end{aligned}$$

Notice that this code could have been written as follows

$$\begin{aligned}&\mathsf{sum}\ [\,] = id\ 0\\ &\mathsf{sum}\ (h:t) = \mathbf{let}\ x = \mathsf{sum}\ t\ \mathbf{in}\ id\ (h+x)\end{aligned}$$

using let notation and two instances of the identity function. Question: what is the point of such a "baroque" version of the starting, so simple piece of code? Answer:

- The let ... in ... notation stresses the fact that recursive call happens *earlier* than the delivery of the result.
- The id functions signal the exit points of the algorithm, that is, the points where it *returns* something to the caller.

Next, let us

- re-write id into return—;
- re-write let x = ... in ...— into `do { x <- ... ; ... }`

One will obtain the following version of sum:

$$msum\ [\,] = \mathsf{return}\ 0$$
$$msum\ (h : t) = \mathbf{do}\ \{x \leftarrow msum\ t; \mathsf{return}\ (h + x)\}$$

Typewise, while sum has type $(Num\ a) \Rightarrow [a] \rightarrow a$, $msum$ has type

$$(Monad\ m, Num\ a) \Rightarrow [a] \rightarrow m\ a$$

That is, $msum$ is monadic — parametric on monad m — while sum is not.

There is a particular monad for which sum and $msum$ coincide: the **identity** monad $\mathsf{Id}\ X = X$. It is very easy to show that inside this monad return— is the identity and do— means the same as let— — enough for the pointwise versions of the two functions to be the same. Thus the "invisible" monad mentioned earlier is the identity monad.

In summary, the monadic version is *generic* in the sense that it runs on whatever monad you like, enabling you to perform *side effects* while the code runs. If you don't need any effects then you get the "non-monadic" version as special case, as seen above. Otherwise, Haskell will automatically switch to the effects you want, depending on the monad you choose (often determined by context).

For each particular monad we may decide to add specific monadic code like `get` and `put` in the `decBTree` example, where we want to take advantage of the state monad. As another example, check the following enrichment of $msum$ with state-monadic code helping you to trace the execution of your program:

$$msum'\ [\,] = \mathsf{return}\ 0$$
$$msum'\ (h : t) =$$
$$\quad \mathbf{do}\ \{x \leftarrow msum'\ t;$$
$$\quad\quad print\ (\texttt{"x= "} +\!\!+\ show\ x);$$
$$\quad\quad \mathsf{return}\ (h + x)\}$$

Thus one obtains traces the code in the way prescribed by the particular usage of the $print$ (state monadic) function:

```
*Main> msum' [3,5,1,3,4]
"x= 0"
"x= 4"
"x= 7"
"x= 8"
"x= 13"
*Main>
```

In the reverse direction, one may try and see what happens to monadic code upon removing all monad-specific functions and going into the identity monad once it gets monad generic. In the case of `decBTree`, for instance, we will get

$$
\begin{aligned}
&decBTree\ Empty = \mathsf{return}\ Empty\\
&decBTree\ (Node\ (a,(t1,t2))) =\\
&\quad \mathbf{do}\\
&\qquad x \leftarrow decBTree\ t1;\\
&\qquad y \leftarrow decBTree\ t2;\\
&\qquad \mathsf{return}\ (Node\ (a,(x,y)))
\end{aligned}
$$

once `get` and `put` are removed (and therefore all instances of `n`), and then

$$
\begin{aligned}
&decBTree\ Empty = Empty\\
&decBTree\ (Node\ (a,(t1,t2))) =\\
&\quad \mathbf{let}\\
&\qquad x = decBTree\ t1\\
&\qquad y = decBTree\ t2\\
&\quad \mathbf{in}\ Node\ (a,(x,y))
\end{aligned}
$$

This is the identity function on type BTree, recall the cata-reflection law (3.68). So, the *archetype* of (inspiration for) much monadic code is the most basic of all tree traversal functions — the identity [4]. The same could be said about imperative code of a particular class — the *recursive descent* one — much used in construction, for instance.

