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

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

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

$$\begin{cases} \mathsf{for}\ f\ i\ 0 = i \\ \mathsf{for}\ f\ i\ (n+1) = f\ (\mathsf{for}\ f\ i\ n) \end{cases} \tag{3.4}$$

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,

$$(a\times)\cdot[\underline{0}\,,(+1)] \quad = \quad [\underline{0}\,,(+a)]\cdot(id+(a\times)) \tag{3.5}$$

— thanks to +-absorption (2.41).

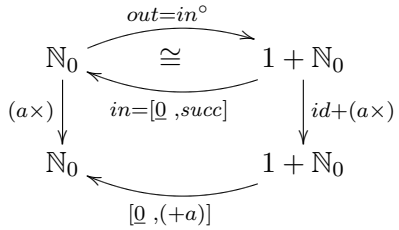
As we already know, equalities of compositions are nicely drawn as diagrams. That of (3.5) is as follows:

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

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.*

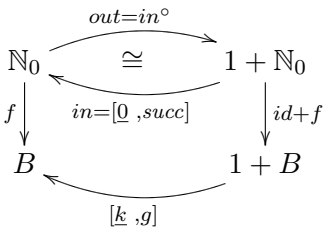
We will denote such an isomorphism by in and its converse by out in the following version of the same diagram



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 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 (3.4):

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

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":

$$f = \mathsf{for}\ g\ k \quad \equiv \quad f \cdot in = [\underline{k}\ , g] \cdot (id + f) \tag{3.6}$$

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

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 (3.6) for g and k we obtain the reflexion law:

$$\mathsf{for}\ succ\ 0 \quad = \quad id \tag{3.7}$$

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 (3.6). 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*

$$(a+) \quad = \quad \mathsf{for}\ succ\ a \tag{3.8}$$

□

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

$$h \cdot (\mathsf{for}\ g\ k) = \mathsf{for}\ j\ (h\ k) \quad \Leftarrow \quad h \cdot g = j \cdot h \tag{3.9}$$

can be derived from universal-property (3.6) [2]*. Since* $(a+) \cdot id = (a+)$*, provide an alternative derivation of (3.8) using the fusion-law above.*
□

Exercise 3.3. *Show that* $f = \mathsf{for}\ \underline{k}\ k$ *and* $g = \mathsf{for}\ id\ k$ *are the same program (function).*
□

Exercise 3.4. *Generic function* $k = \mathsf{for}\ f\ i$ *can be encoded in the syntax of C by writing*

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

for some predefined f*. Encode the functions* f *and* g *of exercise 3.3 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

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

```
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:

$$\begin{cases} L &= 1 + N \\ N &= A \times (1 + N) \end{cases} \tag{3.10}$$

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

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

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

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

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

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

System (3.10) is thus equivalent to:

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

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,

$$x = 1 + \frac{x}{2} \qquad ? \tag{3.12}$$

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

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

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 $b \neq 0$:

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

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

Back to (3.11), we would like to submit each of the equations, *e.g.*

$$L \quad = \quad 1 + A \times L \tag{3.13}$$

to a similar reasoning. Can we do it? The analogy which can be found between this equation and (3.12) goes beyond pattern similarity. From chapter 2 we know that many properties required in the reasoning above hold in the context of (3.13), 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 (2.83), 1 is the unit of multiplication (2.85), multiplication distributes over addition (2.50) *etc*. Moreover, the first step above assumed that addition is compatible (monotonic) with respect to equality,

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

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

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

— recall (2.44) for isos f and g.

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

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

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 (3.12) 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 equation (3.12). But we would have loved it in case it were the only known way, and this is precisely what happens with respect to (3.13). In this case we have:

$$L = 1 + A \times L$$

[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.

$$
\begin{aligned}
\equiv \quad & \{ \text{ substitution of } 1 + A \times L \text{ for } L \} \\
& L = 1 + A \times (1 + A \times L) \\
\equiv \quad & \{ \text{ distributive property (2.50) } \} \\
& L \cong 1 + A \times 1 + A \times (A \times L) \\
\equiv \quad & \{ \text{ unit of product (2.85) and associativity of product (2.32) } \} \\
& L \cong 1 + A + (A \times A) \times L \\
\equiv \quad & \{ \text{ by (2.86), (2.88) and (2.91) } \} \\
& L \cong A^0 + A^1 + A^2 \times L \\
\equiv \quad & \{ \text{ another substitution as above and similar simplifications } \} \\
& L \cong A^0 + A^1 + A^2 + A^3 \times L \\
\equiv \quad & \{ \text{ after } (n+1)\text{-many similar steps } \} \\
& L \cong \sum_{i=0}^{n} A^i + A^{n+1} \times L
\end{aligned}
$$

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:

$$A^\star \stackrel{\text{def}}{=} \bigcup_{i \geq 0} A^i \tag{3.14}$$

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

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

Because $\sum_{i=0}^{\infty} A^i$ is isomorphic to $\bigcup_{i=0}^{\infty} A^i$ (see exercise 2.26), we finally have:

$$L \quad \cong \quad A^\star$$

All in all, we have obtained $A^\star$ as a solution to equation (3.13). 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 (3.13) 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 7, 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 (3.13) 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 equality "$=$" in this equation should give place to set-theoretical isomorphism ("$\cong$"):

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

Which isomorphism $L \xleftarrow{in} 1 + A \times L$ do we expect to witness (3.13)? This will depend on which particular solution to (3.13) 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

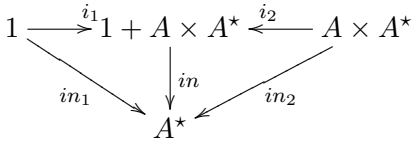
$$A^\star \xleftarrow{in} 1 + A \times A^\star$$

[4] By analogy, we may regard (3.12) as a "recursive definition" of number 2.

do we expect to witness the tautology which arises from (3.13) by replacing unknown L with solution $A^\star$, that is

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

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

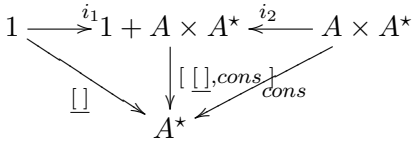


(3.16)

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 $in_2 = \underline{cons}$, where $cons$ is the standard "left append" sequence constructor, which we for the moment introduce rather informally as follows:

$$\begin{array}{l} cons:\ A \times A^\star \longrightarrow A^\star \\ cons(a, [a_1, \ldots, a_n]) = [a, a_1, \ldots, a_n] \end{array} \tag{3.17}$$

In a diagram:



(3.18)

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

$$out \quad \stackrel{\text{def}}{=} \quad (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?) \tag{3.19}$$

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

$$\begin{array}{l} hd:\ A^\star \longrightarrow A \\ hd\,[a_1, a_2, \ldots, a_n] = a_1 \end{array} \tag{3.20}$$

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

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

$$
\begin{array}{ll}
 & in \cdot out = id \\
\equiv & \quad \{ \text{ definitions of } in \text{ and } out \ \} \\
 & [\,\underline{[\,]}, cons\,] \cdot (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?) = id \\
\equiv & \quad \{ \ +\text{-absorption (2.41) and (2.15)} \ \} \\
 & [\,\underline{[\,]}, cons \cdot \langle hd, tl \rangle\,] \cdot (=_{[\,]}?) = id \\
\equiv & \quad \{ \text{ property of sequences: } cons(hd\, s, tl\, s) = s \ \} \\
 & [\,\underline{[\,]}, id\,] \cdot (=_{[\,]}?) = id \\
\equiv & \quad \{ \ \text{ going pointwise (2.61) } \} \\
 & \left\{ \begin{array}{lcl} =_{[\,]}\ a & \Rightarrow & [\,\underline{[\,]}, id\,]\,(i_1\, a) \\ \neg(=_{[\,]}\ a) & \Rightarrow & [\,\underline{[\,]}, id\,]\,(i_2\, a) \end{array} \right. = a \\
\equiv & \quad \{ \ +\text{-cancellation (2.38) } \} \\
 & \left\{ \begin{array}{lcl} =_{[\,]}\ a & \Rightarrow & \underline{[\,]}\, a \\ \neg(=_{[\,]}\ a) & \Rightarrow & id\, a \end{array} \right. = a \\
\equiv & \quad \{ \ a = [\,] \text{ in one case and identity function (2.9) in the other } \} \\
 & \left\{ \begin{array}{lcl} a = [\,] & \Rightarrow & a \\ \neg(a = [\,]) & \Rightarrow & a \end{array} \right. = a \\
\equiv & \quad \{ \ \text{property } (p \rightarrow f, f) = f \text{ holds } \} \\
 & a = a
\end{array}
$$

A comment on the particular choice of terminology above: symbol in suggests that we are going inside, or constructing (synthesizing) values of $A^\star$; symbol out suggests that we are going out, or destructing (analyzing) values of $A^\star$. We shall often resort to this duality in the sequel.

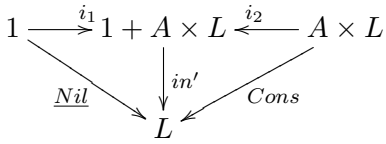
Are there more solutions to equation (3.15)? In trying to implement this equation, a HASKELL programmer could have written, after the declaration of type A, the following datatype declaration:

```
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)
```
(3.22)

and generates diagram



(3.23)

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 (3.15), 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 (3.22):

```
data T = Nil | Cons (A,T)
```
(3.24)

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 (3.24) 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

$$\begin{aligned} &out_{\mathsf{T}} : \mathsf{T} \longrightarrow 1 + A \times \mathsf{T} \\ &out_{\mathsf{T}}\, Nil = i_1\, \text{NIL} \\ &out_{\mathsf{T}}(Cons(a, l)) = i_2(a, l) \end{aligned} \tag{3.25}$$

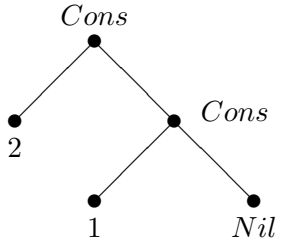
which can be straightforwardly encoded in HASKELL using the `Either` realization of $+$ (recall sections 2.9 and 2.18):

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

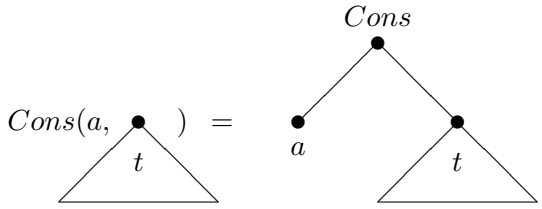
In summary, isomorphism

$$\mathsf{T} \underset{in_{\mathsf{T}}}{\overset{out_{\mathsf{T}}}{\cong}} 1 + A \times \mathsf{T} \tag{3.26}$$

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 (3.24). Therefore, an alternative declaration such as, for instance,

```
data U = Stop | Join (A,U)
```
(3.27)

would have been perfectly acceptable, generating another solution for the equation under algebra $[\ \underline{Stop}, Join\ ]$. It is easy to check that (3.27) 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 7 .

Exercise 3.5. *Obviously,*

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

does not work as a HASKELL *realization of the mediating arrow in diagram (3.18). 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 (3.24), 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 $\mathbb{N}_0$,

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

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 $\underline{0}$,

$$\mathbb{N}_0 \xleftarrow{[\ \underline{0}, add\ ]} 1 + \mathbb{N}_0 \times \mathbb{N}_0 \tag{3.28}$$

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

$$\mathsf{T} \xleftarrow{in_{\mathsf{T}}} 1 + \mathbb{N}_0 \times \mathsf{T} \tag{3.29}$$

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:

$$\begin{array}{ccc} \mathsf{T} & \xleftarrow{in_{\mathsf{T}}} & 1 + \mathbb{N}_0 \times \mathsf{T} \\ {\scriptstyle f}\downarrow & & \vdots \\ \mathbb{N}_0 & \xleftarrow[[\ \underline{0}, add\ ]]{} & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \end{array} \tag{3.30}$$

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

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

which pictures the following property of f

$$f \cdot [\ \underline{Nil}, Cons\ ] \ = \ [\ \underline{0}, add\ ] \cdot (id + id \times f) \tag{3.32}$$

and is easy to convert to pointwise notation:

$$
\begin{array}{ll}
& f \cdot [\ \underline{Nil}, Cons\ ] = [\ \underline{0}, add\ ] \cdot (id + id \times f) \\
\equiv & \quad \{\ \text{(2.40) on the lefthand side, (2.41) and identity } id \text{ on the righthand side}\ \} \\
& [\ f \cdot \underline{Nil}, f \cdot Cons\ ] = [\ \underline{0}, add \cdot (id \times f)\ ] \\
\equiv & \quad \{\ \textit{either}\ \text{structural equality (2.58)}\ \} \\
& \left\{ \begin{array}{l} f \cdot \underline{Nil} = \underline{0} \\ f \cdot Cons = add \cdot (id \times f) \end{array} \right. \\
\equiv & \quad \{\ \text{going pointwise}\ \} \\
& \left\{ \begin{array}{l} (f \cdot \underline{Nil})x = \underline{0}\, x \\ (f \cdot Cons)(a, x) = (add \cdot (id \times f))(a, x) \end{array} \right. \\
\equiv & \quad \{\ \text{composition (2.6), constant (2.12), product (2.22) and definition of } add\ \} \\
& \left\{ \begin{array}{l} f\, Nil = 0 \\ f(Cons(a, x)) = a + f\, x \end{array} \right.
\end{array}
$$

Note that we could have used out_{T} in diagram (3.30),

$$
\begin{array}{ccc}
\mathsf{T} & \xrightarrow{\quad out_{\mathsf{T}} \quad} & 1 + \mathbb{N}_0 \times \mathsf{T} \\
{\scriptstyle f}\downarrow & & \downarrow{\scriptstyle id + id \times f} \\
\mathbb{N}_0 & \xleftarrow[[\ \underline{0}, add\ ]]{} & 1 + \mathbb{N}_0 \times \mathbb{N}_0
\end{array}
\tag{3.33}
$$

obtaining another version of the *definition* of f,

$$
f \quad = \quad [\ \underline{0}, add\ ] \cdot (id + id \times f) \cdot out_{\mathsf{T}} \tag{3.34}
$$

which would lead to exactly the same pointwise recursive definition:

$$
\begin{array}{ll}
& f = [\ \underline{0}, add\ ] \cdot (id + id \times f) \cdot out_{\mathsf{T}} \\
\equiv & \quad \{\ \text{(2.41) and identity } id \text{ on the righthand side}\ \} \\
& f = [\ \underline{0}, add \cdot (id \times f)\ ] \cdot out_{\mathsf{T}} \\
\equiv & \quad \{\ \text{going pointwise on } out_{\mathsf{T}}\ \text{(3.25)}\ \} \\
& \left\{ \begin{array}{l} f\, Nil = ([\ \underline{0}, add \cdot (id \times f)\ ] \cdot out_{\mathsf{T}})Nil \\ f(Cons(a, x)) = ([\ \underline{0}, add \cdot (id \times f)\ ] \cdot out_{\mathsf{T}})(a, x) \end{array} \right. \\
\equiv & \quad \{\ \text{definition of } out_{\mathsf{T}}\ \text{(3.25)}\ \}
\end{array}
$$

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

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

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

$\equiv$ { 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 (3.19) — taking place of out_{T}, we get a "more algorithmic" version of f:

$$f = [\,\underline{0}, add\,] \cdot (id + id \times f) \cdot (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?)$$

$\equiv$ { $+$-functor (2.42), identity and $\times$-absorption (2.25) }

$$f = [\,\underline{0}, add\,] \cdot (! + \langle hd, f \cdot tl \rangle) \cdot (=_{[\,]}?)$$

$\equiv$ { $+$-absorption (2.41) and constant $\underline{0}$ }

$$f = [\,\underline{0}, add \cdot \langle hd, f \cdot tl \rangle\,] \cdot (=_{[\,]}?)$$

$\equiv$ { going pointwise on guard $=_{[\,]}?$ (2.61) and simplifying }

$$f\,l = \begin{cases} l = [\,] & \Rightarrow & \underline{0}\,l \\ \neg(l = [\,]) & \Rightarrow & (add \cdot \langle hd, f \cdot tl \rangle)\,l \end{cases}$$

$\equiv$ { simplification }

$$f\,l = \begin{cases} l = [\,] & \Rightarrow & 0 \\ \neg(l = [\,]) & \Rightarrow & hd\,l + f(tl\,l) \end{cases}$$

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

