Chapter 2

Recursion in the Pointfree Style

How useful from a programmer's point of view are the abstract concepts presented in the previous chapter? Recall that a table was presented — table 1.1 — which records an analogy between abstract type notation and the corresponding data-structures available in common, imperative languages.

This analogy is precisely our point of departure for extending the abstract notation towards a most important field of programming: *recursion*.

2.1 Motivation

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

```
L  =  ^N;
N  =  record
         P: A;
         S: ^N
      end;
```

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

```
        L x;
```

in the context of datatype definition

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

and so on.

What interests us in such "first year programming course" datatype declarations? Records and pointers have already been dealt with in table 1.1. So we can use this table to find the abstract version of datatype L, by replacing pointers by the "$1 + \cdots$" notation and records (*structs*) by the "$\ldots \times \ldots$" notation:

$$\left\{\begin{array}{lcl} L & = & 1+N \\ N & = & A\times(1+N) \end{array}\right. \tag{2.1}$$

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

$$\begin{array}{cl} & \left\{\begin{array}{lcl} L & = & 1+N \\ N & = & A\times(1+N) \end{array}\right. \\ \leftrightarrow & \quad \{ \text{ substituting } L \text{ for } 1+N \text{ in the second equation} \} \\ & \left\{\begin{array}{lcl} L & = & 1+N \\ N & = & A\times L \end{array}\right. \\ \leftrightarrow & \quad \{ \text{ substituting } A\times L \text{ for } N \text{ in the first equation} \} \\ & \left\{\begin{array}{lcl} L & = & 1+A\times L \\ N & = & A\times L \end{array}\right. \end{array}$$

System (2.1) is thus equivalent to:

$$\left\{\begin{array}{lcl} L & = & 1+A\times L \\ N & = & A\times(1+N) \end{array}\right. \tag{2.2}$$

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,

$$x = 1 + \frac{x}{2} \qquad\qquad ? \tag{2.3}$$

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

$$\begin{array}{cl} & x = 1+\dfrac{x}{2} \\ \leftrightarrow & \quad \{ \text{ adding } -\frac{x}{2} \text{ to both sides of the equation} \} \\ & x-\dfrac{x}{2} = 1+\dfrac{x}{2}-\dfrac{x}{2} \\ \leftrightarrow & \quad \{ -\frac{x}{2} \text{ cancels } \frac{x}{2} \} \\ & x-\dfrac{x}{2} = 1 \end{array}$$

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

$2 \times x - x = 2$

$\leftrightarrow$ $\{$ subtraction$\}$

$x = 2$

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

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

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

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

$\leftrightarrow$ $\{ +$ is associative$\}$

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

$\leftrightarrow$ $\{ -\frac{x}{2}$ is the additive inverse of $\frac{x}{2}\}$

$$x - \frac{x}{2} = 1 + 0$$

$\leftrightarrow$ $\{$ 0 is the unit of addition$\}$

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

$\leftrightarrow$ $\{$ multiplying both sides of the equation by 2$\}$

$$2 \times (x - \frac{x}{2}) = 2 \times 1$$

$\leftrightarrow$ $\{$ 1 is the unit of multiplication$\}$

$$2 \times (x - \frac{x}{2}) = 2$$

$\leftrightarrow$ $\{$ multiplication distributes over addition$\}$

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

$\leftrightarrow$ $\{$ 2 cancels its inverse $\frac{1}{2}$ $\}$

$$2 \times x - 1 \times x = 2$$

$\leftrightarrow$ $\{$ multiplication distributes over addition$\}$

$$(2 - 1) \times x = 2$$

$\leftrightarrow$ $\{ 2 - 1 = 1$ and 1 is the unit of multiplication$\}$

$$x = 2$$

Back to (2.2), we would like to submit each of the equations, *e.g.*

$$L \quad = \quad 1 + A \times L \tag{2.4}$$

to a similar reasoning. Can we do it? The analogy which can be found between this equation and (2.3) goes beyond pattern similarity. From chapter 1 we know that many properties required in the reasoning above hold in the context of (2.4), provided the "=" sign is replaced by the "$\cong$" sign, that of set-theoretical isomorphism. Recall that, for instance, $+$ is associative (1.46), 0 is the unit of addition (1.77), 1 is the unit of multiplication (1.79), multiplication distributes over addition (1.50) *etc.* Moreover, the first step above assumed that addition is compatible (monotonic) with respect to equality,

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

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

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

— recall (1.44) for isos f and g.

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

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

which hold about numbers but do not hold about datatypes. In fact, neither products nor coproducts have arbitrary inverses [1], and so we cannot "calculate by cancellation". How do we circumvent this limitation?

Just think of how we would have gone about (2.3) 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...

$$\begin{array}{ll} & x = 1 + \dfrac{x}{2} \\ \leftrightarrow & \quad \{ \, x \text{ by } 1 + \frac{x}{2} \text{ substitution followed by simplification} \} \\ & x = 1 + \dfrac{1 + \frac{x}{2}}{2} = 1 + \dfrac{1}{2} + \dfrac{x}{4} \\ \leftrightarrow & \quad \{ \text{ the same as above} \} \end{array}$$

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

$$
\begin{array}{ll}
& x = 1 + \frac{1}{2} + \frac{1+\frac{x}{2}}{4} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{x}{8} \\
\leftrightarrow & \quad \{ \text{ over and over again, } n\text{-times} \} \\
& \ldots \\
\leftrightarrow & \quad \{ \text{ simplification} \} \\
& x = \sum_{i=0}^{n} \frac{1}{2^i} + \frac{x}{2^{n+1}} \\
\leftrightarrow & \quad \{ \text{ sum of } n \text{ first terms of a geometric progression } \} \\
& x = (2 - \frac{1}{2^n}) + \frac{x}{2^{n+1}} \\
\leftrightarrow & \quad \{ \text{ let } n \to \infty \} \\
& x = (2 - 0) + 0 \\
\leftrightarrow & \quad \{ \text{ simplification } \} \\
& x = 2
\end{array}
$$

Clearly, this is a much more complicated way of finding solution $x = 2$ for equation (2.3). But we would have loved it in case it were the only known way, and this is precisely what happens with respect to (2.4). In this case we have:

$$
\begin{array}{ll}
& L = 1 + A \times L \\
\leftrightarrow & \quad \{ \text{ substitution of } 1 + A \times L \text{ for } L \} \\
& L = 1 + A \times (1 + A \times L) \\
\leftrightarrow & \quad \{ \text{ distributive property (1.50) } \} \\
& L \cong 1 + A \times 1 + A \times (A \times L) \\
\leftrightarrow & \quad \{ \text{ unit of product (1.79) and associativity of product (1.32)} \} \\
& L \cong 1 + A + (A \times A) \times L \\
\leftrightarrow & \quad \{ \text{ by (1.80), (1.82) and (1.85)} \} \\
& L \cong A^0 + A^1 + A^2 \times L \\
\leftrightarrow & \quad \{ \text{ another substitution as above and similar simplifications} \} \\
& L \cong A^0 + A^1 + A^2 + A^3 \times L \\
\leftrightarrow & \quad \{ \text{ after } (n+1)\text{-many similar steps} \} \\
& L \cong \sum_{i=0}^{n} A^i + A^{n+1} \times L
\end{array}
$$

Bearing a large n in mind, let us deliberately (but temporarily) ignore term $A^{n+1} \times$

L. Then L will be isomorphic to the sum of n-many contributions A^i,

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

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

$$A^\star \stackrel{\mathrm{def}}{=} \bigcup_{i \geq 0} A^i \tag{2.5}$$

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

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

Because $\sum_{i=0}^{\infty} A^i$ is isomorphic to $\bigcup_{i=0}^{\infty} A^i$ (see exercise 1.19), we finally have:

$$L \cong A^\star$$

All in all, we have obtained $A^\star$ as a solution to equation (2.4). 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 (2.4) 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 3, 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.

