Chapter 2

Pointfree Recursive Programming

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 the abstract 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

```
        N *x;
```

in the context of datatype definition

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

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,

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

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

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

$$
\begin{array}{ll}
\leftrightarrow & \{ \text{ multiplying both sides of the equation by } 2 \textit{ etc.} \ \} \\
& 2 \times x - x = 2 \\
\leftrightarrow & \{ \text{ subtraction} \} \\
& x = 2
\end{array}
$$

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 $a + (-b)$:

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

We would like to submit each of our 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 (1.45), 0 is the unit of addition (1.71), 1 is the unit of multiplication (1.73), multiplication distributes over addition (1.49) *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 (1.43) for isos f and g.

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

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

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 would we go about (2.3) in case we didn't know anything about the *cancellation properties*? Well, 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...

$$\begin{array}{ll} & x = 1 + \dfrac{x}{2} \\ \leftrightarrow & \qquad \{ \text{ substitution followed by simplification} \} \\ & x = 1 + \dfrac{1 + \frac{x}{2}}{2} = 1 + \dfrac{1}{2} + \dfrac{x}{4} \\ \leftrightarrow & \qquad \{ \text{ the same as above} \} \\ & x = 1 + \dfrac{1 + \frac{1+\frac{x}{2}}{2}}{2} = 1 + \dfrac{1}{2} + \dfrac{1}{4} + \dfrac{x}{8} \end{array}$$

[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.

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

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:

$$
\begin{aligned}
& L = 1 + A \times L \\
\leftrightarrow \quad & \{ \text{ substitution of } 1 + A \times L \text{ for } L\} \\
& L = 1 + A \times (1 + A \times L) \\
\leftrightarrow \quad & \{ \text{ distributive property (1.49) } \} \\
& L \cong 1 + A \times 1 + A \times (A \times L) \\
\leftrightarrow \quad & \{ \text{ by (1.73) and (1.31)}\} \\
& L \cong 1 + A + (A \times A) \times L \\
\leftrightarrow \quad & \{ \text{ by (1.74), (1.76) and (1.79)}\} \\
& L \cong A^0 + A^1 + A^2 \times L \\
\leftrightarrow \quad & \{ \text{ another substitution as above and similar simplifications}\} \\
& L \cong A^0 + A^1 + A^2 + A^3 \times L \\
\leftrightarrow \quad & \{ \text{ after } (n+1)\text{-many similar steps}\} \\
& L \cong \sum_{i=0}^{n} A^i + A^{n+1} \times L
\end{aligned}
$$

Bearing a large n in mind, let us deliberate but temporarily ignore term $A^{n+1} \times L$. Then L will be isomorphic to the sum of n-many contributions 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, two element sequence $[a_1, a_2]$ which belongs to A^2 provided $a_1, a_2 \in A$, *etc.* 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:

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

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 was larger and larger and, "in the limit", it could be ignored. By analogy, taking a similar limit in the calculation just 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 2.1), we finally have:

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

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 — *e.g.* $x = \frac{x^2+3}{4}$, which admits two solutions 1 and 3 —, which have no solution at all — *e.g.* $x = x + 1$ —, or which admit an infinite number of — *e.g.* $x = x$.

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.

Exercise 2.1 Let A and B be two disjoint datatypes, that is, $A \cap B = \emptyset$ holds. Show that isomorphism

$$A \cup B \quad \cong \quad A + B \tag{2.6}$$

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

□

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 our discussion above, it is clear that set-theoretical equality "$=$" in this equation should give place to set-theoretical isomorphism ("$\cong$"):

$$L \quad \cong \quad 1 + A \times L \tag{2.7}$$

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, $A^\star$, so we may rephrase the question as: which isomorphism

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

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:

$$\begin{tikzcd} 1 \arrow[r, "i_1"] \arrow[dr, "in_1"'] & 1 + A \times A^\star \arrow[d, "in"] & A \times A^\star \arrow[l, "i_2"'] \arrow[dl, "in_2"] \\ & A^\star & \end{tikzcd} \tag{2.8}$$