Playing with effects

As it may seem from the previous examples, adding effects to produce monadic code is far from arbitrary. This can be further appreciated by defining the function that yields the smallest element of a list,

[4] We have seen the same kind of "inspiration" before in building type functors (3.76) which, for $f = id$, boil down to the identity.

$$
\begin{aligned}
&getmin\ [a] = a \\
&getmin\ (h : t) = min\ h\ (getmin\ t)
\end{aligned}
$$

which is incomplete in the sense that it does not specify the meaning of $getmin\ [\,]$. As this is mathematically undefined, it should be expressed "outside the maths", that is, as an effect. Thus, to complete the defintion we first go monadic, as we did before,

$$
\begin{aligned}
&mgetmin\ [a] = \mathsf{return}\ a \\
&mgetmin\ (h : t) = \mathbf{do}\ \{x \leftarrow mgetmin\ t; \mathsf{return}\ (min\ h\ x)\} \\
&\lambda end\ \{verbatim\}
\end{aligned}
$$

and then chose a monad in which to express the meaning of $getmin\ [\,]$, for instance the `Maybe` monad

$$
\begin{aligned}
&mgetmin\ [\,] = \mathsf{Nothing} \\
&mgetmin\ [a] = \mathsf{return}\ a \\
&mgetmin\ (h : t) = \mathbf{do}\ \{x \leftarrow mgetmin\ t; \mathsf{return}\ (min\ h\ x)\}
\end{aligned}
$$

Alternatively, we might have written

$$mgetmin\ [\,] = Error\ \texttt{"Empty input"}$$

going into the `Error` monad, or even the simpler (yet interesting) $mgetmin\ [\,] = [\,]$, which shifts the code into the list monad, yielding singleton lists in the success case, otherwise the empty list.

Function $getmin$ above is an example of a partial function, that is, a function which is undefined for some of its inputs. These functions cause much interference in functional programming, which monads help us to keep under control.

Let us see how such interference is coped with in the case of higher order functions, taking map as example

$$
\begin{aligned}
&\mathsf{map}\ f\ [\,] = [\,] \\
&\mathsf{map}\ f\ (h : t) = (f\ h) : \mathsf{map}\ f\ t
\end{aligned}
$$

and supposing f is not a total function. How do we cope with erring evaluations of $f\ h$? As before, we first "letify" the code,

$$
\begin{aligned}
&\mathsf{map}\ f\ [\,] = [\,] \\
&\mathsf{map}\ f\ (h : t) = \mathbf{let}
\end{aligned}
$$

$$b = f\ h$$
$$x = \mathsf{map}\ f\ t\ \mathbf{in}\ b : x$$

we go monadic in the usual way,

$$mmap\ f\ [\,] = \mathsf{return}\ [\,]$$
$$mmap\ f\ (h : t) = \mathbf{do}\ \{\, b \leftarrow f\ h; x \leftarrow mmap\ f\ t; \mathsf{return}\ (b : x)\}$$

and everything goes smoothly — as can be checked, the function thus built is of the expected (monadic) type:

$$mmap :: (Monad\ m) \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ [b]$$

Run $mmap$ Just $[1, 2, 3, 4]$, for instance: you will obtain Just $[1, 2, 3, 4]$. Now run $mmap\ print\ [1, 2, 3, 4]$. You will see the items in the sequence printed sequentially.

One may wonder about the behaviour of the $mmap$ for f the identity function: will we get an error? No, we get a well-typed function of type $[m\ a] \rightarrow m\ [a]$, for m a monad. We thus obtain the well-known monadic function sequence which evaluates each *action* in the input sequence, from left to right, collecting the results. For instance, applying this function to input sequence $[\mathsf{Just}\ 1, \mathsf{Nothing}, \mathsf{Just}\ 2]$ the output will be Nothing.

There is much more one could say about monadic programming, but this is enough to see *where such a programming style comes from*.