2.2 Introducing inductive datatypes

Datatype L as defined by (2.4) is said to be *recursive* because L "recurs" in the definition of L itself [2]. From the discussion above, it is clear that set-theoretical equality "$=$" in this equation should give place to set-theoretical isomorphism ("$\cong$"):

$$L \quad \cong \quad 1 + A \times L \tag{2.6}$$

Which isomorphism $L \xleftarrow{in} 1 + A \times L$ do we expect to witness (2.4)? This will depend on which particular solution to (2.4) we are thinking of. So far we have seen only one, $A^\star$. By recalling the notion of *algebra* of a datatype (section 1.18), so we may rephrase the question as: which algebra

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

do we expect to witness the tautology which arises from (2.4) by replacing unknown L with solution $A^\star$, that is

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

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

$$\begin{tikzcd}
1 \arrow[r, "i_1"] \arrow[dr, "in_1"'] & 1 + A \times A^\star \arrow[d, "in"] & A \times A^\star \arrow[l, "i_2"'] \arrow[dl, "in_2"] \\
& A^\star &
\end{tikzcd} \tag{2.7}$$

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

$$\begin{array}{l} cons : \ A \times A^\star \longrightarrow A^\star \\ cons(a, [a_1, \ldots, a_n]) = [a, a_1, \ldots, a_n] \end{array} \tag{2.8}$$

In a diagram:

$$\begin{tikzcd}
1 \arrow[r, "i_1"] \arrow[dr, "\underline{[\,]}"'] & 1 + A \times A^\star \arrow[d, "{[\, \underline{[\,]}, cons \,]}"] & A \times A^\star \arrow[l, "i_2"'] \arrow[dl, "cons"] \\
& A^\star &
\end{tikzcd} \tag{2.9}$$

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

$$out \quad \stackrel{\text{def}}{=} \quad (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?) \tag{2.10}$$

[2]By analogy, we may regard (2.3) as a "recursive definition" of number 2.

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

$$\begin{array}{l} hd : A^\star \longrightarrow A \\ hd\,[a_1, a_2, \ldots, a_n] = a_1 \end{array} \tag{2.11}$$

$$\begin{array}{l} tl : A^\star \longrightarrow A^\star \\ tl\,[a_1, a_2, \ldots, a_n] = [a_2, \ldots, a_n] \end{array} \tag{2.12}$$

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

$$\begin{array}{ll}
& in \cdot out = id \\
\leftrightarrow & \quad \{ \text{ definitions of } in \text{ and } out \} \\
& [\, \underline{[\,]}, cons \,] \cdot (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?) = id \\
\leftrightarrow & \quad \{ +\text{-absorption (1.41) and (1.15)} \} \\
& [\, \underline{[\,]}, cons \cdot \langle hd, tl \rangle \,] \cdot (=_{[\,]}?) = id \\
\leftrightarrow & \quad \{ \text{ property of sequences: } cons(hd\ s, tl\ s) = s \} \\
& [\, \underline{[\,]}, id \,] \cdot (=_{[\,]}?) = id \\
\leftrightarrow & \quad \{ \text{ going pointwise } \} \\
& \left\{ \begin{array}{rcl} =_{[\,]}\ a & \Rightarrow & [\, \underline{[\,]}, id \,]\,(i_1\, a) \\ \neg(=_{[\,]}\ a) & \Rightarrow & [\, \underline{[\,]}, id \,]\,(i_2\, a) \end{array} \right. = a \\
\leftrightarrow & \quad \{ +\text{-cancellation (1.38) } \} \\
& \left\{ \begin{array}{rcl} =_{[\,]}\ a & \Rightarrow & \underline{[\,]}\, a \\ \neg(=_{[\,]}\ a) & \Rightarrow & id\, a \end{array} \right. = a \\
\leftrightarrow & \quad \{ \ a = [\,] \text{ in one case and identity function (1.9) in the other } \} \\
& \left\{ \begin{array}{rcl} a = [\,] & \Rightarrow & a \\ \neg(a = [\,]) & \Rightarrow & a \end{array} \right. = a \\
\leftrightarrow & \quad \{ \text{ property } (p \rightarrow f, f) = f \text{ holds } \} \\
& a = a
\end{array}$$

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 (2.6)? 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 1.18, can be written simply as

$$\texttt{data L = Nil | Cons (A,L)} \tag{2.13}$$

and generates diagram

$$\begin{tikzcd} 1 \arrow[r, "i_1"] \arrow[dr, "\underline{Nil}"'] & 1 + A \times L \arrow[d, "in'"] & A \times L \arrow[l, "i_2"'] \arrow[dl, "Cons"] \\ & L & \end{tikzcd} \tag{2.14}$$

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 (2.6), 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 (2.13):

$$\texttt{data T = Nil | Cons (A,T)} \tag{2.15}$$

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 (2.15) so that we can "see" the inhabitants of the T datatype. The information associated to T is thus:

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

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

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

By typing `Nil`

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

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

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

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

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

or

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

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

$$\begin{array}{l} out_{\mathsf{T}} : \ \mathsf{T} \longrightarrow 1 + A \times \mathsf{T} \\ out_{\mathsf{T}}\, Nil = i_1\ \text{NIL} \\ out_{\mathsf{T}}(Cons(a,l)) = i_2(a,l) \end{array} \tag{2.16}$$

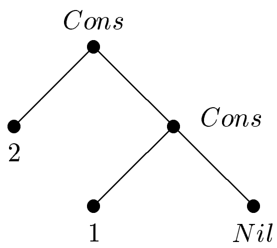
which can be straightforwardly encoded in HASKELL using the `Either` realization of $+$ (recall sections 1.9 and 1.18):

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

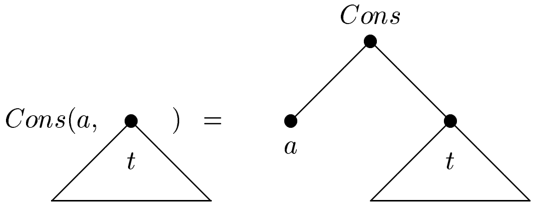
In summary, isomorphism

$$\begin{tikzcd} \mathsf{T} \arrow[rr, bend left, "out_{\mathsf{T}}"] \arrow[rr, phantom, "\cong"] & & 1 + A \times \mathsf{T} \arrow[ll, bend left, "in_{\mathsf{T}}"] \end{tikzcd} \tag{2.17}$$

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

```
data U = Stop | Join (A,U)
```
(2.18)

would have been perfectly acceptable, generating another solution for the equation under algebra $[\,Stop, Join\,]$. It is easy to check that (2.18) 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 3.