```
f l | l == []   = 0
| otherwise = head l + f (tail l)
```

or

```
f l = if l == []
then 0
else head l + f (tail l)
```

both requiring the equality predicate "`==`" 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 (3.24), 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 (3.33) 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,

$$f \quad = \quad in_{\mathsf{T}} \cdot (id + id \times f) \cdot g \tag{3.35}$$

which is equivalent to:

$$f \quad = \quad [\, \underline{Nil}, Cons \cdot (id \times f)\,] \cdot g \tag{3.36}$$

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,

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

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 $n > 0$,

$$pred\,n \stackrel{\text{def}}{=} n - 1 \tag{3.37}$$

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

$$g \quad = \quad (! + \langle id, pred \rangle) \cdot (=_0?) \tag{3.38}$$

Filling this in (3.36) we get

$$\begin{array}{ll} & f = [\ \underline{Nil}, Cons \cdot (id \times f)\ ] \cdot (! + \langle id, pred \rangle) \cdot (=_0?) \\ \equiv & \quad \{\ +\text{-absorption (2.41) followed by } \times\text{-absorption (2.25) } \textit{etc.}\ \} \\ & f = [\ \underline{Nil}, Cons \cdot \langle id, f \cdot pred \rangle\ ] \cdot (=_0?) \\ \equiv & \quad \{\ \text{going pointwise on guard } =_0? \text{ (2.61) and simplifying }\ \} \\ & f\,n = \left\{ \begin{array}{rcl} n = 0 & \Rightarrow & Nil \\ \neg(n = 0) & \Rightarrow & Cons(n, f\,(n-1)) \end{array} \right. \end{array}$$

which matches the function we had in mind:

```
f n | n == 0     = Nil
| otherwise = Cons(n,f(n-1))
```

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

3.6 Introducing (list) catas, anas and hylos

Suppose that, back to section 3.4, we want to *multiply*, rather than add, the elements occurring in lists of type T (3.24). 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 (3.33) 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,

$$\mathbb{N}_0 \xleftarrow{[\ \underline{1},mul\ ]} 1 + \mathbb{N}_0 \times \mathbb{N}_0 \tag{3.39}$$

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 3.8). Intuition will once again help us in deciding that only arrow g in (3.35) needs to be updated.

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

$$\begin{array}{ccc} \mathsf{T} & \xrightarrow{out_{\mathsf{T}}} & 1 + \mathbb{N}_0 \times \mathsf{T} \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle id+id\times f} \\ B & \xleftarrow[g]{} & 1 + \mathbb{N}_0 \times B \end{array} \qquad \begin{array}{ccc} \mathsf{T} & \xleftarrow{in_{\mathsf{T}}} & 1 + \mathbb{N}_0 \times \mathsf{T} \\ {\scriptstyle f}\uparrow & & \uparrow{\scriptstyle id+id\times f} \\ B & \xrightarrow[g]{} & 1 + \mathbb{N}_0 \times B \end{array} \tag{3.40}$$

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

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

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

[5] The theory which supports the statements of this paragraph will not be dealt with until chapter 7 .

which obeys to property (or is defined by)

$$(\!|g|\!) \quad = \quad g \cdot (id + id \times (\!|g|\!)) \cdot out_{\mathsf{T}} \tag{3.41}$$

which is the same as

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

- 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)

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

From (3.40) 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:

$$\begin{array}{ccc} B & \xleftarrow{\quad g \quad} & 1 + \mathbb{N}_0 \times B \\ {\scriptstyle (\!|g|\!)}\uparrow & & \uparrow{\scriptstyle id+id\times(\!|g|\!)} \\ \mathsf{T} & \xleftarrow{\quad in_{\mathsf{T}} \quad} & 1 + \mathbb{N}_0 \times \mathsf{T} \\ {\scriptstyle [\!(h)\!]}\uparrow & & \uparrow{\scriptstyle id+id\times[\!(h)\!]} \\ C & \xrightarrow[\quad h \quad]{} & 1 + \mathbb{N}_0 \times C \end{array} \tag{3.44}$$

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 (3.44)):

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

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:

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

As an example, let us see what comes out of $(\!|g|\!) \cdot [\!(h)\!]$ for h and g respectively given by (3.38) and (3.39):

$$\begin{array}{ll} & (\!|g|\!) \cdot [\!(h)\!] = g \cdot (id + id \times (\!|g|\!) \cdot [\!(h)\!]) \cdot h \\ \equiv & \quad \{\ (\!|g|\!) \cdot [\!(h)\!] \text{ abbreviated to } f \text{ and instantiating } h \text{ and } g\ \} \\ & f = [\ \underline{1}, mul\ ] \cdot (id + id \times f) \cdot (! + \langle id, pred \rangle) \cdot (=_0?) \\ \equiv & \quad \{\ +\text{-functor (2.42) and identity}\ \} \\ & f = [\ \underline{1}, mul\ ] \cdot (! + (id \times f) \cdot \langle id, pred \rangle) \cdot (=_0?) \\ \equiv & \quad \{\ \times\text{-absorption (2.25) and identity}\ \} \\ & f = [\ \underline{1}, mul\ ] \cdot (! + \langle id, f \cdot pred \rangle) \cdot (=_0?) \\ \equiv & \quad \{\ +\text{-absorption (2.41) and constant } \underline{1}\ (2.15)\ \} \\ & f = [\ \underline{1}, mul \cdot \langle id, f \cdot pred \rangle\ ] \cdot (=_0?) \\ \equiv & \quad \{\ \text{McCarthy conditional (2.60)}\ \} \\ & f = (=_0?) \to \underline{1}, mul \cdot \langle id, f \cdot pred \rangle \end{array}$$

Going pointwise, we get — via (2.60) —

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

and

$$\begin{array}{rcl} f(n+1) & = & [\ \underline{1}, mul \cdot \langle id, f \cdot pred \rangle\ ](i_2(n+1)) \\ & = & \quad \{\ +\text{-cancellation (2.38)}\ \} \end{array}$$

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

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 (3.44) for the particular g and h we have considered above and recall a popular "definition" of factorial:

$$
n! \quad = \quad \underbrace{n \times (n-1) \times \ldots \times 1}_{n \text{ times}} \tag{3.46}
$$

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

Exercise 3.6. *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*

$$sq\, n \stackrel{\text{def}}{=} n^2 \tag{3.47}$$

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

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

Exercise 3.7. *Write function* x^n *as a* T*-hylomorphism and encode it in* HASKELL.
□

Exercise 3.8. *The following function in* HASKELL *computes the* T*-sequence of all even numbers less or equal to* n*:*

