and then define the "extended" version of head:

```
exthead :: [a] → ExtVal a
exthead [ ] = Error
exthead x = Ok (head x)
```

Note that *ExtVal* is a *parametric* type which extends an arbitrary data type a with its (polymorphic) exception (or error value). It turns out that, in HASKELL, *ExtVal* is redundant because such a parametric type already exists and is called *Maybe*:

```
data Maybe a = Nothing | Just a
```

Clearly, the isomorphisms hold:

$$\mathsf{ExtVal}\,A \cong \mathsf{Maybe}\,A \cong 1 + A$$

So, we might have written the more standard code

```
exthead :: [a] → Maybe a
exthead [ ] = Nothing
exthead x = Just (head x)
```

In abstract terms, both alternatives coincide, since one may regard as *partial* every function of type

$$\begin{tikzcd} 1 + A & B \arrow[l, "g"'] \end{tikzcd}$$

for some A and B [1].

4.2 PUTTING PARTIAL FUNCTIONS TOGETHER

Do partial functions compose? Their types won't match in general:

$$\begin{tikzcd} & 1 + B \arrow[d, dotted, dash] & A \arrow[l, "g"'] \\ 1 + C & B \arrow[l, "f"'] & \end{tikzcd}$$

Clearly, we have to extend f — which is itself a partial function — to some f' able to accept arguments from $1 + B$:

$$\begin{tikzcd} & 1 \arrow[d, "i_1"] \arrow[ddl, "\cdots"'] & \\ & 1 + B \arrow[dl, "f'"] & A \arrow[l, "g"'] \\ 1 + C & B \arrow[l, "f"] \arrow[u, "i_2"'] & \end{tikzcd}$$

[1] In conventional programming, every function delivering a *pointer* as result — as in *e.g.* the C programming language — can be regarded as one of these functions.

From an adjunction (4.57) a monad $\mathsf{T} = \mathsf{F} \cdot \mathsf{G}$ arises defined by $\eta = \lceil id \rceil$ and $\mu = \mathsf{F}\lfloor id \rfloor$. Finally, from all this we can infer the generic version of $f \bullet g$,

$$f \bullet g = \lceil \lfloor f \rfloor \cdot \lfloor g \rfloor \rceil \tag{4.67}$$

by replaying the calculation which lead to (4.43):

$$\begin{array}{cl}
 & f \bullet g \\
= & \quad \{\ (4.5)\ \} \\
 & \mu \cdot \mathsf{T} f \cdot g \\
= & \quad \{\ \mathsf{T} = \mathsf{F} \cdot \mathsf{G} ; \mu = \mathsf{F}\lfloor id \rfloor\ \} \\
 & \mathsf{F}\lfloor id \rfloor \cdot (\mathsf{F}\ (\mathsf{G} f)) \cdot g \\
= & \quad \{\ \text{functor } \mathsf{F}\ \} \\
 & \mathsf{F}\ (\lfloor id \rfloor \cdot \mathsf{G} f) \cdot g \\
= & \quad \{\ \text{cancellation: } \lfloor id \rfloor \cdot \mathsf{G} f = \lfloor f \rfloor ; g = \lceil \lfloor g \rfloor \rceil\ \} \\
 & \mathsf{F}\ \lfloor f \rfloor \cdot \lceil \lfloor g \rfloor \rceil \\
= & \quad \{\ \text{absorption: } (\mathsf{F}\ g) \cdot \lceil h \rceil = \lceil g \cdot h \rceil\ \} \\
 & \lceil \lfloor f \rfloor \cdot \lfloor g \rfloor \rceil
\end{array}$$