Exercise 2.1 Obviously,

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

does not work as a HASKELL realization of the mediating arrow in diagram (2.9). What do you need to write instead?
□

2.3 Observing an inductive datatype

Suppose that one is asked to express a particular *observation* of an inductive such as T (2.15), 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 guarantee that addition is available in $\mathbb{N}_0$,

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

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

$$\mathbb{N}_0 \xleftarrow{[\,\underline{0},add\,]} 1 + \mathbb{N}_0 \times \mathbb{N}_0 \tag{2.19}$$

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

$$\mathsf{T} \xleftarrow{in_{\mathsf{T}}} 1 + \mathbb{N}_0 \times \mathsf{T} \tag{2.20}$$

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

$$\begin{tikzcd}
\mathsf{T} \arrow[d, "f"'] & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[l, "in_{\mathsf{T}}"'] \arrow[d, dotted] \\
\mathbb{N}_0 & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \arrow[l, "{[\,\underline{0},add\,]}"]
\end{tikzcd} \tag{2.21}$$

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 1.10), where the "$\cdots$" are very naturally filled in by identity functions. All in all, we obtain diagram

$$\begin{tikzcd}
\mathsf{T} \arrow[d, "f"'] & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[l, "{[\,\underline{Nil},Cons\,]}"'] \arrow[d, "id + id \times f"] \\
\mathbb{N}_0 & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \arrow[l, "{[\,\underline{0},add\,]}"]
\end{tikzcd} \tag{2.22}$$

which pictures the following property of f

$$f \cdot [\,\underline{Nil}, Cons\,] \;=\; [\,\underline{0}, add\,] \cdot (id + id \times f) \tag{2.23}$$

and is easy to convert to pointwise notation:

$$
\begin{array}{ll}
& f \cdot [\,\underline{Nil}, Cons\,] = [\,\underline{0}, add\,] \cdot (id + id \times f) \\
\leftrightarrow & \quad \{\ (1.40) \text{ on the lefthand side, } (1.41) \text{ and identity } id \text{ on the righthand side}\ \} \\
& [\,f \cdot \underline{Nil}, f \cdot Cons\,] = [\,\underline{0}, add \cdot (id \times f)\,] \\
\leftrightarrow & \quad \{\ \textit{either} \text{ structural equality } (1.58)\ \}
\end{array}
$$

$$\left\{\begin{array}{l} f \cdot \underline{Nil} = \underline{0} \\ f \cdot Cons = add \cdot (id \times f) \end{array}\right.$$

$\leftrightarrow$ { going pointwise}

$$\left\{\begin{array}{l} (f \cdot \underline{Nil})x = \underline{0}\, x \\ (f \cdot Cons)(a, x) = (add \cdot (id \times f))(a, x) \end{array}\right.$$

$\leftrightarrow$ { composition (1.6), constant (1.12), product (1.22) and definition of add }

$$\left\{\begin{array}{l} f\, Nil = 0 \\ f(Cons(a, x)) = a + f\, x \end{array}\right.$$

Note that we could have used out_{T} in diagram (2.21),

$$\begin{tikzcd} \mathsf{T} \arrow[r, "out_{\mathsf{T}}"] \arrow[d, "f"'] & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[d, "id + id \times f"] \\ \mathbb{N}_0 & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \arrow[l, "{[\,\underline{0}, add\,]}"] \end{tikzcd} \tag{2.24}$$

obtaining another version of the *definition* of f,

$$f \quad = \quad [\,\underline{0}, add\,] \cdot (id + id \times f) \cdot out_{\mathsf{T}} \tag{2.25}$$

which would lead to exactly the same pointwise recursive definition:

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

$\leftrightarrow$ { (1.41) and identity id on the righthand side }

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

$\leftrightarrow$ { going pointwise on out_{T} (2.16) }

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

$\leftrightarrow$ { definition of out_{T} (2.16)}

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

$\leftrightarrow$ { $+$-cancellation (1.38) }

$$\left\{\begin{array}{l} f\, Nil = \underline{0}\, Nil \\ f(Cons(a, x)) = (add \cdot (id \times f))\,(a, x) \end{array}\right.$$

$\leftrightarrow$ { simplification }

$$\left\{\begin{array}{l} f\, Nil = 0 \\ f(Cons(a, x)) = a + f\, x \end{array}\right.$$

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

$$
\begin{array}{ll}
& f = [\,\underline{0}, add\,] \cdot (id + id \times f) \cdot (! + \langle hd, tl\rangle) \cdot (=_{[\,]}?) \\
\leftrightarrow & \quad \{\ +\text{-functor (1.42), identity and } \times\text{-absorption (1.25)}\ \} \\
& f = [\,\underline{0}, add\,] \cdot (! + \langle hd, f \cdot tl\rangle) \cdot (=_{[\,]}?) \\
\leftrightarrow & \quad \{\ +\text{-absorption (1.41) and constant } \underline{0}\ \} \\
& f = [\,\underline{0}, add \cdot \langle hd, f \cdot tl\rangle\,] \cdot (=_{[\,]}?) \\
\leftrightarrow & \quad \{\ \text{going pointwise on guard } =_{[\,]}?\ \text{(1.60) and simplifying}\ \} \\
& f\,l = \left\{ \begin{array}{lll} l = [\,] & \Rightarrow & \underline{0}\,l \\ \neg(l = [\,]) & \Rightarrow & (add \cdot \langle hd, f \cdot tl\rangle)\,l \end{array} \right. \\
\leftrightarrow & \quad \{\ \text{simplification}\ \} \\
& f\,l = \left\{ \begin{array}{lll} l = [\,] & \Rightarrow & 0 \\ \neg(l = [\,]) & \Rightarrow & hd\,l + f(tl\,l) \end{array} \right.
\end{array}
$$