Exercise 4.9. *Use the* monadification *technique to encode monadic function*

$$filterM :: Monad\ m \Rightarrow (a \rightarrow m\ \mathbb{B}) \rightarrow [a] \rightarrow m\ [a]$$

which generalizes the list-based filter function.
□

Exercise 4.10. *"Reverse" the following monadic code into its non-monadic archetype:*

$$f :: (Monad\ m) \Rightarrow (a \rightarrow m\ \mathbb{B}) \rightarrow [a] \rightarrow m\ [a]$$
$$f\ p\ [\,] = \mathsf{return}\ [\,]$$
$$f\ p\ (h : t) = \mathbf{do}\ \{$$
$$\quad b \leftarrow p\ h;$$
$$\quad t' \leftarrow f\ p\ t;$$

```
    return (if b then h : t' else [])
    }
```

Which function of the Haskell Prelude do you get by such reverse monadification*?*
□

4.11 Where do monads come from?

In the current context, a good way to find an answer this question is to recall the universal property of exponentials (2.73):

$$k = \overline{f} \Leftrightarrow f = ap \cdot (k \times id) \qquad \begin{array}{ccc} B^A & & B^A \times A & \xrightarrow{ap} & B \\ {\scriptstyle k=\overline{f}}\uparrow & & {\scriptstyle k\times id}\uparrow & \nearrow_{f} & \\ C & & C \times A & & \end{array}$$

Let us re-draw this diagram by unfolding $B^A \times A$ into the composition of two functors $\mathsf{G}\ (\mathsf{F}\ B)$ where $\mathsf{F}\ X = X^A$ and $\mathsf{G}\ X = X \times A$:

$$k = \overline{f} \ \Leftrightarrow \ f = \underbrace{ap \cdot \mathsf{G}\ k}_{\widehat{k}} \qquad \begin{array}{ccc} \mathsf{F}\ B & & \mathsf{G}\ (\mathsf{F}\ B) & \xrightarrow{ap} & B \\ {\scriptstyle k=\overline{f}}\uparrow & & {\scriptstyle \mathsf{G}\ k}\uparrow & \nearrow_{f} & \\ C & & \mathsf{G}\ C & & \end{array}$$

As we already know, this establishes the $(curry/uncurry)$ isomorphism

$$\mathsf{G}\ C \to B \ \cong \ C \to \mathsf{F}\ B \tag{4.52}$$

assuming F and G as defined above.

Note how (4.52) expresses a kind of "shunting rule" at type level: Gs on the input side can be "shunted" to the output if replaced by Fs. This is exactly what $curry$ and $uncurry$ do typewise. The corollaries of the universal property can also be expressed in terms of F and G:

- Reflection: $\overline{ap} = id$, that is, $ap = \widehat{id}$ – recall (2.75)
- Cancellation: $\widehat{id} \cdot \mathsf{G}\ \overline{f} = f$ – recall (2.74)

- Fusion: $\overline{h} \cdot g = \overline{h \cdot \mathsf{G}\ g}$ — recall (2.76)
- Absorption: $(\mathsf{F}\ g) \cdot \overline{h} = \overline{g \cdot h}$ — recall (2.78)
- Naturality: $h \cdot \overline{id} = \overline{id} \cdot \mathsf{G}\ (\mathsf{F}\ h)$
- Functor: $\mathsf{F}\ h = \overline{h \cdot ap}$
- Closed definitions: $\widehat{k} = ap \cdot (\mathsf{G}\ k)$ and $\overline{g} = (\mathsf{F}\ g) \cdot \overline{id}$, the latter following from absorption.

Now observe what happens if the functor composition $\mathsf{G}\cdot\mathsf{F}$ is swapped: $\mathsf{F}\ (\mathsf{G}\ X) = (X \times A)^A$. We get the *state monad* out of this construction,

$$(\mathsf{G} \cdot \mathsf{F})\ X = (X \times A)^A = St\ A\ X$$

— recall (4.35). Interestingly, the same universal property can be expressed in terms of such a monad structure, as the simple calculation shows,

$$\begin{array}{ll} & k = \overline{f}\ \Leftrightarrow\ ap \cdot \mathsf{G}\ k = f \\ \equiv & \quad \{\ \text{see above}\ \} \\ & k = (\mathsf{F}\ f) \cdot \overline{id}\ \Leftrightarrow\ f = \widehat{k} \\ \equiv & \quad \{\ \text{swapping variables } k \text{ and } f\text{, to match the starting diagram}\ \} \\ & f = (\mathsf{F}\ k) \cdot \overline{id}\ \Leftrightarrow\ k = \widehat{f} \\ \Box & \end{array}$$