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:

$$\begin{array}{l} cons : \; A \times A^\star \longrightarrow A^\star \\ cons(a, [a_1, \ldots, a_n]) = [a, a_1, \ldots, a_n] \end{array} \tag{2.9}$$

In a diagram:

$$\begin{tikzcd} 1 \arrow[r, "i_1"] \arrow[dr, "\underline{[\,]}"'] & 1 + A \times A^\star \arrow[d, "{[\, \underline{[\,]}, cons \,]}"] & A \times A^\star \arrow[l, "i_2"'] \arrow[dl, "cons"] \\ & A^\star & \end{tikzcd} \tag{2.10}$$

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

$$out \quad \stackrel{\mathrm{def}}{=} \quad (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?) \tag{2.11}$$

[2] By analogy, we may take (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:

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

$$\begin{array}{l} tl : A^\star \longrightarrow A^\star \\ tl\,[a_1, a_2, \ldots, a_n] = [a_2, \ldots, a_n] \end{array} \tag{2.13}$$

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

$$\begin{array}{cl}
& in \cdot out = id \\
\leftrightarrow & \quad \{ \text{ definitions of } in \text{ and } out \} \\
& [\,\underline{[\,]}, cons\,] \cdot (! + \langle hd, tl \rangle) \cdot (=_{[\,]}?) = id \\
\leftrightarrow & \quad \{ \text{ +-absorption (1.40) and (1.13)} \} \\
& [\,\underline{[\,]}, cons \cdot \langle hd, tl \rangle\,] \cdot (=_{[\,]}?) = id \\
\leftrightarrow & \quad \{ \text{ property of sequences: } cons(hd\ s, tl\ s) = s \} \\
& [\,\underline{[\,]}, id\,] \cdot (=_{[\,]}?) = id \\
\leftrightarrow & \quad \{ \text{ going pointwise } \} \\
& \left\{ \begin{array}{rcl} =_{[\,]}\ a & \Rightarrow & [\,\underline{[\,]}, id\,](i_1\,a) \\ \neg(=_{[\,]}\ a) & \Rightarrow & [\,\underline{[\,]}, id\,](i_2\,a) \end{array} \right. = a \\
\leftrightarrow & \quad \{ \text{ +-cancellation (1.37) } \} \\
& \left\{ \begin{array}{rcl} =_{[\,]}\ a & \Rightarrow & \underline{[\,]}\,a \\ \neg(=_{[\,]}\ a) & \Rightarrow & id\,a \end{array} \right. = a \\
\leftrightarrow & \quad \{ \ a = [\,] \text{ in one case and identity function (1.9) in the other } \} \\
& \left\{ \begin{array}{rcl} a = [\,] & \Rightarrow & a \\ \neg(a = [\,]) & \Rightarrow & a \end{array} \right. = a \\
\leftrightarrow & \quad \{ \text{ property } (p \rightarrow f, f) = f \text{ holds } \} \\
& a = a
\end{array}$$

A comment of 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 (2.7)? 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 1.17, can be written simply as

$$\texttt{data L = Nil} \mid \texttt{Cons (A,L)} \tag{2.14}$$

and generates diagram

$$\begin{tikzcd} 1 \arrow[r, "i_1"] \arrow[dr, "\underline{Nil}"'] & 1 + A \times L \arrow[d, "in'"] & A \times L \arrow[l, "i_2"'] \arrow[dl, "Cons"] \\ & L & \end{tikzcd} \tag{2.15}$$

where, of course, $in' = [\, \underline{Nil}, Cons \,]$.