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

```
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`".

2.4 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 (2.15), 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 (2.24) applicable to the new situation. In trying to "re-use" this diagram, it is immediate that arrow f should be reversed. Bearing duality in mind, we may feel tempted to reverse all arrows just to see what happens. Identity functions are their own inverses, and in_{T} takes the place of out_{T}:

$$
\begin{tikzcd}
\mathsf{T} & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[l, "in_{\mathsf{T}}"'] \\
\mathbb{N}_0 \arrow[u, "f"] \arrow[r, dotted] & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \arrow[u, "id + id \times f"']
\end{tikzcd}
$$

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

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

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

$$f \quad = \quad in_{\mathsf{T}} \cdot (id + id \times f) \cdot g \tag{2.26}$$

which is equivalent to:

$$f \quad = \quad [\, \underline{Nil}, Cons \cdot (id \times f)\,] \cdot g \tag{2.27}$$

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

$$
\begin{tikzcd}
\mathbb{N}_0 \arrow[r, "=_0?"] \arrow[rr, bend right, "g"'] & \mathbb{N}_0 + \mathbb{N}_0 \arrow[r, "! + h"] & 1 + \mathbb{N}_0 \times \mathbb{N}_0
\end{tikzcd}
$$

leaving h to fill in. This will be a *split* providing, on the lefthand side, for the value to be $Cons$'ed to the output and, on the righthand side, for the "seed" to the next recursive call. Since we want the output values to be produced contiguously and in decreasing order, we may define $h = \langle id, \mathit{pred} \rangle$ where, for $n > 0$,

$$\mathit{pred}\; n \stackrel{\text{def}}{=} n - 1 \tag{2.28}$$

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

$$g \quad = \quad (! + \langle id, \mathit{pred} \rangle) \cdot (=_0?) \tag{2.29}$$

Filling this in (2.27) we get

$$
\begin{array}{ll}
& f = [\, \underline{Nil}, Cons \cdot (id \times f)\,] \cdot (! + \langle id, \mathit{pred} \rangle) \cdot (=_0?) \\
\leftrightarrow & \quad \{\ +\text{-absorption (1.41) followed by } \times\text{-absorption (1.25) } \textit{etc.} \} \\
& f = [\, \underline{Nil}, Cons \cdot \langle id, f \cdot \mathit{pred} \rangle\,] \cdot (=_0?) \\
\leftrightarrow & \quad \{\text{ going pointwise on guard } =_0? \text{ (1.60) and simplifying } \} \\
& f\,n = \left\{ \begin{array}{lcl} n = 0 & \Rightarrow & Nil \\ \neg(n = 0) & \Rightarrow & Cons(n, f\,(n-1)) \end{array} \right.
\end{array}
$$

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.

2.5 Introducing (list) catas, anas and hylos

Suppose that, back to section 2.3, we want to *multiply*, rather than add, the elements occurring in lists of type T (2.15). 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 (2.24) 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,

$$\mathbb{N}_0 \xleftarrow{[\,\underline{1},mul\,]} 1 + \mathbb{N}_0 \times \mathbb{N}_0 \tag{2.30}$$

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

Moreover, in the previous section we might have wanted our number-list generator to produce the list of even numbers smaller than a given number, in decreasing order (see exercise 2.4). Intuition will once again help us in deciding that only arrow g in (2.26) needs to be updated.

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

$$\begin{tikzcd}
\mathsf{T} \arrow[r, "out_{\mathsf{T}}"] \arrow[d, "f"'] & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[d, "id+id\times f"] \\
B & 1 + \mathbb{N}_0 \times B \arrow[l, "g"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{T} & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[l, "in_{\mathsf{T}}"'] \\
B \arrow[u, "f"] \arrow[r, "g"'] & 1 + \mathbb{N}_0 \times B \arrow[u, "id+id\times f"']
\end{tikzcd} \tag{2.31}$$

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 [3].

Following the standard terminology, we express these facts by writing $f = (\!|g|\!)$ with respect to the lefthand side diagram and by writing $f = [\!(g)\!]$ with respect to the righthand side diagram. Read $(\!|g|\!)$ as "the T-*catamorphism* induced by g" and $[\!(g)\!]$ as

[3] The theory which supports the statements of this paragraph will not be dealt with until chapter 3.

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

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

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

$$(\!|g|\!) \quad = \quad g \cdot (id + id \times (\!|g|\!)) \cdot out_{\mathsf{T}} \tag{2.32}$$

  which is the same as

$$(\!|g|\!) \cdot in_{\mathsf{T}} \quad = \quad g \cdot (id + id \times (\!|g|\!)) \tag{2.33}$$

- 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)

$$[\!(g)\!] \quad = \quad in_{\mathsf{T}} \cdot (id + id \times [\!(g)\!]) \cdot g \tag{2.34}$$

From (2.31) 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:

$$\begin{tikzcd}
B & & 1 + \mathbb{N}_0 \times B \arrow[ll, "g"'] \\
\mathsf{T} \arrow[u, "(\!|g|\!)"] & & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[ll, "in_{\mathsf{T}}"'] \arrow[u, "id+id\times(\!|g|\!)"'] \\
C \arrow[u, "[\!(h)\!]"] \arrow[rr, "h"'] & & 1 + \mathbb{N}_0 \times C \arrow[u, "id+id\times[\!(h)\!]"']
\end{tikzcd} \tag{2.35}$$

What can we say about the $(\!|g|\!) \cdot [\!(h)\!]$ composition? It is a function from B to C which resorts to T as an *intermediate* data-structure and can be subject to the following calculation (*cf.* outermost rectangle in (2.35)):

$$(\!|g|\!) \cdot [\!(h)\!] = g \cdot (id + id \times (\!|g|\!)) \cdot (id + id \times [\!(h)\!]) \cdot h$$

$$\leftrightarrow \qquad \{ \text{ +-functor (1.42) } \}$$

$$
\begin{array}{ll}
& (\!|g|\!) \cdot [\!(h)\!] = g \cdot ((id \cdot id) + (id \times (\!|g|\!)) \cdot (id \times [\!(h)\!])) \cdot h \\
\leftrightarrow & \quad \{ \text{ identity and } \times\text{-functor (1.28) } \} \\
& (\!|g|\!) \cdot [\!(h)\!] = g \cdot (id + id \times (\!|g|\!) \cdot [\!(h)\!]) \cdot h
\end{array}
$$

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:

$$
\begin{tikzcd}
B & 1 + \mathbb{N}_0 \times B \arrow[l, "g"'] \\
C \arrow[u, "(\!|g|\!) \cdot [\!(h)\!]"] \arrow[r, "h"'] & 1 + \mathbb{N}_0 \times C \arrow[u, "id + id \times (\!|g|\!) \cdot [\!(h)\!]"']
\end{tikzcd}
\tag{2.36}
$$

As an example, let us see what comes out of $(\!|g|\!) \cdot [\!(h)\!]$ for h and g respectively given by (2.29) and (2.30):

$$
\begin{array}{ll}
& (\!|g|\!) \cdot [\!(h)\!] = g \cdot (id + id \times (\!|g|\!) \cdot [\!(h)\!]) \cdot h \\
\leftrightarrow & \quad \{ \; (\!|g|\!) \cdot [\!(h)\!] \text{ abbreviated to } f \text{ and instantiating } h \text{ and } g \; \} \\
& f = [\, \underline{1}, mul \,] \cdot (id + id \times f) \cdot (! + \langle id, pred \rangle) \cdot (=_0?) \\
\leftrightarrow & \quad \{ \; +\text{-functor (1.42) and identity } \} \\
& f = [\, \underline{1}, mul \,] \cdot (! + (id \times f) \cdot \langle id, pred \rangle) \cdot (=_0?) \\
\leftrightarrow & \quad \{ \; \times\text{-absorption (1.25) and identity } \} \\
& f = [\, \underline{1}, mul \,] \cdot (! + \langle id, f \cdot pred \rangle) \cdot (=_0?) \\
\leftrightarrow & \quad \{ \; +\text{-absorption (1.41) and constant } \underline{1} \text{ (1.15) } \} \\
& f = [\, \underline{1}, mul \cdot \langle id, f \cdot pred \rangle \,] \cdot (=_0?) \\
\leftrightarrow & \quad \{ \; \text{McCarthy conditional (1.59) } \} \\
& f = (=_0?) \rightarrow \underline{1}, mul \cdot \langle id, f \cdot pred \rangle
\end{array}
$$

Going pointwise, we get

$$
\begin{array}{lll}
f\,0 & = & [\, \underline{1}, mul \cdot \langle id, f \cdot pred \rangle \,](i_1\, 0) \\
& = & \quad \{ \; +\text{-cancellation (1.38) } \} \\
& & \underline{1}\, 0 \\
& = & \quad \{ \; \text{constant function (1.12) } \} \\
& & 1
\end{array}
$$

and

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

$$
\begin{array}{ll}
= & \quad \{ \text{ +-cancellation (1.38) } \} \\
& mul \cdot \langle id, f \cdot pred \rangle (n+1) \\
= & \quad \{ \text{ pointwise definitions of } \mathit{split}\text{, identity, predecessor and } mul \text{ } \} \\
& (n+1) \times f\, n
\end{array}
$$

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

$$
\left\{ \begin{array}{l} f\,0 = 1 \\ f(n+1) = (n+1) \times f\,n \end{array} \right.
$$

This result comes to no surprise if we look at diagram (2.35) for the particular g and h we have considered above and recall a popular "definition" of factorial:

$$
n! \quad = \quad \underbrace{n \times (n-1) \times \ldots \times 1}_{n \text{ times}} \tag{2.37}
$$