that is,

$$k = \widehat{f}\ \Leftrightarrow\ f = \underbrace{\mathsf{F}\ k \cdot \eta}_{\overline{k}} \qquad \begin{array}{ccc} \mathsf{G}\ B & & \mathsf{F}\ (\mathsf{G}\ B) & \xleftarrow{\eta} & B \\ {\scriptstyle k=\widehat{f}}\downarrow & & {\scriptstyle \mathsf{F}\ k}\downarrow & \swarrow_{f} & \\ C & & \mathsf{F}\ A & & \end{array}$$

for $\eta = \overline{id}$, the unit of the monad $\mathsf{T} = \mathsf{F} \cdot \mathsf{G}$. To complete the definition of the T monad in this way, we recall (4.42)

$$\mu = \mathsf{F}\ \widehat{id} \tag{4.53}$$

with type $(\mathsf{T} \cdot \mathsf{T})\ X \xrightarrow{\mu} \mathsf{T}\ X$, where $id : \mathsf{T}\ X \to \mathsf{T}\ X$.

Adjunctions

The reasoning we have made above for exponentials and the state monad generalizes for any other monad. In general, isomorphisms of shape (4.52) are called an *adjunction* of the two functors F and G, which are said to be *adjoint* to each other. One writes $\mathsf{G} \dashv \mathsf{F}$ and says that G is *left* adjoint and that F is *right* adjoint. Using notation $\lfloor k \rfloor$ and $\lceil k \rceil$ for the generic witnesses of the isomorphism we write

$$\mathsf{G}\ C \to B \quad \underset{\lfloor - \rfloor}{\overset{\lceil - \rceil}{\cong}} \quad C \to \mathsf{F}\ B \tag{4.54}$$

From this a monad $\mathsf{T} = \mathsf{F} \cdot \mathsf{G}$ arises defined by $\eta = \lceil id \rceil$ and $\mu = \mathsf{F}\ \lfloor id \rfloor$.

Let us see another example of a monad arising from an adjunction (4.54). Recall exercise 2.23, on page 43, where pair / unpair witness an isomorphism similar to that of $curry/uncurry$, for $\mathsf{pair}\ (f, g) = \langle f, g \rangle$ and $\mathsf{unpair}\ k = (\pi_1 \cdot k, \pi_2 \cdot k)$. This can be cast into an adjunction as follows

$$\begin{array}{ll} & k = \mathsf{pair}\ (f, g)\ \Leftrightarrow\ (\pi_1 \cdot k, \pi_2 \cdot k) = (f, g) \\ \equiv & \quad \{\ \text{see below}\ \} \\ & k = \mathsf{pair}\ (f, g)\ \Leftrightarrow\ (\pi_1, \pi_2) \cdot (\mathsf{G}\ k) = (f, g) \end{array}$$

where $\mathsf{G}\ k = (k, k)$. Note the abuse of notation, on the righthand side, of extending function composition notation to composition of *pairs* of functions, defined in the expected way: $(f, g) \cdot (h, k) = (f \cdot h, g \cdot k)$. Note that, for $f : A \to B$ and $g : C \to D$, the pair (f, g) has type $(A \to B) \times (C \to D)$. However, we shall abuse of notation again and declare the type $(f, g) : (A, C) \to (B, D)$.[5] In the opposite direction, $\mathsf{F}\ (f, g) = f \times g$:

$$\begin{array}{ccc} B \times A & & (B \times A, B \times A) \xrightarrow{(\pi_1, \pi_2)} (B, A) \\ {\scriptstyle k = \mathsf{pair}\ (f,g)}\uparrow & & {\scriptstyle (k,k)}\uparrow \quad \nearrow_{(f,g)} \\ C & & (C, C) \end{array}$$

This is but another way of writing the universal property of products (2.55), since $(f, g) = (h, k)\ \Leftrightarrow\ f = h \wedge g = k$ and $\mathsf{pair}\ (f, g) = \langle f, g \rangle$, recall exercise 2.23.

[5] Strictly speaking, we are not abusing notation but rather working on a new *category*, that is, another mathematical system where functions and objects always come in pairs. For more on categories see the standard textbook [24].