Finally, let us see another example of a monad arising from one such adjunction (4.57). Recall exercise 2.27, on page 41, where pair / unpair witness an isomorphism similar to that of *curry*/*uncurry*, for $\mathsf{pair}\ (f,g) = \langle f,g \rangle$ and $\mathsf{unpair}\ k = (\pi_1 \cdot k, \pi_2 \cdot k)$. This can be cast into an adjunction as follows

$$\begin{array}{cl}
 & k = \mathsf{pair}\ (f,g)\ \Leftrightarrow\ (\pi_1 \cdot k, \pi_2 \cdot k) = (f,g) \\
\equiv & \quad \{\ \text{see below}\ \} \\
 & k = \mathsf{pair}\ (f,g)\ \Leftrightarrow\ (\pi_1, \pi_2) \cdot (\mathsf{G}\ k) = (f,g)
\end{array}$$

where $\mathsf{G}\ k = (k,k)$. Note the abuse of notation, on the righthand side, of extending function composition notation to composition of *pairs* of functions, defined in the expected way: $(f,g) \cdot (h,k) = (f \cdot h, g \cdot k)$. Note that, for $f : A \to B$ and $g : C \to D$, the pair (f,g) has type $(A \to B) \times (C \to D)$. However, we shall abuse of notation again and declare the type $(f,g) : (A,C) \to (B,D)$.[7] In the opposite direction, $\mathsf{F}\ (f,g) = f \times g$:

$$\begin{tikzcd}
B \times A & (B \times A, B \times A) \arrow[r, "(\pi_1,\pi_2)"] & (B,A) \\
C \arrow[u, "k=\mathsf{pair}\ (f,g)"] & (C,C) \arrow[u, "(k,k)"] \arrow[ur, "(f,g)"'] &
\end{tikzcd} \tag{4.68}$$

[7] Strictly speaking, we are not abusing notation but rather working on a new *category*, that is, another mathematical system where functions and objects always come in pairs. For more on categories see the standard textbook [37].

Clearly, not every relation obeys (5.5), for instance

$$2 < 3 \wedge 1 < 3 \quad \not\Rightarrow \quad 2 = 1$$

Relations obeying (5.5) will be referred to as *simple*, according to a terminology to follow shortly.

Implication (5.6) expresses the (philosophically) interesting fact that no function (observation) can be found able to distinguish between two equal objects. This is another fact true about functions which does not generalize to binary relations, as we shall see when we come back to this later.

Recapitulating: we regard *function* $f : A \longrightarrow B$ as the binary relation which relates b to a iff $b = f\ a$. So,

$$b\ f\ a \ \text{ literally means } \ b = f\ a \tag{5.7}$$

The purpose of this chapter is to generalize from

$$\boxed{\begin{array}{c} B \xleftarrow{\ f\ } A \\ b = f\ a \end{array}} \quad \text{to} \quad \boxed{\begin{array}{c} B \xleftarrow{\ R\ } A \\ b\ R\ a \end{array}}$$

5.3 PRE/POST CONDITIONS

It should be noted that relations are used in virtually every body of science and it is hard to think of another way to express human knowledge in philosophy, epistemology and common life, as suggestively illustrated in figure 5.1. This figure is also illustrative of another popular ingredient when using relations — the *arrows* drawn to denote relationships.[1]

In real life, "everything appears to be a relation". This has lead software theorists to invent linguistic layouts for relational specification, leading to so-called *specification languages*. One such language, today historically relevant, is the language of the Vienna Development Method (VDM). In this notation, the relation described in (5.3) will be written:

$$\begin{array}{l} R\ (x : \mathbb{N}_0)\ y : \mathbb{N}_0 \\ \mathsf{post}\ y \geqslant x + 1 \end{array}$$

where the clause prefixed by post is said to be a post-condition. The format also includes pre-conditions, if necessary. Such is the case of the following pre / post -styled specification of the operation that extracts an arbitrary element from a set:

$$\begin{array}{l} \mathit{Pick}\ (x : \mathsf{P}A)\ (r : A, y : \mathsf{P}A) \\ \mathsf{pre}\ x \neq \{\ \} \\ \mathsf{post}\ r \in x \wedge y = x - \{r\} \end{array} \tag{5.8}$$

[1] Our extensive use of arrows to denote relations in the sequel is therefore rooted on common, informal practice. Unfortunately, mathematicians do not follow such practice and insist on regarding relations just as sets of pairs.