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 (2.37) 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* [4] 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 2.2 A way of computing n^2, the square of a given natural number n, is to sum up the n first odd numbers. In fact, $1^2 = 1$, $2^2 = 1 + 3$, $3^2 = 1 + 3 + 5$, *etc.*, $n^2 = (2n - 1) + (n - 1)^2$. Following this hint, express function

$$
sq\, n \stackrel{\mathrm{def}}{=} n^2 \tag{2.38}
$$

[4] 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 2.3 Write function x^n as a T-hylomorphism and encode it in HASKELL.
□

Exercise 2.4 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

$$\begin{tikzcd}
\mathsf{T} & 1 + \mathbb{N}_0 \times \mathsf{T} \arrow[l, "in_{\mathsf{T}}"'] \\
\mathbb{N}_0 \arrow[u, "{[\![(g)]\!]}"] \arrow[r, "g"'] & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \arrow[u, "{id+id\times[\![(g)]\!]}"']
\end{tikzcd}$$

□

2.6 Inductive types more generally

So far we have focussed our attention exclusively to a particular inductive type T (2.20) — that of finite sequences of non-negative integers. This is, of course, of a very limited scope. First, because one could think of finite sequences of other datatypes, *e.g.* Booleans or many others. Second, because other datatypes such as trees, hash-tables *etc.* exist which our notation and method should be able to take into account.

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

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

$$\begin{tikzcd}
\mathsf{T} \arrow[r, bend left, "out_{\mathsf{T}}"] \arrow[r, phantom, "\cong"] & \mathsf{F}\,\mathsf{T} \arrow[l, bend left, "in_{\mathsf{T}}"]
\end{tikzcd} \qquad (2.39)$$

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

$$\mathsf{F}\,X \stackrel{\mathrm{def}}{=} 1 + \mathbb{N}_0 \times X \qquad (2.40)$$

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

$$X \quad \cong 1 + X \times \mathbb{N}_0 \qquad (2.41)$$

— 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*:

$$X \quad \cong 1 + X \times \mathbb{N}_0 \times X \qquad (2.42)$$

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

$$X \quad \cong \quad \mathsf{F}\,X \qquad (2.43)$$

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

2.7 Functors

The concept of a functor F, borrowed from category theory, is a most generic and useful device in programming [5]. As we have seen, F can be regarded as a datatype constructor which, given datatype A, builds a more elaborate datatype $\mathsf{F}\,A$; given another datatype B, builds a similarly elaborate datatype $\mathsf{F}\,B$; and so on. But what is more important and has the most beneficial consequences is that, if F is regarded as a functor, then its data-structuring effect extends smoothly to functions in the following way: suppose that $B \xleftarrow{f} A$ is a function which observes A into B, which are parameters of $\mathsf{F}\,A$ and $\mathsf{F}\,B$, respectively. By definition, if F is a functor then $\mathsf{F}\,B \xleftarrow{\mathsf{F}\,f} \mathsf{F}\,A$ exists for every such f:

$$\begin{tikzcd} A \arrow[r, dotted, no head] \arrow[d, "f"'] & \mathsf{F}\,A \arrow[d, "\mathsf{F}\,f"] \\ B \arrow[r, dotted, no head] & \mathsf{F}\,B \end{tikzcd}$$

[5] 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.

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

$$\mathsf{F}\, id_A = id_{(\mathsf{F}\, A)} \tag{2.44}$$

and with composition

$$\mathsf{F}(g \cdot h) = (\mathsf{F}\, g) \cdot (\mathsf{F}\, h) \tag{2.45}$$

Two simple examples of a functor follow:

- Identity functor: define $\mathsf{F}\, X = X$, for every datatype X, and $\mathsf{F}\, f = f$. Properties (2.44) and (2.45) hold trivially just by removing symbol F wherever it occurs.
- Constant functors: for a given C, define $\mathsf{F}\, X = C$ (for all datatypes X) and $\mathsf{F}\, f = id_C$, as expressed in the following diagram:

$$\begin{tikzcd} A \arrow[r, dotted, no head] \arrow[d, "f"'] & C \arrow[d, "id_C"] \\ B \arrow[r, dotted, no head] & C \end{tikzcd}$$

  Properties (2.44) and (2.45) 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 (2.44) and (2.45) have to hold for every parameter of an n-ary functor. For a binary functor B, for instance, equation (2.44) becomes

$$\mathsf{B}\,(id_A, id_B) = id_{\mathsf{B}\,(A,B)} \tag{2.46}$$

and equation (2.45) becomes

$$\mathsf{B}\,(g \cdot h, i \cdot j) = \mathsf{B}\,(g, i) \cdot \mathsf{B}\,(h, j) \tag{2.47}$$

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 (1.22). Properties (1.29) and (1.28) instantiate (2.46) and (2.47), 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 (1.37) — and functorial properties (1.43) and (1.42). Finally, exponentiation is a functorial construction too: assuming A, one has $\mathsf{F}\, X \stackrel{\mathrm{def}}{=} X^A$ and $\mathsf{F}\, f \stackrel{\mathrm{def}}{=} \overline{f \cdot ap}$ and functorial properties (1.71) and (1.72). All this is summarized in table 2.1.

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

$$(\mathsf{F} \cdot \mathsf{G})X \stackrel{\mathrm{def}}{=} \mathsf{F}\,(\mathsf{G}\, X) \tag{2.48}$$

$$(\mathsf{F} \cdot \mathsf{G})f \stackrel{\mathrm{def}}{=} \mathsf{F}\,(\mathsf{G}\, f) \tag{2.49}$$

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

Table 2.1: Datatype constructions and associated operators.

2.8 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 (2.6). 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:

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

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

It should be clear at this point that what was referred to in section 1.10 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 (2.40) is assumed, the diagrams in (2.31) can be written as simply as

$$\begin{tikzcd} \mathsf{T} \arrow[r, "out_{\mathsf{T}}"] \arrow[d, "f"'] & \mathsf{F}\,\mathsf{T} \arrow[d, "\mathsf{F}\,f"] \\ B & \mathsf{F}\,B \arrow[l, "g"] \end{tikzcd} \qquad \begin{tikzcd} \mathsf{T} & \mathsf{F}\,\mathsf{T} \arrow[l, "in_{\mathsf{T}}"'] \\ B \arrow[u, "f"] \arrow[r, "g"'] & \mathsf{F}\,B \arrow[u, "\mathsf{F}\,f"'] \end{tikzcd} \tag{2.50}$$

It is useful to know that, thanks to the isomorphism laws studied in chapter 1, every polynomial functor F may be put into the canonical form,

$$\begin{array}{rcl} \mathsf{F}\,X & \cong & C_0 + (C_1 \times X) + (C_2 \times X^2) + \cdots + (C_n \times X^n) \\ & = & \sum_{i=0}^{n} C_i \times X^i \end{array} \tag{2.51}$$

and that *Newton's binomial formula*

$$(A+B)^n \cong \sum_{p=0}^{n} {}^nC_p \times A^{n-p} \times B^p \tag{2.52}$$

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

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

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

$$\begin{array}{cl} \mathsf{F}\,X = & A \times (1+X)^2 \\ \cong & \quad \{ \text{ law (1.85) } \} \\ & A \times ((1+X) \times (1+X)) \\ \cong & \quad \{ \text{ law (1.50) } \} \\ & A \times ((1+X) \times 1 + (1+X) \times X)) \\ \cong & \quad \{ \text{ laws (1.79), (1.31) and (1.50) } \} \\ & A \times ((1+X) + (1 \times X + X \times X)) \\ \cong & \quad \{ \text{ laws (1.79) and (1.85) } \} \\ & A \times ((1+X) + (X + X^2)) \\ \cong & \quad \{ \text{ law (1.46) } \} \\ & A \times (1 + (X+X) + X^2) \\ \cong & \quad \{ \text{ canonical form obtained via laws (1.50) and (1.86) } \} \\ & \underbrace{A}_{C_0} + \underbrace{A \times 2}_{C_1} \times X + \underbrace{A}_{C_2} \times X^2 \end{array}$$

Exercise 2.5 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.
□

2.9 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 (2.43) is polynomial. For instance, datatype T (2.20) is polynomial ($n = 1$) and its associated polynomial functor is canonically defined with coefficients $C_0 = 1$ and $C_1 = \mathbb{N}_0$. For reasons that will become apparent later on, we shall always impose $C_0 \neq 0$ to hold in a *polynomial* datatype expressed in canonical form.

Polynomial types are easy to encode in HASKELL wherever the associated functor is in canonical polynomial form, that is, wherever one has

$$\begin{tikzcd} \mathsf{T} \arrow[r, phantom, "\cong"] & \sum_{i=0}^{n} C_i \times \mathsf{T}^i \arrow[l, bend left, "in_{\mathsf{T}}"] \end{tikzcd} \tag{2.53}$$

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 $\mathsf{T} \longleftarrow C_i \times \mathsf{T}^i$. Since n is finite, one may expand exponentials according to (1.85) 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 [6]. Several instances of polynomial inductive types (in canonical form) will be mentioned in section 2.13. Section 2.15 will address the conversion between inductive datatypes induced by so-called *natural transformations*.

The concepts of catamorphism, anamorphism and hylomorphism introduced in section 2.5 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 3.

[6] A more traditional (but less close to (2.53)) encoding will be