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 (2.7), 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 (2.14):

$$\texttt{data T = Nil} \mid \texttt{Cons (A,T)} \tag{2.16}$$

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 in answering these questions. Suppose that A is a primitive numeric datatype, and that we add `deriving Show` to (2.16) 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

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

we confirm that Nil is itself an inhabitant of T. By typing

```
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
```

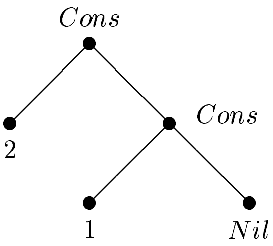
etc. We conclude that *expressions* involving Nil and $Cons$ are inhabitants of type T. Are these the *only* ones? The answer is *yes* because, by design of the HASKELL language, the constructors of type T will remain fixed once its declaration is interpreted, that is, no further constructor can be added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

$$\begin{array}{l} out_{\mathsf{T}} : \ \mathsf{T} \longrightarrow 1 + A \times \mathsf{T} \\ out_{\mathsf{T}}\, Nil = i_1 \text{ NIL} \\ out_{\mathsf{T}}(Cons(a, l)) = i_2(a, l) \end{array} \qquad (2.17)$$

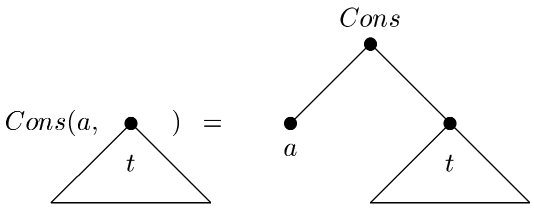
which is straightforwardly encoded in HASKELL using the `Either` realization of $+$ (recall sections 1.9 and 1.17):

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

In summary, isomorphism $\mathsf{T} \cong 1 + A \times \mathsf{T}$ holds, where datatype T is inhabited by symbolic expressions which we may visualize very conveniently as trees, for instance tree



which pictures 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 (2.16). Therefore, an alternative declaration such as, for instance,

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

would have been perfectly acceptable, generating another solution for the equation under isomorphism $[\underline{Stop}, Join]$. It is easy to check that we are but renaming Nil to $Stop$ and $Cons$ 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 2.2 Explain why

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

does not work as a HASKELL realization of the mediating arrow in diagram (2.10). How do you fix the problem?
□

2.3 Observing an inductive datatype

We are now interested in expressing *observations* of an inductive such as T (2.16), that is, functions of signature $B \xleftarrow{f} \mathsf{T}$ for some target type B. Suppose, for instance, that A is $\mathbb{N}$ (the set of all natural numbers) and that we want to multiply all elements which occur in a T list. Of course, we have to guarantee that multiplication is available in A,

$$mul : \ A \times A \longrightarrow A$$
$$mul(x,y) \stackrel{\mathrm{def}}{=} x\ y$$

and that $1 \in A$ is available to express the "multiplication of nothing". So arrow $A \xleftarrow{\underline{1}} 1$ is available. Of course, $mul(1,x) = mul(x,1) = x$ holds, for all $x \in A$. This property means that A, together with operator mul and constant 1, forms a *monoid*, a very important algebraic structure in computing science which will be exploited intensively throughout this book. The following arrow "packaging" A, mul and 1,

$$A \xleftarrow{[\,\underline{1}, mul\,]} 1 + A \times A \qquad (2.18)$$

is a convenient way to express such a structure. Combining this arrow with the isomorphism which defines solution T,

$$\mathsf{T} \xleftarrow{in_{\mathsf{T}}} 1 + A \times \mathsf{T}$$

and the function f we want to define, the target of which is $B = A$, we get the almost closed diagram which follows, in which only the dashed arrow is yet to be filled in:

$$\begin{tikzcd} \mathsf{T} \arrow[d, "f"'] & 1 + A \times \mathsf{T} \arrow[l, "in_{\mathsf{T}}"'] \arrow[d, dotted] \\ A & 1 + A \times A \arrow[l, "{[\,\underline{1}, mul\,]}"] \end{tikzcd} \tag{2.19}$$