```
f n = if n <= 1
      then Nil
      else Cons(m,f(m-2))
 where m = if even n then n else n-1
```

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 (3.29) — 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 $\mathsf{F}\,\mathsf{T}$

$$\mathsf{T} \underset{in_{\mathsf{T}}}{\overset{out_{\mathsf{T}}}{\cong}} \mathsf{F}\,\mathsf{T} \tag{3.48}$$

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

$$\mathsf{F}\,X \stackrel{\text{def}}{=} 1 + \mathbb{N}_0 \times X \tag{3.49}$$

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

$$X \quad \cong 1 + X \times \mathbb{N}_0 \tag{3.50}$$

— 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*:

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

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

$$X \quad \cong \quad \mathsf{F}\,X \tag{3.52}$$

(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

[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.

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

$$\mathsf{F}\,id_A \quad = \quad id_{(\mathsf{F}\,A)} \tag{3.53}$$

and with composition

$$\mathsf{F}(g \cdot h) \quad = \quad (\mathsf{F}\,g) \cdot (\mathsf{F}\,h) \tag{3.54}$$

Two simple examples of a functor follow:

- Identity functor: define $\mathsf{F}\,X = X$, for every datatype X, and $\mathsf{F}\,f = f$. Properties (3.53) and (3.54) hold trivially just by removing symbol F wherever it occurs.
- Constant functors: for a given C, define $\mathsf{F}\,X = C$ (for all datatypes X) and $\mathsf{F}\,f = id_C$, as expressed in the following diagram:

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

  Properties (3.53) and (3.54) 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 (3.53) and (3.54) have to hold for every parameter of an n-ary functor. For a binary functor B, for instance, equation (3.53) becomes

$$\mathsf{B}\,(id_A, id_B) \quad = \quad id_{\mathsf{B}\,(A,B)} \tag{3.55}$$

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.

and equation (3.54) becomes

$$\mathsf{B}\,(g \cdot h, i \cdot j) \quad = \quad \mathsf{B}\,(g, i) \cdot \mathsf{B}\,(h, j) \tag{3.56}$$

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 (3.55) and (3.56), 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 (2.77) and (2.78). 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

$$(\mathsf{F} \cdot \mathsf{G})X \quad \stackrel{\text{def}}{=} \quad \mathsf{F}\,(\mathsf{G}\,X) \tag{3.57}$$

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

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 (3.15). 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:

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

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

It should be clear at this point that what was referred to in section 2.10 as a "symbolic pattern" applicable to both datatypes and arrows is after all a functor in the mathematical sense. The fact that the same polynomial expression is used to denote both the data and the operators which structurally transform such data is of great conceptual economy and practical application. For instance, once polynomial functor (3.49) is assumed, the diagrams in (3.40) can be written as simply as

$$\begin{array}{ccc}
\mathsf{T} & \xrightarrow{out_{\mathsf{T}}} & \mathsf{F}\,\mathsf{T} \\
{\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,f} \\
B & \xleftarrow[g]{} & \mathsf{F}\,B
\end{array}
\qquad\qquad
\begin{array}{ccc}
\mathsf{T} & \xleftarrow{in_{\mathsf{T}}} & \mathsf{F}\,\mathsf{T} \\
{\scriptstyle f}\uparrow & & \uparrow{\scriptstyle \mathsf{F}\,f} \\
B & \xrightarrow[g]{} & \mathsf{F}\,B
\end{array}
\tag{3.59}$$

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,

$$\begin{array}{lll}
\mathsf{F}\,X & \cong & C_0 + (C_1 \times X) + (C_2 \times X^2) + \cdots + (C_n \times X^n) \\
& = & \sum_{i=0}^{n} C_i \times X^i
\end{array} \tag{3.60}$$

and that *Newton's binomial formula*

$$(A + B)^n \quad \cong \quad \sum_{p=0}^{n} {}^nC_p \times A^{n-p} \times B^p \tag{3.61}$$

can be used in such conversions. These are performed up to isomorphism, that is to say, after the conversion one gets a different but isomorphic datatype. Consider, for instance, functor

$$\mathsf{F}\,X \stackrel{\mathrm{def}}{=} A \times (1 + X)^2$$

(where A is a constant datatype) and check the following reasoning:

$$\begin{array}{lcl}
\mathsf{F}\,X & = & A \times (1+X)^2 \\
& \cong & \quad \{ \text{ law (2.91) } \} \\
& & A \times ((1+X) \times (1+X)) \\
& \cong & \quad \{ \text{ law (2.50) } \} \\
& & A \times ((1+X) \times 1 + (1+X) \times X)) \\
& \cong & \quad \{ \text{ laws (2.85), (2.31) and (2.50) } \} \\
& & A \times ((1+X) + (1 \times X + X \times X)) \\
& \cong & \quad \{ \text{ laws (2.85) and (2.91) } \} \\
& & A \times ((1+X) + (X + X^2)) \\
& \cong & \quad \{ \text{ law (2.46) } \} \\
& & A \times (1 + (X+X) + X^2) \\
& \cong & \quad \{ \text{ canonical form obtained via laws (2.50) and (2.92) } \} \\
& & \underbrace{A}_{C_0} + \underbrace{A \times 2}_{C_1} \times X + \underbrace{A}_{C_2} \times X^2
\end{array}$$

Exercise 3.9. *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 (3.52) is polynomial. For instance, datatype T (3.29) 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

$$\mathsf{T} \underset{in_{\mathsf{T}}}{\overset{\cong}{\leftarrow}} \sum_{i=0}^{n} C_i \times \mathsf{T}^i \tag{3.62}$$

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 (2.91) 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 7.

3.11 F-algebras and F-homomorphisms

Our interest in polynomial types is basically due to the fact that, for polynomial F, equation (3.52) 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

[8] A more traditional (but less close to (3.62)) encoding will be