What is, then, the monad behind this *pairing* adjunction? It is the *pairing monad* $(\mathsf{F} \cdot \mathsf{G})\ X = \mathsf{F}\ (\mathsf{G}\ X) = \mathsf{F}\ (X, X) = X \times X$, where $\eta = \langle id, id \rangle$ and $\mu = \pi_1 \times \pi_2$. This monad allows us to work with pairs regarded as 2-dimensional *vectors* (y, x). For instance, the **do**-expression

$$\mathbf{do}\ \{x \leftarrow (2,3); y \leftarrow (4,5); \mathsf{return}\ (x + y)\}$$

yields $(6, 8)$ as result in this monad — the vectorial sum of vectors $(2, 3)$ and $(4, 5)$. Below we show a simple encoding of this monad in Haskell:

$$\begin{array}{l}
\mathbf{data}\ P\ a = P\ (a, a)\ \mathbf{deriving}\ \mathit{Show} \\
\mathbf{instance}\ \mathit{Functor}\ P\ \mathbf{where} \\
\quad \mathsf{fmap}\ f\ (P\ (a, b)) = P\ (f\ a, f\ b) \\
\mathbf{instance}\ \mathit{Monad}\ P\ \mathbf{where} \\
\quad x \gg\!= f = (\mathit{muP} \cdot \mathsf{fmap}\ f)\ x \\
\quad \mathsf{return}\ a = P\ (a, a) \\
\mathit{muP} :: P\ (P\ a) \to P\ a \\
\mathit{muP}\ (P\ (P\ (a, b), P\ (c, d))) = P\ (a, d)
\end{array}$$

Exercise 4.11. *What is the vectorial operation expressed by the definition*

$$op\ k\ v = \mathbf{do}\ \{x \leftarrow v; \mathsf{return}\ (k \times x)\}$$

in the pairing monad?
□

4.12 Bibliography notes

The use of monads in computer science started with Moggi [29], who had the idea that monads should supply the extra semantic information needed to implement the lambda-calculus theory. Haskell [22] is among the computer languages which make systematic use of monads for implementing effects and imperative constructs in an otherwise purely functional language.

Category theorists invented monads in the 1960's to concisely express certain aspects of universal algebra. Functional programmers invented list comprehensions in the 1970's to concisely express certain programs involving lists. Philip

Wadler [37] made a great contribution to the field by showing that list comprehensions could be generalised to arbitrary monads and unify with imperative "do"-notation in case of the monad which explains imperative computations.

Monads are nowadays an essential feature of functional programming and are used in fields as diverse as language parsing [18], component-oriented programming [4], strategic programming [23], multimedia [17] and probabilistic programming [8]. Adjunctions play a major role in [15].

Part II

Calculating with Relations

Chapter 5

Specifying functional programs

not given in the current version of this textbook

Chapter 6

When everything becomes a relation

not given in the current version of this textbook

Chapter 7

Theorems for free: a calculational approach

not given in the current version of this textbook

Chapter 8

Design by Contract — calculationally

not given in the current version of this textbook

Chapter 9

Programs as Relational Hylomorphisms

not given in the current version of this textbook

Chapter 10

Quasi-inductive datatypes

not given in the current version of this textbook

Chapter 11

Calculational Program Refinement

not given in the current version of this textbook

Chapter 12

Relational thinking

not given in the current version of this textbook

Part III

Calculating with Matrices

Chapter 13

Towards a Linear Algebra of Programming

not given in the current version of this textbook

will build upon references [32, 30, 34]

Appendix A

Appendix

A.1 Haskell support library

$$\textbf{infix}\ 5\ \times$$
$$\textbf{infix}\ 4\ +$$

Products

$$\langle \cdot, \cdot \rangle :: (a \to b) \to (a \to c) \to a \to (b, c)$$
$$\langle f, g \rangle\ x = (f\ x, g\ x)$$
$$(\times) :: (a \to b) \to (c \to d) \to (a, c) \to (b, d)$$
$$f \times g = \langle f \cdot \pi_1, g \cdot \pi_2 \rangle$$