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 A 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

$$\begin{tikzcd} \mathsf{T} \arrow[d, "f"'] & 1 + A \times \mathsf{T} \arrow[l, "{[\,\underline{Nil}, Cons\,]}"'] \arrow[d, "id + id \times f"] \\ A & 1 + A \times A \arrow[l, "{[\,\underline{1}, mul\,]}"] \end{tikzcd} \tag{2.20}$$

which pictures the following property of f

$$f \cdot [\,\underline{Nil}, Cons\,] \quad = \quad [\,\underline{1}, mul\,] \cdot (id + id \times f) \tag{2.21}$$

and which, in fact, defines f as the (recursive) function we had in mind, as the following calculation shows:

$$f \cdot [\,\underline{Nil}, Cons\,] = [\,\underline{1}, mul\,] \cdot (id + id \times f)$$

$\leftrightarrow$ { (1.39) on the lefthand side, (1.40) and identity id on the righthand side }

$$[\, f \cdot \underline{Nil}, f \cdot Cons\,] = [\,\underline{1}, mul \cdot (id \times f)\,]$$

$\leftrightarrow$ { *either* uniqueness}

$$f \cdot \underline{Nil} = \underline{1} \ \wedge \ f \cdot Cons = mul \cdot (id \times f)$$

$\leftrightarrow$ { going pointwise}

$$\left\{ \begin{array}{l} (f \cdot \underline{Nil})x = \underline{1}\, x \\ (f \cdot Cons)(a, x) = (mul \cdot (id \times f))(a, x) \end{array} \right.$$

$\leftrightarrow$ { standard pointwise simplifications}

$$\left\{ \begin{array}{l} f\, Nil = 1 \\ f(Cons(a, x)) = mul(a, f\, x) \end{array} \right.$$

Note that we could have used out_{T} in diagram (2.19),

$$\begin{tikzcd} \mathsf{T} \arrow[r, "out_{\mathsf{T}}"] \arrow[d, "f"'] & 1 + A \times \mathsf{T} \arrow[d, "id+id\times f"] \\ A & 1 + A \times A \arrow[l, "{[\,\underline{1}, mul\,]}"] \end{tikzcd} \tag{2.22}$$

obtaining another version of the *definition* of f,

$$f \quad = \quad [\,\underline{1}, mul\,] \cdot (id + id \times f) \cdot out_{\mathsf{T}} \tag{2.23}$$

which will lead to exactly the same pointwise recursive definition:

$$\begin{array}{ll}
& f = [\,\underline{1}, mul\,] \cdot (id + id \times f) \cdot out_{\mathsf{T}} \\
\leftrightarrow & \qquad \{ \text{ (1.40) and identity } id \text{ on the righthand side } \} \\
& f = [\,\underline{1}, mul \cdot (id \times f)\,] \cdot out_{\mathsf{T}} \\
\leftrightarrow & \qquad \{ \text{ going pointwise on } out_{\mathsf{T}} \text{ (2.17) } \} \\
& \left\{ \begin{array}{l} f\,Nil = ([\,\underline{1}, mul \cdot (id \times f)\,] \cdot out_{\mathsf{T}})Nil \\ f(Cons(a,x)) = ([\,\underline{1}, mul \cdot (id \times f)\,] \cdot out_{\mathsf{T}})(a,x) \end{array} \right. \\
\leftrightarrow & \qquad \{ \text{ definition of } out_{\mathsf{T}} \text{ (2.17)} \} \\
& \left\{ \begin{array}{l} f\,Nil = ([\,\underline{1}, mul \cdot (id \times f)\,] \cdot i_1)Nil \\ f(Cons(a,x)) = ([\,\underline{1}, mul \cdot (id \times f)\,] \cdot i_2)(a,x) \end{array} \right. \\
\leftrightarrow & \qquad \{ \text{ +-cancellation (1.37) } \} \\
& \left\{ \begin{array}{l} f\,Nil = \underline{1}\,Nil \\ f(Cons(a,x)) = (mul \cdot (id \times f))\,(a,x) \end{array} \right. \\
\leftrightarrow & \qquad \{ \text{ simplification } \} \\
& \left\{ \begin{array}{l} f\,Nil = 1 \\ f(Cons(a,x)) = mul(a, f\,x) \end{array} \right.
\end{array}$$