```
data T = C0 | C1 C1 T | C2 C2 T T | ...  | Cn Cn T ...  T
```
(3.63)

delivering every constructor in curried form.

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 $[\ \underline{0}, add\ ]$ (3.28) and in_{T} (3.29), 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 function observing a particular F-algebra α into another F-algebra β:

$$\begin{array}{ccc} A & \xleftarrow{\alpha} & \mathsf{F}\,A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,f} \\ B & \xleftarrow[\beta]{} & \mathsf{F}\,B \end{array} \qquad f \cdot \alpha = \beta \cdot (\mathsf{F}\,f) \tag{3.64}$$

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 (3.53) — identity functions are always (trivial) F-homomorphisms and that — thanks to (3.54) — 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 (3.52). 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 $(\!|\alpha|\!)_{\mathsf{F}}$:

[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*.

[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 7 .

$$\begin{array}{ccc} \mu\mathsf{F} & \xleftarrow{in} & \mathsf{F}\,\mu\mathsf{F} \\ {\scriptstyle f=(\!|\alpha|\!)_{\mathsf{F}}}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,(\!|\alpha|\!)_{\mathsf{F}}} \\ A & \xleftarrow[\alpha]{} & \mathsf{F}\,A \end{array} \tag{3.65}$$

We will drop the F subscript in $(\!|\alpha|\!)_{\mathsf{F}}$ wherever deducible from the context, and often call α the "gene" of $(\!|\alpha|\!)_{\mathsf{F}}$.

As happens with *splits*, *eithers* and transposes, the uniqueness of the catamorphism construct is captured by a universal property established in the class of all F-homomorphisms:

$$k = (\!|\alpha|\!) \quad \Leftrightarrow \quad k \cdot in = \alpha \cdot \mathsf{F}\,k \tag{3.66}$$

According to the experience gathered from section 2.12 onwards, a few properties can be expected as consequences of (3.66). For instance, one may wonder about the "gene" of the identity catamorphism. Just let $k = id$ in (3.66) and see what happens:

$$\begin{array}{ll} & id = (\!|\alpha|\!) \Leftrightarrow id \cdot in = \alpha \cdot \mathsf{F}\,id \\ = & \quad \{ \text{ identity (2.10) and F is a functor (3.53) } \} \\ & id = (\!|\alpha|\!) \Leftrightarrow in = \alpha \cdot id \\ = & \quad \{ \text{ identity (2.10) once again } \} \\ & id = (\!|\alpha|\!) \Leftrightarrow in = \alpha \\ = & \quad \{ \ \alpha \text{ replaced by } in \text{ and simplifying } \} \\ & id = (\!|in|\!) \end{array}$$

Thus one finds out that the genetic material of the identity catamorphism is the initial algebra in. Which is the same as establishing the *reflection property* of catamorphisms:

Cata-reflection :

$$\begin{array}{ccc} \mu\mathsf{F} & \xleftarrow{in} & \mathsf{F}\,\mu\mathsf{F} \\ {\scriptstyle (\!|in|\!)}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,(\!|in|\!)} \\ \mu\mathsf{F} & \xleftarrow[in]{} & \mathsf{F}\,\mu\mathsf{F} \end{array} \qquad\qquad (\!|in|\!) = id_{\mu\mathsf{F}} \tag{3.67}$$

In a more intuitive way, one might have observed that $(\!|in|\!)$ is, by definition of in, the unique arrow mediating $\mu\mathsf{F}$ and itself. But another arrow of the same type is already known: the identity $id_{\mu\mathsf{F}}$. So these two arrows must be the same.

Another property following immediately from (3.66), for $k = (\!|\alpha|\!)$, is

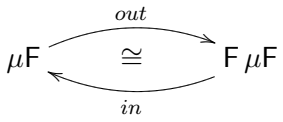
Cata-cancellation :

$$(\!|\alpha|\!) \cdot in = \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) \tag{3.68}$$

Because in is iso, this law can be rephrased as follows

$$(\!|\alpha|\!) = \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) \cdot out \tag{3.69}$$

where out denotes the inverse of in:



Now, let f be F-homomorphism (3.64) between F-algebras α and β. How does it relate to $(\!|\alpha|\!)$ and $(\!|\beta|\!)$? Note that $f \cdot (\!|\alpha|\!)$ is an arrow mediating $\mu\mathsf{F}$ and B. But B is the carrier of β and $(\!|\beta|\!)$ is the unique arrow mediating $\mu\mathsf{F}$ and B. So the two arrows are the same:

Cata-fusion :

$$\begin{array}{ccc} \mu\mathsf{F} & \xleftarrow{in} & \mathsf{F}\,\mu\mathsf{F} \\ {\scriptstyle (\!|\alpha|\!)}\downarrow & & \downarrow{\scriptstyle \mathsf{F}(\!|\alpha|\!)} \\ A & \xleftarrow[\alpha]{} & \mathsf{F}\,A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,f} \\ B & \xleftarrow[\beta]{} & \mathsf{F}\,B \end{array} \qquad f \cdot (\!|\alpha|\!) = (\!|\beta|\!) \quad \text{if} \quad f \cdot \alpha = \beta \cdot \mathsf{F}\,f \tag{3.70}$$

Of course, this law is also a consequence of the universal property, for $k = f \cdot (\!|\alpha|\!)$:

$$\begin{array}{lll} f \cdot (\!|\alpha|\!) = (\!|\beta|\!) & \Leftrightarrow & (f \cdot (\!|\alpha|\!)) \cdot in = \beta \cdot \mathsf{F}\,(f \cdot (\!|\alpha|\!)) \\ & \Leftrightarrow & \quad \{ \text{ composition is associative and } \mathsf{F} \text{ is a functor (3.54) } \} \\ & & f \cdot ((\!|\alpha|\!) \cdot in) = \beta \cdot (\mathsf{F}\,f) \cdot (\mathsf{F}\,(\!|\alpha|\!)) \\ & \Leftrightarrow & \quad \{ \text{ cata-cancellation (3.68) } \} \\ & & f \cdot \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) = \beta \cdot \mathsf{F}\,f \cdot \mathsf{F}\,(\!|\alpha|\!) \\ & \Leftarrow & \quad \{ \text{ require } f \text{ to be a } \mathsf{F}\text{-homomorphism (3.64) } \} \\ & & f \cdot \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) = f \cdot \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) \wedge f \cdot \alpha = \beta \cdot \mathsf{F}\,f \\ & \Leftrightarrow & \quad \{ \text{ simplify } \} \\ & & f \cdot \alpha = \beta \cdot \mathsf{F}\,f \end{array}$$

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 (3.29). It can be checked that

$$(\!|\, [\, \underline{Nil}, Cons \cdot (sq \times id)\,]\, |\!) \tag{3.71}$$

will do this for us, where $\mathbb{N}_0 \xleftarrow{sq} \mathbb{N}_0$ is given by (3.47). 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 (3.71). However, the output type no longer is T, but rather type $\mathsf{T}' \cong 1 + B \times \mathsf{T}'$.