The 0-adic split is the unique function of its type

$$(!) :: a \to ()$$
$$(!) = \underline{()}$$

Renamings:

$$\pi_1 = \mathsf{fst}$$
$$\pi_2 = \mathsf{snd}$$

Coproduct

Renamings:

$$i_1 = i_1$$
$$i_2 = i_2$$

Either is predefined:

$$(+) :: (a \to b) \to (c \to d) \to a + c \to b + d$$
$$f + g = [i_1 \cdot f\,, i_2 \cdot g]$$

McCarthy's conditional:

$$p \to f, g = [f\,, g] \cdot p?$$

Exponentiation

Curry is predefined.

$$ap :: (a \to b, a) \to b$$
$$ap = \widehat{(\$)}$$

Functor:

$$\cdot^{\cdot} :: (b \to c) \to (a \to b) \to a \to c$$
$$f^{\cdot} = \overline{f \cdot ap}$$

Pair to predicate isomorphism (2.95):

$$p2p :: (b, b) \to \mathbb{B} \to b$$
$$p2p\ p\ b = \mathbf{if}\ b\ \mathbf{then}\ (\mathsf{snd}\ p)\ \mathbf{else}\ (\mathsf{fst}\ p)$$

The exponentiation functor is $(a \to)$ predefined:

$$\mathbf{instance}\ \mathit{Functor}\ ((\to)\ s)\ \mathbf{where}$$
$$\quad \mathsf{fmap}\ f\ g = f \cdot g$$

Others

$\underline{\cdot} :: a \to b \to a$ such that $\underline{a}\ x = a$ is predefined. Guards:

$$\cdot? :: (a \to \mathbb{B}) \to a \to a + a$$
$$p?\ x = \mathbf{if}\ p\ x\ \mathbf{then}\ i_1\ x\ \mathbf{else}\ i_2\ x$$

Natural isomorphisms

$$\mathsf{swap} :: (a, b) \to (b, a)$$
$$\mathsf{swap} = \langle \pi_2, \pi_1 \rangle$$

$$\mathsf{assocr} :: ((a, b), c) \to (a, (b, c))$$
$$\mathsf{assocr} = \langle \pi_1 \cdot \pi_1, \mathsf{snd} \times id \rangle$$

$$\mathsf{assocl} :: (a, (b, c)) \to ((a, b), c)$$
$$\mathsf{assocl} = \langle id \times \pi_1, \pi_2 \cdot \pi_2 \rangle$$

$$\mathsf{undistr} :: (a, b) + (a, c) \to (a, b + c)$$
$$\mathsf{undistr} = [id \times i_1 , id \times i_2]$$

$$\mathsf{undistl} :: (b, c) + (a, c) \to (b + a, c)$$
$$\mathsf{undistl} = [i_1 \times id , i_2 \times id]$$

$$\mathsf{coswap} :: a + b \to b + a$$
$$\mathsf{coswap} = [i_2 , i_1]$$

$$\mathsf{coassocr} :: (a + b) + c \to a + (b + c)$$
$$\mathsf{coassocr} = [id + i_1 , i_2 \cdot i_2]$$

$$\mathsf{coassocl} :: b + (a + c) \to (b + a) + c$$
$$\mathsf{coassocl} = [i_1 \cdot i_1 , i_2 + id]$$

$$\mathsf{distr} = \widehat{f} \ \textbf{where} \ f \ a = (g \ a) + (g \ a) \ \textbf{where} \ g \ a \ x = (a, x)$$

$$\mathsf{distl} = (\mathsf{swap} + \mathsf{swap}) \cdot \mathsf{distr} \cdot \mathsf{swap}$$

$$\mathit{flatr} :: (a, (b, c)) \to (a, b, c)$$
$$\mathit{flatr} \ (a, (b, c)) = (a, b, c)$$

$$\mathit{flatl} :: ((a, b), c) \to (a, b, c)$$
$$\mathit{flatl} \ ((b, c), d) = (b, c, d)$$

$$br = \langle id, ! \rangle$$

$$bl = \mathsf{swap} \cdot br$$

Class bifunctor

$$\textbf{class} \ \mathit{BiFunctor} \ f \ \textbf{where}$$
$$\quad \mathit{bmap} :: (a \to b) \to (c \to d) \to (f \ a \ c \to f \ b \ d)$$

$$\textbf{instance} \ \mathit{BiFunctor} \ \ \cdot + \cdot \ \ \textbf{where}$$
$$\quad \mathit{bmap} \ f \ g = f + g$$

$$\textbf{instance} \ \mathit{BiFunctor} \ (,) \ \textbf{where}$$
$$\quad \mathit{bmap} \ f \ g = f \times g$$