Note that 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 $A^\star$ instead of T, that is, for $out = (! + \langle hd, tl\rangle) \cdot (=_{[\,]}?)$ — recall (2.11) — taking place of out_{T}, we get a "more algorithmic" version of f:

$$\begin{array}{ll}
& f = [\,\underline{1}, mul\,] \cdot (id + id \times f) \cdot (! + \langle hd, tl\rangle) \cdot (=_{[\,]}?) \\
\leftrightarrow & \qquad \{ \text{ +-functor (1.41), identity and } \times\text{-absorption (1.24) } \} \\
& f = [\,\underline{1}, mul\,] \cdot (! + \langle hd, f \cdot tl\rangle) \cdot (=_{[\,]}?) \\
\leftrightarrow & \qquad \{ \text{ +-absorption (1.40) and constant } \underline{1} \text{ } \}
\end{array}$$

$$f = [\,\underline{1}, mul \cdot \langle hd, f \cdot tl\rangle\,] \cdot (=_{[\,]}?)$$

$\leftrightarrow$ { going pointwise on guard $=_{[\,]}?$ (1.55) and simplifying }

$$f\,l = \begin{cases} l = [\,] & \Rightarrow \underline{1}\,l \\ \neg(l = [\,]) & \Rightarrow (mul \cdot \langle hd, f \cdot tl\rangle)\,l \end{cases}$$

$\leftrightarrow$ { simplification }

$$f\,l = \begin{cases} l = [\,] & \Rightarrow 1 \\ \neg(l = [\,]) & \Rightarrow mul(hd\,l, f(tl\,l)) \end{cases}$$

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

```
f   :: [A] -> A
f l | l == []   = 1
    | otherwise = mul(head l, (f (tail l)))
```

or

```
f   :: [A] -> A
f l = if l == []
      then 1
      else mul(head l, (f (tail l)))
```

both requiring the equality predicate "==" and destructors "`head`" and "`tail`". The inductive version is somewhat more economic:

```
f   :: T -> A
f Nil       = 1
f(Cons(a,l))= mul(a,f l)
```

2.4 Synthesizing an inductive datatype

The issue which concerns us in this section dualizes what we have dealt with in the previous one: instead of analyzing or *observing* an inductive type such as T (2.16), 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. For example, let $B = A = \mathbb{N}_0$ 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 (2.22) applicable to the new situation. In trying to copy 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. Both identity functions are their own inverses, and in_{T} takes the place of out_{T}:

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

It is interesting to see that only the bottom arrow is not obvious to reverse. This means that we have to "invent" a particular destructor of A, say $A \xrightarrow{g} 1 + A \times A$, 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,

$$f \quad = \quad in_{\mathsf{T}} \cdot (id + id \times f) \cdot g \tag{2.24}$$

which is equivalent to:

$$f \quad = \quad [\, \underline{Nil}, Cons \cdot (id \times f) \,] \cdot g \tag{2.25}$$