Types T and T' are very close to each other. They share the same "shape" (recursive pattern) and only differ with respect to the type of elements — $\mathbb{N}_0$ in T and B in T'. This suggests that these two types can be regarded as instances of a more generic list datatype List

$$\mathsf{List}\, X \underset{in=[\, \underline{Nil}, Cons\,]}{\overset{\cong}{\leftarrow}} 1 + X \times \mathsf{List}\, X \tag{3.72}$$

in which the type of elements X is allowed to vary. Thus one has $\mathsf{T} = \mathsf{List}\, \mathbb{N}_0$ and $\mathsf{T}' = \mathsf{List}\, B$.

By inspection, it can be checked that, for every $B \xleftarrow{f} A$,

$$(\!|\, [\, \underline{Nil}, Cons \cdot (f \times id)\,]\, |\!) \tag{3.73}$$

maps $\mathsf{List}\, A$ to $\mathsf{List}\, B$. Moreover, for $f = id$ one has:

$$\begin{array}{ll} & (\!|\, [\, \underline{Nil}, Cons \cdot (id \times id)\,]\, |\!) \\ = & \quad \{ \text{ by the } \times\text{-functor-id property (2.29) and identity } \} \\ & (\!|\, [\, \underline{Nil}, Cons\,]\, |\!) \\ = & \quad \{ \text{ cata-reflection (3.67) } \} \\ & id \end{array}$$

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 (3.53) 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.* (3.54). 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 (3.49). Moreover, one has

$$\mathsf{F}\, f \;=\; \mathsf{B}\,(id, f) \tag{3.74}$$

and so every F-homomorphism can be written in terms of the base-functor of F, *e.g.*

$$f \cdot \alpha = \beta \cdot \mathsf{B}\,(id, f)$$

instead of (3.64).

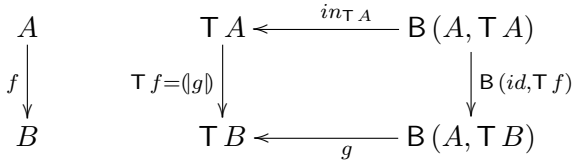
$\mathsf{T}\,X$ will be referred to as the *type functor* generated by B:

$$\underbrace{\mathsf{T}X}_{\text{type functor}} \quad \cong \quad \underbrace{\mathsf{B}(X, \mathsf{T}X)}_{\text{base functor}}$$

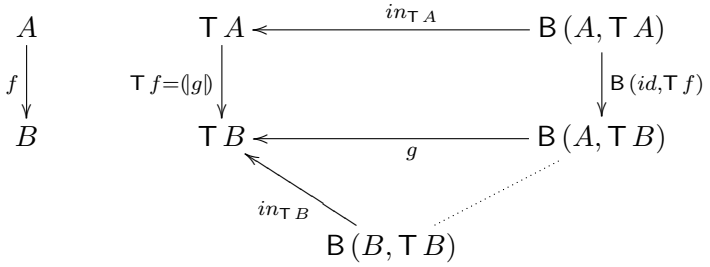
We proceed to the description of its functorial behaviour — $\mathsf{T}\,f$ — for a given $B \xleftarrow{f} A$. As far as typing rules are concerned, we shall have

$$\frac{B \xleftarrow{f} A}{\mathsf{T}\,B \xleftarrow{\mathsf{T}\,f} \mathsf{T}\,A}$$

So we should be able to express $\mathsf{T}\,f$ as a $\mathsf{B}\,(A, \_)$-catamorphism $(\!|g|\!)$:



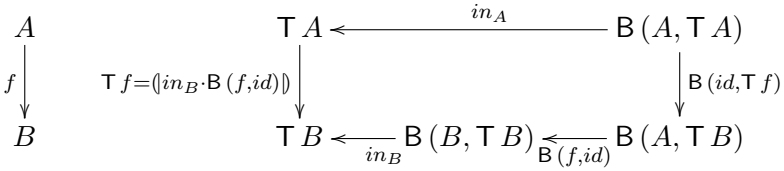
As we know that $in_{\mathsf{T}\,B}$ is the standard constructor of values of type $\mathsf{T}\,B$, we may put it into the diagram too:



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

$$\mathsf{T}\,f \quad \stackrel{\text{def}}{=} \quad (\!|in_{\mathsf{T}\,B} \cdot \mathsf{B}\,(f, id)|\!) \tag{3.75}$$

and a final diagram, where $in_{\mathsf{T}\,A}$ is abbreviated by in_A (ibid. $in_{\mathsf{T}\,B}$ by in_B):



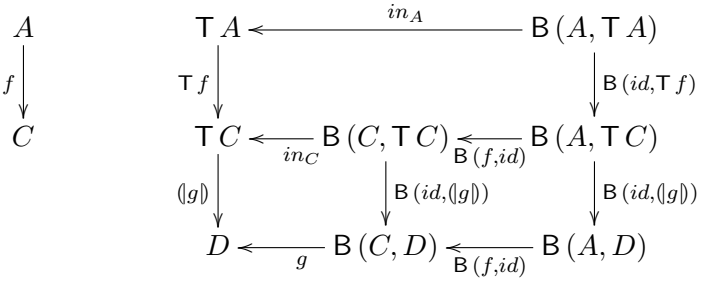
Next, we proceed to derive the useful law of *cata-absorption*

$$(\!|g|\!) \cdot \mathsf{T}\, f \quad = \quad (\!|g \cdot \mathsf{B}\,(f, id)|\!) \tag{3.76}$$

as consequence of the laws studied in section 3.12. Our target is to show that, for $k = (\!|g|\!) \cdot \mathsf{T}\, f$ in (3.66), one gets $\alpha = g \cdot \mathsf{B}\,(f, id)$:

$$\begin{array}{ll}
& (\!|g|\!) \cdot \mathsf{T}\, f = (\!|\alpha|\!) \\
\Leftrightarrow & \quad \{ \text{ type-functor definition (3.75) } \} \\
& (\!|g|\!) \cdot (\!|in_B \cdot \mathsf{B}\,(f, id)|\!) = (\!|\alpha|\!) \\
\Leftarrow & \quad \{ \text{ cata-fusion (3.70) } \} \\
& (\!|g|\!) \cdot in_B \cdot \mathsf{B}\,(f, id) = \alpha \cdot \mathsf{B}\,(id, (\!|g|\!)) \\
\Leftrightarrow & \quad \{ \text{ cata-cancellation (3.68) } \} \\
& g \cdot \mathsf{B}\,(id, (\!|g|\!)) \cdot \mathsf{B}\,(f, id) = \alpha \cdot \mathsf{B}\,(id, (\!|g|\!)) \\
\Leftrightarrow & \quad \{ \ \mathsf{B} \text{ is a bi-functor (3.56) } \} \\
& g \cdot \mathsf{B}\,(id \cdot f, (\!|g|\!) \cdot id) = \alpha \cdot \mathsf{B}\,(id, (\!|g|\!)) \\
\Leftrightarrow & \quad \{ \ id \text{ is natural (2.11) } \} \\
& g \cdot \mathsf{B}\,(f \cdot id, id \cdot (\!|g|\!)) = \alpha \cdot \mathsf{B}\,(id, (\!|g|\!)) \\
\Leftrightarrow & \quad \{ \text{ (3.56) again, this time from left to right } \} \\
& g \cdot \mathsf{B}\,(f, id) \cdot \mathsf{B}\,(id, (\!|g|\!)) = \alpha \cdot \mathsf{B}\,(id, (\!|g|\!)) \\
\Leftarrow & \quad \{ \text{ Leibniz } \} \\
& g \cdot \mathsf{B}\,(f, id) = \alpha
\end{array}$$

The following diagram pictures this property of catamorphisms:



It remains to show that (3.75) indeed defines a functor. This can be verified by checking properties (3.53) and (3.54) for $\mathsf{F} = \mathsf{T}$:

- Property **type-functor-id**, *cf.* (3.53):

$$
\begin{array}{ll}
 & \mathsf{T}\, id \\
= & \qquad \{ \text{ by definition (3.75) } \} \\
 & (\!|in_B \cdot \mathsf{B}\,(id, id)|\!) \\
= & \qquad \{ \ \mathsf{B} \text{ is a bi-functor (3.55) } \} \\
 & (\!|in_B \cdot id|\!) \\
= & \qquad \{ \text{ identity and cata-reflection (3.67) } \} \\
 & id
\end{array}
$$

- Property **type-functor**, *cf.* (3.54) :

$$
\begin{array}{ll}
 & \mathsf{T}\,(f \cdot g) \\
= & \qquad \{ \text{ by definition (3.75) } \} \\
 & (\!|in_B \cdot \mathsf{B}\,(f \cdot g, id)|\!) \\
= & \qquad \{ \ id \cdot id = id \text{ and } \mathsf{B} \text{ is a bi-functor (3.56) } \} \\
 & (\!|in_B \cdot \mathsf{B}\,(f, id) \cdot \mathsf{B}\,(g, id)|\!) \\
= & \qquad \{ \text{ cata-absorption (3.76) } \} \\
 & (\!|in_B \cdot \mathsf{B}\,(f, id)|\!) \cdot \mathsf{T}\, g \\
= & \qquad \{ \text{ again cata-absorption (3.76) } \} \\
 & (\!|in_B|\!) \cdot \mathsf{T}\, f \cdot \mathsf{T}\, g \\
= & \qquad \{ \text{ cata-reflection (3.67) followed by identity } \} \\
 & \mathsf{T}\, f \cdot \mathsf{T}\, g
\end{array}
$$

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 (3.62). The table contains two extra columns which may be used as bookmarks for equations (3.74) and (3.75), respec-

tively [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	$\mathsf{LTree}\,X$	$X+Y^2$	$id+f^2$	$f+id$

(3.77)

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 (3.77) 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	$\mathsf{FTree}(X_1,X_2)$	$X_1+X_2\times Y^2$	$id+id\times f^2$	$f+g\times id$

(3.78)

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 (2.91), see for instance