```
data T = C0 | C1 C1 T | C2 C2 T T | ... | Cn Cn T ... T
```
(2.54)

delivering every constructor in curried form.

2.10 F-algebras and F-homomorphisms

Our interest in polynomial types is basically due to the fact that, for polynomial F, equation (2.43) always has a particularly interesting solution which corresponds to our notion of a recursive datatype.

In order to explain this, we need two notions which are easy to understand: first, that of an F-*algebra*, which simply is any function α of signature $A \xleftarrow{\alpha} \mathsf{F}\,A$. A is called the *carrier* of F-algebra α and contains the values which α manipulates by computing new A-values out of existing ones, according to the F-pattern (the "type" of the algebra). As examples, consider $[\,\underline{0}, add\,]$ (2.19) and in_{T} (2.20), 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 [7].

Secondly, we introduce the notion of an F-*homomorphism* which is but a function observing a particular F-algebra α into another F-algebra β:

$$
\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{F}\,A \arrow[l, "\alpha"'] \arrow[d, "\mathsf{F}\,f"] \\
B & \mathsf{F}\,B \arrow[l, "\beta"]
\end{tikzcd}
\qquad f \cdot \alpha = \beta \cdot (\mathsf{F}\,f) \tag{2.55}
$$

Clearly, f can be regarded as a structural translation between A and B, that is, A and B have a similar structure [8]. Note that — thanks to (2.44) — identity functions are always (trivial) F-homomorphisms and that — thanks to (2.45) — these homomorphisms compose, that is, the composition of two F-homomorphisms is an F-homomorphism.

2.11 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 (2.43). 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" [9]. 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 2.5, is in general denoted by $(\!|\alpha|\!)_{\mathsf{F}}$:

[7] 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.*

[8] Cf. *homomorphism* = *homo* (the same) + *morphos* (structure, shape).

[9] $\mu\mathsf{F}$ means the least fixpoint solution of equation $X \cong \mathsf{F}\,X$, as will be described in chapter 3.

$$\begin{tikzcd} \mu\mathsf{F} \arrow[d, "f=(\!|\alpha|\!)_{\mathsf{F}}"'] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "in"'] \arrow[d, "\mathsf{F}\,(\!|\alpha|\!)_{\mathsf{F}}"] \\ A & \mathsf{F}\,A \arrow[l, "\alpha"] \end{tikzcd} \tag{2.56}$$

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:

$$k = (\!|\alpha|\!) \quad \Leftrightarrow \quad k \cdot in = \alpha \cdot \mathsf{F}\,k \tag{2.57}$$

According to the experience gathered from section 1.12 onwards, a few properties can be expected as consequences of (2.57). For instance, one may wonder about the "gene" of the identity catamorphism. Just let $k = id$ in (2.57) and see what happens:

$$\begin{array}{cl} & id = (\!|\alpha|\!) \Leftrightarrow id \cdot in = \alpha \cdot \mathsf{F}\,id \\ = & \quad \{ \text{ identity (1.10) and } \mathsf{F} \text{ is a functor (2.44) } \} \\ & id = (\!|\alpha|\!) \Leftrightarrow in = \alpha \cdot id \\ = & \quad \{ \text{ identity (1.10) once again } \} \\ & id = (\!|\alpha|\!) \Leftrightarrow in = \alpha \\ = & \quad \{ \ \alpha \text{ replaced by } in \text{ and simplifying } \} \\ & id = (\!|in|\!) \end{array}$$

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

Cata-reflection :

$$\begin{tikzcd} \mu\mathsf{F} \arrow[d, "(\!|in|\!)"'] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "in"'] \arrow[d, "\mathsf{F}\,(\!|in|\!)"] \\ \mu\mathsf{F} & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "in"] \end{tikzcd} \qquad (\!|in|\!) = id_{\mu\mathsf{F}} \tag{2.58}$$

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 (2.57), for $k = (\!|\alpha|\!)$, is

Cata-cancellation :

$$(\!|\alpha|\!) \cdot in = \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) \tag{2.59}$$

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

$$(\!|\alpha|\!) = \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) \cdot out \tag{2.60}$$

where out denotes the inverse of in:

$$\begin{tikzcd} \mu\mathsf{F} \arrow[r, bend left, "out"] \arrow[r, phantom, "\cong"] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, bend left, "in"] \end{tikzcd}$$

Now, let f be F-homomorphism (2.55) between F-algebras α and β. How does it relate to $(\!|\alpha|\!)$ and $(\!|\beta|\!)$? Note that $f \cdot (\!|\alpha|\!)$ is an arrow mediating $\mu\mathsf{F}$ and B. But B is the carrier of β and $(\!|\beta|\!)$ is the unique arrow mediating $\mu\mathsf{F}$ and B. So the two arrows are the same:

Cata-fusion :

$$\begin{tikzcd} \mu\mathsf{F} \arrow[d, "(\!|\alpha|\!)"'] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "in"'] \arrow[d, "\mathsf{F}(\!|\alpha|\!)"] \\ A \arrow[d, "f"'] & \mathsf{F}\,A \arrow[l, "\alpha"] \arrow[d, "\mathsf{F}\,f"] \\ B & \mathsf{F}\,B \arrow[l, "\beta"] \end{tikzcd}$$

$$f \cdot (\!|\alpha|\!) = (\!|\beta|\!) \quad \text{if} \quad f \cdot \alpha = \beta \cdot \mathsf{F}\,f \tag{2.61}$$

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

$$\begin{array}{lcl}
f \cdot (\!|\alpha|\!) = (\!|\beta|\!) & \Leftrightarrow & (f \cdot (\!|\alpha|\!)) \cdot in = \beta \cdot \mathsf{F}\,(f \cdot (\!|\alpha|\!)) \\
& \Leftrightarrow & \quad \{ \text{ composition is associative and } \mathsf{F} \text{ is a functor (2.45) } \} \\
& & f \cdot (\!|\alpha|\!) \cdot in = \beta \cdot \mathsf{F}\,f \cdot \mathsf{F}\,(\!|\alpha|\!) \\
& \Leftrightarrow & \quad \{ \text{ cata-cancellation (2.59) } \} \\
& & f \cdot \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) = \beta \cdot \mathsf{F}\,f \cdot \mathsf{F}\,(\!|\alpha|\!) \\
& \Leftrightarrow & \quad \{ \text{ require } f \text{ to be a } \mathsf{F}\text{-homomorphism (2.55) } \} \\
& & f \cdot \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) = f \cdot \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) \wedge f \cdot \alpha = \beta \cdot \mathsf{F}\,f \\
& \Leftarrow & \quad \{ \text{ simplify } \} \\
& & f \cdot \alpha = \beta \cdot \mathsf{F}\,f
\end{array}$$

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.

2.12 Parameterization, type functors and cata-absorption

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 (2.20). It can be checked that

$$(\![\ \underline{Nil}, Cons \cdot (sq \times id)\ ]\!) \tag{2.62}$$