Monads

Kleisli monadic composition:

```
infix 4 •
(•) :: Monad a ⇒ (b → a c) → (d → a b) → d → a c
(f • g) a = (g a) >>= f
```

Multiplication, also known as join:

```
mult :: (Monad m) ⇒ m (m b) → m b
mult = (>>=id)
```

Monadic binding:

```
ap' :: (Monad m) ⇒ (a → m b, m a) → m b
ap' = flip (>>=)
```

List monad:

```
singl :: a → [a]
singl = return
```

Strong monads:

```
class (Functor f, Monad f) ⇒ Strong f where
  rstr :: (f a, b) → f (a, b)
  rstr (x, b) = do a ← x; return (a, b)
  lstr :: (b, f a) → f (b, a)
  lstr (b, x) = do a ← x; return (b, a)
instance Strong IO
instance Strong []
instance Strong Maybe
```

Double strength:

```
dstr :: Strong m ⇒ (m a, m b) → m (a, b)
dstr = rstr • lstr
```

Exercise 4.8.13 in Jacobs' "Introduction to Coalgebra" [19]:

```
splitm :: Strong ff ⇒ ff (a → b) → a → ff b
splitm = fmap ap · rstr
```

Monad transformers:

$$\textbf{class}\ (Monad\ m, Monad\ (t\ m)) \Rightarrow MT\ t\ m\ \textbf{where} \quad \text{-- monad transformer class}$$
$$\quad lift :: m\ a \to t\ m\ a$$

Nested lifting:

$$dlift :: (MT\ t\ (t1\ m), MT\ t1\ m) \Rightarrow m\ a \to t\ (t1\ m)\ a$$
$$dlift = lift \cdot lift$$

Basic functions, abbreviations

$$\mathsf{zero} = \underline{0}$$
$$\mathsf{one} = \underline{1}$$
$$\mathsf{nil} = \underline{[\,]}$$
$$\mathsf{cons} = \widehat{:}$$
$$\mathsf{add} = \widehat{+}$$
$$\mathsf{mul} = \widehat{*}$$
$$\mathsf{conc} = \widehat{++}$$
$$inMaybe :: () + a \to Maybe\ a$$
$$inMaybe = [\underline{\mathsf{Nothing}}, \mathsf{Just}]$$

More advanced

$$\textbf{class}\ (Functor\ f) \Rightarrow Unzipable\ f\ \textbf{where}$$
$$\quad unzp :: f\ (a, b) \to (f\ a, f\ b)$$
$$\quad unzp = \langle \mathsf{fmap}\ \pi_1, \mathsf{fmap}\ \pi_2 \rangle$$
$$\textbf{class}\ Functor\ g \Rightarrow DistL\ g\ \textbf{where}$$
$$\quad \lambda :: Monad\ m \Rightarrow g\ (m\ a) \to m\ (g\ a)$$
$$\textbf{instance}\ DistL\ [\,]\ \textbf{where}\ \lambda = \mathsf{sequence}$$
$$\textbf{instance}\ DistL\ Maybe\ \textbf{where}$$
$$\quad \lambda\ \mathsf{Nothing} = \mathsf{return}\ \mathsf{Nothing}$$
$$\quad \lambda\ (\mathsf{Just}\ a) = mp\ \mathsf{Just}\ a\ \textbf{where}\ mp\ f = (\mathsf{return} \cdot f) \bullet id$$

Convert Monad into Applicative:

$$aap :: Monad\ m \Rightarrow m\ (a \to b) \to m\ a \to m\ b$$
$$aap\ mf\ mx = \textbf{do}\ \{f \leftarrow mf; x \leftarrow mx; \mathsf{return}\ (f\ x)\}$$

A.2 Alloy support library

not given in the current version of this textbook

Bibliography

[1] C. Aarts, R.C. Backhouse, P. Hoogendijk, E.Voermans, and J. van der Woude. A relational theory of datatypes, December 1992. Available from `www.cs.nott.ac.uk/~rcb`.