```
data BTree a = Empty | Node(a, (BTree a, BTree a))
```

Moreover, one may chose to curry the type constructors as in, *e.g.*

```
data BTree a = Empty | Node a (BTree a) (BTree a)
```

Exercise 3.10. *Write as a catamorphisms*

- *the function which counts the number of elements of a non-empty list (type* NList *in (3.77)).*

[12] Since $(id_A)^2 = id_{(A^2)}$ one writes id^2 for id in this table.

- *the function which computes the maximum element of a binary-tree of natural numbers*.

□

Exercise 3.11. *Characterize the function which is defined by* $(\!|[\ \underline{[]}, h\ ]|\!)$ *for each of the following definitions of* h*:*

$$h(x,(y_1,y_2)) = y_1 +\!\!+ [x] +\!\!+ y_2 \quad (3.79)$$

$$h = +\!\!+ \cdot (singl \times +\!\!+) \quad (3.80)$$

$$h = +\!\!+ \cdot (+\!\!+ \times singl) \cdot swap \quad (3.81)$$

assuming $singl\ a = [a]$*. Identify in (3.77) which datatypes are involved as base functors*.
□

Exercise 3.12. *Write as a catamorphism the function which computes the* frontier *of a tree of type* LTree *(3.77), listed from left to right*.
□

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:

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

So `fmap g` encodes 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

```
instance Functor [] where
    fmap f []     = []
    fmap f (x:xs) = f x : fmap f xs
```

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

```
instance Functor BTree
         where fmap f =
               cataBTree ( inBTree . (id -|- (f >< id)) )
```

concerning the binary-tree datatype of (3.77) 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

```
class BiFunctor f where
      bmap :: (a -> b) -> (c -> d) -> (f a c -> f b d)
```

Exercise 3.13. *Declare all datatypes in (3.77) in* HASKELL *notation and turn them into* HASKELL *type functors, that is, define* `fmap` *in each case*.
□

Exercise 3.14. *Declare datatype (3.78) 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*:

$$\begin{array}{ccc} \mathsf{T} & \xleftarrow{in} & \mathsf{F}\,\mathsf{T} \\ {\scriptstyle (\!|\langle h,k\rangle|\!)}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,(\!|\langle h,k\rangle|\!)} \\ A\times B & \xleftarrow[\langle h,k\rangle]{} & \mathsf{F}\,(A\times B) \end{array}$$

As we did for the exchange rule, we put (2.55) and the universal property of catamorphisms (3.66) against each other and calculate:

$$\begin{array}{ll} & \langle f,g\rangle = (\!|\langle h,k\rangle|\!) \\ \equiv & \quad \{\ \text{cata-universal (3.66)}\ \} \\ & \langle f,g\rangle \cdot in = \langle h,k\rangle \cdot \mathsf{F}\,\langle f,g\rangle \\ \equiv & \quad \{\ \times\text{-fusion (2.24) twice}\ \} \\ & \langle f\cdot in, g\cdot in\rangle = \langle h\cdot \mathsf{F}\,\langle f,g\rangle, k\cdot \mathsf{F}\,\langle f,g\rangle\rangle \\ \equiv & \quad \{\ (2.56)\ \} \\ & f\cdot in = h\cdot \mathsf{F}\,\langle f,g\rangle \quad \wedge \quad g\cdot in = k\cdot \mathsf{F}\,\langle f,g\rangle \end{array}$$

The rule thus obtained,

$$\left\{\begin{array}{l} f\cdot in = h\cdot \mathsf{F}\,\langle f,g\rangle \\ g\cdot in = k\cdot \mathsf{F}\,\langle f,g\rangle \end{array}\right. \quad \equiv \quad \langle f,g\rangle = (\!|\langle h,k\rangle|\!) \tag{3.82}$$

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

$$\begin{array}{ccc} \mathsf{T} & \xleftarrow{in} & \mathsf{F}\,\mathsf{T} \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,\langle f,g\rangle} \\ A & \xleftarrow[h]{} & \mathsf{F}\,(A\times B) \end{array} \qquad \begin{array}{ccc} \mathsf{T} & \xleftarrow{in} & \mathsf{F}\,\mathsf{T} \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,\langle f,g\rangle} \\ B & \xleftarrow[k]{} & \mathsf{F}\,(A\times B) \end{array}$$

into a single catamorphism.

When applied from left to right, law (3.82) is surprisingly useful in optimizing recursive functions in a way which saves redundant traversals of the input inductive type T. Let us take the Fibonacci function as example:

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

It can be shown that fib is a hylomorphism of type LTree (3.77), $fib = [\![count, fibd]\!]$, 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 togther,

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

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

$$\mathbb{N}_0 \underset{in = [\, \underline{0}, suc \,]}{\overset{\cong}{\rightleftarrows}} \underbrace{1 + \mathbb{N}_0}_{\mathsf{F}\,\mathbb{N}_0} \tag{3.83}$$

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:

$$\begin{aligned} f \cdot in &= [\, \underline{1}, add \,] \cdot \mathsf{F}\, \langle f, fib \rangle \\ fib \cdot in &= [\, \underline{1}, \pi_1 \,] \cdot \mathsf{F}\, \langle f, fib \rangle \end{aligned}$$

$$
\begin{array}{ll}
\equiv & \{\ (3.82)\ \} \\
& \langle f, fib\rangle \ = \ (\!|\langle [\ \underline{1}, add\ ], [\ \underline{1}, \pi_1\ ]\rangle|\!) \\
\equiv & \{\ \text{exchange law (2.47)}\ \} \\
& \langle f, fib\rangle \ = \ (\!|[\ \langle \underline{1}, \underline{1}\rangle, \langle add, \pi_1\rangle\ ]|\!) \\
\equiv & \{\ \text{going pointwise and denoting } \langle f, fib\rangle \text{ by } fib'\ \} \\
& \left\{ \begin{array}{l} fib'\ 0 = (1,1) \\ fib'\ (n+1) = (x+y, x)\ where\ (x,y) = fib'\ n \end{array} \right.
\end{array}
$$

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 = y where (x,y) = 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 (3.6) — 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" (3.4) 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 (3.83),

$$\begin{aligned} bnm\ 0 &= 1 \\ bnm(n+1) &= 2n + 3 + bnm\ n \end{aligned}$$

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 $lin\ n \stackrel{\text{def}}{=} 2^n + 3$ 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 3.17 below shows how to extend (3.82) to three mutually recursive functions (3.84). (From this it is easy to extend it further to the n-ary case.) It is routine work to show that, by application of (3.84) to the above three functions, one obtains the linear version of ns which follows:

```
ns'' n = let (a,b,c) = aux n in a
         where
           aux 0 = (0,1,3)
           aux(n+1) = let (a,b,c) = aux n
                      in (a+b,b+c,2+c)
```

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 (3.82). This only happens if all f_i have the same "shape", that is, if they share the same base functor F.

Exercise 3.15. *Use the mutual recursion law (3.82) 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 3.16. *The following Haskell function computes the list of the first n natural numbers in reverse order:*

```
insg 0 = []
insg(n+1) = (n+1):insg n
```

1. *Show that $insg$ can also be defined as follows:*

```
insg 0 = []
insg (n+1) = (fsuc n):insg n

fsuc 0 = 1
fsuc (n+1) = fsuc n + 1
```

2. *Based on the mutual recursion law derive from such a definition the following version of* $insg$ *encoded as a for-loop:*

$$\begin{aligned} insg &= \pi_2 \cdot insgfor \\ insgfor &= \mathsf{for}\ \langle(((1+)\cdot\pi_1), cons\rangle\ \underline{(1,[])} \end{aligned}$$

where $cons(n, m) = n : m$.

□

Exercise 3.17. *Show that law (3.82) generalizes to more than two mutually recursive functions, in this case three:*

$$\left\{\begin{array}{l} f\cdot in = h\cdot \mathsf{F}\,\langle f,\langle g,j\rangle\rangle \\ g\cdot in = k\cdot \mathsf{F}\,\langle f,\langle g,j\rangle\rangle \\ j\cdot in = l\cdot \mathsf{F}\,\langle f,\langle g,j\rangle\rangle \end{array}\right. \quad \equiv \quad \langle f,\langle g,j\rangle\rangle = (\!|\langle h,\langle k,l\rangle\rangle|\!) \tag{3.84}$$

□

Exercise 3.18. *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:*

$$e^x \;=\; \sum_{n=0}^{\infty} \frac{x^n}{n!} \tag{3.85}$$

The following function, in Haskell,

```
exp :: Double -> Integer -> Double
exp x 0      = 1
exp x (n+1) = x^(n+1) / fac (n+1)  + (exp x n)
```

computes an approximation of e^x*, where the second parameter tells how many terms to compute. For instance, while* $exp\ 1\ 1 = 2.0$*,* $exp\ 1\ 10$ *yields* 2.7182818011463845.

Function $exp\ x\ n$ *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 (3.84) and obtain the following linear version:*

```
exp x n = let (e,b,c) = aux x n
          in e where
               aux x 0 = (1,2,x)
               aux x (n+1) = let (e,s,h) = aux x n
                             in  (e+h,s+1,(x/s)*h)
```

□

Exercise 3.19. *Show that, for all* $n \in \mathbb{N}_0$, $n = suc^n 0$. ***Hint:*** *use cata-reflexion (3.67).*
□

Mutual recursion over lists. As example of application of (3.82) for T other than $\mathbb{N}_0$, consider the following recursive predicate which checks whether a (non-empty) list is ordered,

$$\begin{aligned} &ord: \ A^+ \longrightarrow 2 \\ &ord\,[a] = \text{TRUE} \\ &ord\,(cons(a,l)) = a \geq (listMax\,l) \wedge (ord\,l) \end{aligned}$$

where $\geq$ is assumed to be a total order on datatype A and

$$listMax = (\!|\,[\,id, max\,]\,|\!) \tag{3.86}$$

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 $\alpha(a,(m,b)) \stackrel{\text{def}}{=} a \geq m \wedge b$) 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:

$$\begin{array}{lcl} \langle listMax, ord\rangle & = & (\!|\langle [\ id, max\cdot(id\times\pi_1)\ ], [\ \underline{\text{TRUE}}, \alpha\ ]\rangle|\!) \\ & = & \quad \{\ \textit{exchange law}\ (2.47)\ \} \\ & & (\!|[\ \langle id, \underline{\text{TRUE}}\rangle, \langle max\cdot(id\times\pi_1), \alpha\rangle\ ]|\!) \end{array}$$

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

$$\begin{array}{l} aux:\ A^+ \longrightarrow A \times 2 \\ aux\,[a] = (a, \text{TRUE}) \\ aux\,(cons(a,l)) = \begin{array}{lll} let & (m,b) = aux\, l \\ in & (max(a,m), (a > m \wedge b)) \end{array} \end{array}$$

Exercise 3.20. *What do the following Haskell functions do?*

```
f1 [] = []
f1(h:t) = h:(f2 t)

f2 [] = []
f2(h:t) = f1 t
```

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 (3.82). Then

$$\begin{array}{ll} & f \cdot in = (i \cdot \mathsf{F}\,\pi_1) \cdot \mathsf{F}\,\langle f, g\rangle \\ \equiv & \quad \{ \text{ composition is associative and } \mathsf{F} \text{ is a functor } \} \\ & f \cdot in = i \cdot \mathsf{F}\,(\pi_1 \cdot \langle f, g\rangle) \\ \equiv & \quad \{ \text{ by } \times\text{-cancellation (2.20) } \} \\ & f \cdot in = i \cdot \mathsf{F}\,f \\ \equiv & \quad \{ \text{ by cata-cancellation } \} \\ & f = (\!|i|\!) \end{array}$$

Similarly, from $k = j \cdot \mathsf{F}\,\pi_2$ we get

$$g = (\!|j|\!)$$

Then, from (3.82), we get

$$\langle (\!|i|\!), (\!|j|\!)\rangle = (\!|\langle i \cdot \mathsf{F}\,\pi_1, j \cdot \mathsf{F}\,\pi_2\rangle|\!)$$

that is

$$\langle (\!|i|\!), (\!|j|\!)\rangle = (\!|(i \times j) \cdot \langle \mathsf{F}\,\pi_1, \mathsf{F}\,\pi_2\rangle|\!) \tag{3.87}$$

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,

$$average \stackrel{\text{def}}{=} (/) \cdot \langle sum, length\rangle \tag{3.88}$$

where sum and $length$ are the expected $\mathbb{N}^+$ catamorphisms:

$$\begin{array}{r} sum = (\!|[\, id, + \,]|\!) \\ length = (\!|[\, \underline{1}, succ \cdot \pi_2 \,]|\!) \end{array}$$

As defined by (3.88), 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":

$$\begin{array}{lll} average\, l & = & x/y \\ & & where \quad (x,y) = aux\, l \\ & & \quad\quad aux[a] = (a,1) \\ & & \quad\quad aux(cons(a,l)) = (a+x, y+1) \\ & & \quad\quad\quad\quad where \quad (x,y) = aux\, l \end{array} \tag{3.89}$$

Exercise 3.21. *Calculate (3.89) from (3.88). Which of these two versions of the same function is easier to understand?*
□

Exercise 3.22. *Show that the standard Haskell function*

```
unzip xs = (map fst xs, map snd xs)
```

*can be defined as a catamorphism (*fold*) thanks to (3.87). Generalize this calculation to the* generic *unzip function over an inductive (polynomial) type* T*:*

$$unzip_{\mathsf{T}} = \langle \mathsf{T}\pi_1, \mathsf{T}\pi_2 \rangle$$

Suggestion: recall (3.75).
□

3.18 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 [MA86], was popularized by [Mal90] and reached textbook format in [BdM97]. A comfortable basis for exploiting *polymorphism* [Wad89], the "datatypes as functors" moto has proved beneficial at a higher level of abstraction, giving birth to *polytypism* [JJ96].

The literature on *anas*, *catas* and *hylos* is vast (see eg. [MH95], [JJ98], [GHA01]) and it is part of a broader discipline which has become known as the *mathematics of program construction* [Bac04]. 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 5 and 7. The results on mutual recursion presented in this chapter were pionered by Maarten Fokkinga [Fok92].