will do this for us, where $\mathbb{N}_0 \xleftarrow{sq} \mathbb{N}_0$ is given by (2.38). This catamorphism, which converted to pointwise notation is nothing but function h which follows

$$\left\{\begin{array}{l} h\,Nil = Nil \\ h(Cons(a,l)) = Cons(sq\,a, h\,l) \end{array}\right.$$

maps type T to itself. This is because *sq* maps $\mathbb{N}_0$ to $\mathbb{N}_0$. Now suppose that, instead of *sq*, one would like to apply a given function $B \xleftarrow{f} \mathbb{N}_0$ (for some B other than $\mathbb{N}_0$) to all elements of the argument list. It is easy to see that it suffices to replace f for *sq* in (2.62). However, the output type no longer is T, but rather type $\mathsf{T}' \cong 1 + B \times \mathsf{T}'$.

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

$$\begin{tikzcd} \mathsf{List}\,X \arrow[r, phantom, "\cong"] & 1 + X \times \mathsf{List}\,X \arrow[l, bend left, "in=[\ \underline{Nil}, Cons\ ]"] \end{tikzcd} \tag{2.63}$$

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

It can be seen by inspection that, for any $B \xleftarrow{f} A$,

$$(\![\ \underline{Nil}, Cons \cdot (f \times id)\ ]\!) \tag{2.64}$$

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

$$\begin{array}{ll} & (\![\ \underline{Nil}, Cons \cdot (id \times id)\ ]\!) \\ = & \quad \{\ \text{by the } \times\text{-functor-id property (1.29) and identity}\ \} \\ & (\![\ \underline{Nil}, Cons\ ]\!) \\ = & \quad \{\ \text{cata-reflection (2.58)}\ \} \\ & id \end{array}$$

Therefore, by defining

$$\mathsf{List}\,f \stackrel{\mathrm{def}}{=} (\![\ \underline{Nil}, Cons \cdot (f \times id)\ ]\!)$$

what we have just seen can be written thus:

$$\mathsf{List}\, id_A \quad = \quad id_{\mathsf{List}\, A}$$

This is nothing but law (2.44) for F replaced by List. Moreover, it will not be too difficult to check that

$$\mathsf{List}\,(g \cdot f) \quad = \quad \mathsf{List}\, g \cdot \mathsf{List}\, f$$

also holds — *cf.* (2.45). 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 its parameter instantiations.

All this can be broadly generalized and leads to what is commonly known by a *type functor*. First of all, it should be clear that the generic format

$$\mathsf{T} \quad \cong \quad \mathsf{F}\,\mathsf{T}$$

adopted so far for the definition of an inductive type is not sufficiently detailed because it does not provide a parametric view of T. For simplicity, let us suppose that only one parameter is identified in T. Then we may factor this out via *type variable* X and write (overloading symbol T)

$$\mathsf{T}\, X \quad \cong \quad \mathsf{B}(X, \mathsf{T}\, X)$$

where B is called the type's *base functor*. Binary functor $\mathsf{B}(X, Y)$ is given this name because it is the basis of the whole inductive type definition. By instantiation of X one obtains F. In the example above, $\mathsf{B}\,(X, Y) = 1 + X \times Y$ and in fact $\mathsf{F}\, Y = \mathsf{B}\,(\mathbb{N}_0, Y) = 1 + \mathbb{N}_0 \times Y$, recall (2.40). Moreover, one has

$$\mathsf{F}\, f \quad = \quad \mathsf{B}\,(id, f) \tag{2.65}$$

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 (2.55).

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

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

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

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

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

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{T}\,A \arrow[d, "\mathsf{T}\,f = (\!|g|\!)"'] & & \mathsf{B}\,(A, \mathsf{T}\,A) \arrow[ll, "in_{\mathsf{T}\,A}"'] \arrow[d, "\mathsf{B}\,(id, \mathsf{T}\,f)"] \\
B & \mathsf{T}\,B & & \mathsf{B}\,(A, \mathsf{T}\,B) \arrow[ll, "g"]
\end{tikzcd}$$

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

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{T}\,A \arrow[d, "\mathsf{T}\,f = (\!|g|\!)"'] & & \mathsf{B}\,(A, \mathsf{T}\,A) \arrow[ll, "in_{\mathsf{T}\,A}"'] \arrow[d, "\mathsf{B}\,(id, \mathsf{T}\,f)"] \\
B & \mathsf{T}\,B & & \mathsf{B}\,(A, \mathsf{T}\,B) \arrow[ll, "g"] \arrow[dl, dotted, no head] \\
& & \mathsf{B}\,(B, \mathsf{T}\,B) \arrow[ul, "in_{\mathsf{T}\,B}"] &
\end{tikzcd}$$

The catamorphism's gene g will be synthesized by filling the dashed arrow in the diagram with the obvious $\mathsf{B}\,(f, id)$. Thus one gets

$$\mathsf{T}\,f \quad \stackrel{\mathrm{def}}{=} \quad (\!|in_{\mathsf{T}\,B} \cdot \mathsf{B}\,(f, id)|\!) \tag{2.66}$$

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

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{T}\,A \arrow[d, "\mathsf{T}\,f = (\!|in_B \cdot \mathsf{B}\,(f, id)|\!)"'] & & \mathsf{B}\,(A, \mathsf{T}\,A) \arrow[ll, "in_A"'] \arrow[d, "\mathsf{B}\,(id, \mathsf{T}\,f)"] \\
B & \mathsf{T}\,B & \mathsf{B}\,(B, \mathsf{T}\,B) \arrow[l, "in_B"] & \mathsf{B}\,(A, \mathsf{T}\,B) \arrow[l, "\mathsf{B}\,(f, id)"]
\end{tikzcd}$$

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

$$(\!|g|\!) \cdot \mathsf{T}\,f \quad = \quad (\!|g \cdot \mathsf{B}\,(f, id)|\!) \tag{2.67}$$

as a consequence of the laws studied in section 2.11. Our target is to show that, for $k = (\!|g|\!) \cdot \mathsf{T}\,f$ in (2.57), one gets $\alpha = g \cdot \mathsf{B}\,(f, id)$:

$$\begin{array}{ll}
& (\!|g|\!) \cdot \mathsf{T}\,f = (\!|\alpha|\!) \\
\Leftrightarrow & \qquad \{ \text{ type-functor definition (2.66) } \} \\
& (\!|g|\!) \cdot (\!|in_B \cdot \mathsf{B}\,(f, id)|\!) = (\!|\alpha|\!) \\
\Leftarrow & \qquad \{ \text{ cata-fusion (2.61) } \} \\
& (\!|g|\!) \cdot in_B \cdot \mathsf{B}\,(f, id) = \alpha \cdot \mathsf{B}\,(id, (\!|g|\!))
\end{array}$$

$$
\begin{array}{ll}
\Leftrightarrow & \{ \text{ cata-cancellation (2.59) } \} \\
& g \cdot \mathsf{B}\,(id, \llparenthesis g \rrparenthesis) \cdot \mathsf{B}\,(f, id) = \alpha \cdot \mathsf{B}\,(id, \llparenthesis g \rrparenthesis) \\
\Leftrightarrow & \{ \text{ } \mathsf{B} \text{ is a bi-functor (2.47) } \} \\
& g \cdot \mathsf{B}\,(id \cdot f, \llparenthesis g \rrparenthesis \cdot id) = \alpha \cdot \mathsf{B}\,(id, \llparenthesis g \rrparenthesis) \\
\Leftrightarrow & \{ \text{ } id \text{ is natural (1.11) } \} \\
& g \cdot \mathsf{B}\,(f \cdot id, id \cdot \llparenthesis g \rrparenthesis) = \alpha \cdot \mathsf{B}\,(id, \llparenthesis g \rrparenthesis) \\