[2] R.C. Backhouse. *Mathematics of Program Construction*. Univ. of Nottingham, 2004. Draft of book in preparation. 608 pages.

[3] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *CACM*, 21(8):613–639, August 1978.

[4] L.S. Barbosa. *Components as Coalgebras*. University of Minho, December 2001. Ph. D. thesis.

[5] R. Bird. Introduction to Functional Programming. Series in Computer Science. Prentice-Hall International, 2nd edition, 1998. C.A.R. Hoare, series editor.

[6] R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall, 1997.

[7] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, January 1977.

[8] M. Erwig and S. Kollmannsberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16:21–34, January 2006.

[9] R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19, pages 19–32. American Mathematical Society, 1967. Proc. Symposia in Applied Mathematics.

[10] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[11] J. Gibbons. Kernels, in a nut shell. *JLAMP*, 85(5, Part 2):921–930, 2016.

[12] J. Gibbons and R. Hinze. Just do it: simple monadic equational reasoning. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP'11, pages 2–14, New York, NY, USA, 2011. ACM.

[13] Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch. When is a function a fold or an unfold?, 2001. WGP, July 2001 (slides).

[14] Ralf Hinze. Adjoint folds and unfolds — an extended study. *Science of Computer Programming*, 78(11):2108–2159, 2013.

[15] Ralf Hinze. Adjoint folds and unfolds — an extended study. *Science of Computer Programming*, 78(11):2108–2159, 2013.

[16] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Conjugate hylomorphisms – or: The mother of all structured recursion schemes. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 527–538, New York, NY, USA, 2015. ACM.

[17] P. Hudak. The Haskell School of Expression - Learning Functional Programming Through Multimedia. Cambridge University Press, 1st edition, 2000. ISBN 0-521-64408-9.

[18] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), 1993.

[19] B. Jacobs. *Introduction to Coalgebra. Towards Mathematics of States and Observations*. Cambridge University Press, 2016.

[20] P. Jansson and J. Jeuring. Polylib — a library of polytypic functions. In *Workshop on Generic Programming (*WGP'98*), Marstrand, Sweden*, 1998.

[21] J. Jeuring and P. Jansson. Polytypic programming. In *Advanced Functional Programming*, number 1129 in LNCS, pages 68–114. Springer-Verlag, 1996.

[22] S.L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003. Also published as a Special Issue of the Journal of Functional Programming, 13(1) Jan. 2003.

[23] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P.L. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

[24] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[25] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[26] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986. D. Gries, series editor.

[27] J. McCarthy. Towards a mathematical science of computation. In C.M. Popplewell, editor, *Proc. of* IFIP *62*, pages 21–28, Amsterdam-London, 1963. North-Holland Pub. Company.

[28] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In S. Peyton Jones, editor, *Proceedings of Functional Programming Languages and Computer Architecture (FPCA95)*, 1995.

[29] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[30] D. Murta and J.N. Oliveira. A study of risk-aware program transformation. *SCP*, 110:51–77, 2015.

[31] P. Naur and B. Randell, editors. *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968*. Scientific Affairs Division, NATO, 1969.

[32] J.N. Oliveira. Towards a linear algebra of programming. *FAoC*, 24(4-6):433–458, 2012.

[33] J.N. Oliveira. Lecture notes on relational methods in software design, 2015. Available from `https://www.researchgate.net/publication/272476403_Lecture_Notes_on_Relational_Methods_in_Software_Design`

[34] J.N. Oliveira and V.C. Miraldo. "Keep definition, change category" — a practical approach to state-based system calculi. *JLAMP*, 85(4):449–474, 2016.

[35] M.S. Paterson and C.E. Hewitt. Comparative schematology. In *Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 119–127, August 1970.

[36] G. Villavicencio and J.N. Oliveira. *Reverse Program Calculation Supported by Code Slicing* . In *Proceedings of the Eighth Working Conference on Reverse Engineering (*WCRE 2001*) 2-5 October 2001, Stuttgart, Germany*, pages 35–46. IEEE Computer Society, 2001.

[37] P.L. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, pages 347–359, London, Sep. 1989. ACM.