Function g will act as a generator of the computation which should distinguish input 0 from all the others in a way that Nil is output by f. We thus may decompose g as follows:

$$A \xrightarrow{=_0?} A + A \xrightarrow{!+h} 1 + A \times A$$

What about h? It 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 produced contiguously and in decreasing order, we may define $h = \langle id, \mathit{pred} \rangle$ where, for $n > 0$, $\mathit{pred}\, n \stackrel{\mathrm{def}}{=} n - 1$ computes the *predecessor* of n. Filling this in (2.25) we get

$$\begin{array}{ll}
& f = [\, \underline{Nil}, Cons \cdot (id \times f) \,] \cdot (! + \langle id, \mathit{pred} \rangle) \cdot (=_0?) \\
\leftrightarrow & \qquad \{ \ +\text{-absorption (1.40) followed by } \times\text{-absorption (1.24) } \textit{etc.} \} \\
& f = [\, \underline{Nil}, Cons \cdot \langle id, f \cdot \mathit{pred} \rangle \,] \cdot (=_0?) \\
\leftrightarrow & \qquad \{ \text{ going pointwise on guard } =_0? \text{ (1.55) and simplifying } \} \\
& f\, n = \left\{ \begin{array}{rcl} n = 0 & \Rightarrow & Nil \\ \neg(n = 0) & \Rightarrow & Cons(n, f\,(n-1)) \end{array} \right.
\end{array}$$

which matches the function we had in mind:

```
f   :: A -> T
f n | n == 0    = Nil
    | otherwise = Cons(n,f(n-1))
```

We shall see briefly that the constructions of the f function calculating the multiplication of 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. Before that, we need to introduce some terminology.

2.5 Introducing (list) catas, anas and hylos

Suppose that, back to section 2.3, we want to *add*, other than multiply, the elements occurring in lists of type T (2.16). 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 $A \xleftarrow{[\,\underline{1}, mul\,]} 1 + A \times A$ of diagram (2.22) needs to be replaced, because this is the only place where we can specify that the target datatype is now regarded as the carrier of another (additive rather than multiplicative) monoidal structure,

$$A \xleftarrow{[\,\underline{0}, add\,]} 1 + A \times A$$

for $add(x, y) \stackrel{\mathrm{def}}{=} x + y$. We are saying that the list is now reduced by the addition operator and that output value 0 is now expected as the result of "adding nothing".

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. Intuition will once again help us in deciding that only arrow g in (2.24) needs to be updated.

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

$$\begin{tikzcd} \mathsf{T} \arrow[r, "out_{\mathsf{T}}"] \arrow[d, "f"'] & 1 + A \times \mathsf{T} \arrow[d, "id + id \times f"] \\ B & 1 + A \times B \arrow[l, "g"] \end{tikzcd} \qquad \begin{tikzcd} \mathsf{T} & 1 + A \times \mathsf{T} \arrow[l, "in_{\mathsf{T}}"'] \\ B \arrow[u, "f"] \arrow[r, "g"'] & 1 + A \times B \arrow[u, "id + id \times f"'] \end{tikzcd} \tag{2.26}$$

It so happens that each f is uniquely dependent on its diagram's g arrow, that is to say, each particular instantiation of a g will determine the corresponding f. So both gs can be regarded as "genetic material" of the f functions they uniquely define.

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 list catamorphism induced by g" and $[\!(g)\!]$ as "the list 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 arrows in each diagram).

In summary, the list catamorphism induced by $B \xleftarrow{g} 1 + A \times B$ is the unique function $B \xleftarrow{(\![g]\!)} \mathsf{T}$ which obeys to property (or is defined by)

$$(\![g]\!) \quad = \quad g \cdot (id + id \times (\![g]\!)) \cdot out_{\mathsf{T}} \tag{2.27}$$

while the list anamorphism induced by $B \xrightarrow{g} 1 + A \times B$ is the unique function $B \xrightarrow{[\!(g)\!]} \mathsf{T}$ which obeys to property (or is defined by)

$$[\!(g)\!] \quad = \quad in_{\mathsf{T}} \cdot (id + id \times [\!(g)\!]) \cdot g \tag{2.28}$$

It should be observed that T can act as a mediator between a list anamorphism and a list catamorphism, that is to say, $B \xleftarrow{[\!(g)\!]} \mathsf{T}$ composes with $\mathsf{T} \xleftarrow{[\!(h)\!]} C$, for