\Leftrightarrow & \{ \text{ (2.47) again, this time from left to right } \} \\
& g \cdot \mathsf{B}\,(f, id) \cdot \mathsf{B}\,(id, \llparenthesis g \rrparenthesis) = \alpha \cdot \mathsf{B}\,(id, \llparenthesis g \rrparenthesis) \\
\Leftarrow & \{ \text{ obvious } \} \\
& g \cdot \mathsf{B}\,(f, id) = \alpha
\end{array}
$$

The following diagram pictures this property of catamorphisms:

$$
\begin{tikzcd}
A \arrow[d, "f"'] & & \mathsf{T}\,A & & \mathsf{B}\,(A, \mathsf{T}\,A) \arrow[ll, "in_A"'] \arrow[d, "\mathsf{B}\,(id, \mathsf{T}\,f)"] \\
B & & \mathsf{T}\,B \arrow[from=u, "\mathsf{T}\,f"'] \arrow[d, "\llparenthesis g \rrparenthesis"'] & \mathsf{B}\,(B, \mathsf{T}\,B) \arrow[l, "in_B"] \arrow[d, "\mathsf{B}\,(B, \llparenthesis g \rrparenthesis)"] & \mathsf{B}\,(A, \mathsf{T}\,B) \arrow[l, "\mathsf{B}\,(f, id)"] \arrow[d, "\mathsf{B}\,(A, \llparenthesis g \rrparenthesis)"] \\
& & C & \mathsf{B}\,(B, C) \arrow[l, "g"] & \mathsf{B}\,(A, C) \arrow[l, "\mathsf{B}\,(f, id)"]
\end{tikzcd}
$$

It remains to show that (2.66) indeed defines a functor. This can be verified by checking properties (2.44) and (2.45) for $\mathsf{F} = \mathsf{T}$:

- Property **type-functor-id**, *cf.* (2.44):

$$
\begin{array}{ll}
& \mathsf{T}\,id \\
= & \{ \text{ by definition (2.66) } \} \\
& \llparenthesis in_B \cdot \mathsf{B}\,(id, id) \rrparenthesis \\
= & \{ \text{ } \mathsf{B} \text{ is a bi-functor (2.46) } \} \\
& \llparenthesis in_B \cdot id \rrparenthesis \\
= & \{ \text{ identity and cata-reflection (2.58) } \} \\
& id
\end{array}
$$

- Property **type-functor**, *cf.* (2.45) :

$$
\begin{array}{ll}
& \mathsf{T}\,(f \cdot g) \\
= & \{ \text{ by definition (2.66) } \}
\end{array}
$$

$$
\begin{array}{cl}
 & (\!|in_B \cdot \mathsf{B}\,(f \cdot g, id)|\!) \\
= & \quad \{ \text{ identities and } \mathsf{B} \text{ is a bi-functor (2.47) } \} \\
 & (\!|in_B \cdot \mathsf{B}\,(f, id) \cdot \mathsf{B}\,(g, id)|\!) \\
= & \quad \{ \text{ cata-absorption (2.67) } \} \\
 & (\!|in_B \cdot \mathsf{B}\,(f, id)|\!) \cdot \mathsf{T}\,g \\
= & \quad \{ \text{ again cata-absorption (2.67) } \} \\
 & (\!|in_B|\!) \cdot \mathsf{T}\,f \cdot \mathsf{T}\,g \\
= & \quad \{ \text{ cata-reflection (2.58) followed by identity } \} \\
 & \mathsf{T}\,f \cdot \mathsf{T}\,g
\end{array}
$$

2.13 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 (2.53). The table contains two extra columns which may be used as bookmarks for equations (2.69) and (2.66), respectively [10]:

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

(2.68)

All type functors T in this table are unary. In general, one may think of inductive datatypes which exhibit more than one parameter. Should n parameters be identified in T, then this will be based on an $n+1$-ary base functor B, *cf.*

$$\mathsf{T}(X_1, \ldots, X_n) \quad \cong \quad \mathsf{B}(X_1, \ldots, X_n, \mathsf{T}(X_1, \ldots, X_n))$$

So, every $n+1$-ary polynomial functor $\mathsf{B}(X_1, \ldots, X_n, X_{n+1})$ can be identified as the basis of an inductive n-ary type functor (the convention is to stick to the canonical form and reserve the last variable X_{n+1} for the "recursive call"). While type bi-functors ($n = 2$) are often found in programming, the situation in which $n > 2$ is relatively rare. For instance, the combination of leaf-trees with binary-trees in (2.68) 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	$\mathsf{FTree}(X_1, X_2)$	$X_1 + X_2 \times Y^2$	$id + id \times f^2$	$f + g \times id$

(2.69)

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

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 (1.85), 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 2.6 Write as a catamorphism the function which counts the number of elements of a non-empty list (type NList in (2.68)).
□

Exercise 2.7 Write the function which computes the maximum element of a binary-tree of natural numbers as a catamorphism.
□

Exercise 2.8 Characterize the function which is defined by $(\![\,[\,\underline{Nil}, h\,]\,]\!)$ for each of the following definitions of h:

$$h(x,(y_1,y_2)) = y_1 +\!\!+ [x] +\!\!+ y_2 \tag{2.70}$$
$$h = +\!\!+ \cdot (singl \times +\!\!+) \tag{2.71}$$
$$h = +\!\!+ \cdot (+\!\!+ \times singl) \cdot swap \tag{2.72}$$

assuming $singl\ a = [a]$. What datatype in (2.68) are we talking about?
□

Exercise 2.9 Write as a catamorphism the function which computes the *frontier* of a tree of type LTree (2.68), listed from left to right.
□

2.14 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 `map` operator:

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

So `map g` encodes $\mathsf{F}\,g$ once we declare F as an instance of class `Functor`. The most popular use of `map` has to do with HASKELL lists and this is allowed by declaration

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

in the HUGS *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, if we write

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

concerning the binary-tree datatype of (2.68) and assuming appropriate declarations of `cataBTree` and `inBTree`, then `map` 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 2.10 Declare all datatypes in (2.68) in HASKELL notation and turn them into HASKELL type functors, that is, define `map` in each case.
□

Exercise 2.11 Declare datatype (2.69) in HASKELL notation and turn it into an instance of class `BiFunctor`.

□

2.15 Inductive datatype conversion and isomorphism

The $\mathsf{T}\,f$ "map" operation is a special case of a transformation between two inductive datatypes (in which the pattern of recursion remains unchanged). In a more general setting, suppose one is given two inductive datatypes T and U defined by functors F and G, respectively:

$$\begin{tikzcd} \mathsf{T} \arrow[r, phantom, "\cong"] & \mathsf{F}\,\mathsf{T} \arrow[l, bend left, "in_{\mathsf{T}}"] \end{tikzcd}$$

and

$$\begin{tikzcd} \mathsf{U} \arrow[r, phantom, "\cong"] & \mathsf{G}\,\mathsf{U} \arrow[l, bend left, "in_{\mathsf{U}}"] \end{tikzcd}$$

Moreover suppose that recursion pattern G can be converted to recursion pattern F via polymorphic map $\mathsf{F}\,X \xleftarrow{\nu_X} \mathsf{G}\,X$. It can be checked that

$$(\!|in_{\mathsf{T}} \cdot \nu_{\mathsf{T}}|\!)_{\mathsf{G}} \tag{2.73}$$