RELATION SHRINKING Given relations $R : A \leftarrow B$ and $S : A \leftarrow A$, define $R \upharpoonright S : A \leftarrow B$, pronounced "$R$ shrunk by S", by

$$X \subseteq R \upharpoonright S \;\equiv\; X \subseteq R \wedge X \cdot R^{\circ} \subseteq S \tag{5.179}$$

cf. diagram:

$$\begin{tikzcd}
 & B \arrow[dl, "R\upharpoonright S"'] \arrow[d, "R"] \\
A & A \arrow[l, "S"]
\end{tikzcd}$$

This states that $R \upharpoonright S$ is the largest part of R such that, if it yields an output for an input x, it must be a maximum, with respect to S, among all possible outputs of x by R. By indirect equality, (5.179) is equivalent to the closed definition:

$$R \upharpoonright S \;=\; R \cap S/R^{\circ} \tag{5.180}$$

(5.179) can be regarded as a Galois connection between the set of all *subrelations* of R and the set of *optimization criteria* (S) on its outputs.

Combinator $R \upharpoonright S$ also makes sense when R and S are finite, relational data structures (eg. tables in a database). Consider, for instance, the following example of $R \upharpoonright S$ in a *data-processing* context: given

Examiner	*Mark*	*Student*
Smith	10	*John*
Smith	11	*Mary*
Smith	15	*Arthur*
Wood	12	*John*
Wood	11	*Mary*
Wood	15	*Arthur*

and wishing to "choose the best mark" for each student, project over *Mark*, *Student* and optimize over the $\geqslant$ ordering on *Mark*:

$$\left(\begin{array}{c|c} \mathit{Mark} & \mathit{Student} \\ \hline 10 & \mathit{John} \\ 11 & \mathit{Mary} \\ 12 & \mathit{John} \\ 15 & \mathit{Arthur} \end{array}\right) \upharpoonright \geqslant \;=\; \left(\begin{array}{c|c} \mathit{Mark} & \mathit{Student} \\ \hline 11 & \mathit{Mary} \\ 12 & \mathit{John} \\ 15 & \mathit{Arthur} \end{array}\right)$$

Relational shrinking can be used in many other contexts. Consider, for instance, a sensor recording temperatures (T), $T \xleftarrow{S} \mathbb{N}_0$, where data in $\mathbb{N}_0$ are "time stamps". Suppose one wishes to filter out repeated temperatures, keeping the first occurrences only. This can be specified by:

$$T \xleftarrow{nub\ S} \mathbb{N}_0 \;=\; (S^{\circ} \upharpoonright \leqslant)^{\circ}$$

That is, *nub* is the function that removes all duplicates while keeping the first instances.

A

BACKGROUND — EINDHOVEN QUANTIFIER CALCULUS

This appendix is a quick reference summary of section 4.3 of reference [4].

A.1 NOTATION

The Eindhoven quantifier calculus adopts the following notation standards:

- $\langle \forall\, x \ : \ R : \ T \rangle$ means: *"for **all** x in the range R, term T holds"*, where R and T are logical expressions involving x.
- $\langle \exists\, x \ : \ R : \ T \rangle$ means: *"for **some** x in the range R, term T holds"*.

A.2 RULES

The main rules of the Eindhoven quantifier calculus are listed below:

Trading:

$$\langle \forall\, k \ : \ R \wedge S : \ T \rangle \ = \ \langle \forall\, k \ : \ R : \ S \Rightarrow T \rangle \tag{A.1}$$

$$\langle \exists\, k \ : \ R \wedge S : \ T \rangle \ = \ \langle \exists\, k \ : \ R : \ S \wedge T \rangle \tag{A.2}$$

de Morgan:

$$\neg \langle \forall\, k \ : \ R : \ T \rangle \ = \ \langle \exists\, k \ : \ R : \ \neg T \rangle \tag{A.3}$$

$$\neg \langle \exists\, k \ : \ R : \ T \rangle \ = \ \langle \forall\, k \ : \ R : \ \neg T \rangle \tag{A.4}$$

One-point:

$$\langle \forall\, k \ : \ k = e : \ T \rangle \ = \ T[k := e] \tag{A.5}$$

$$\langle \exists\, k \ : \ k = e : \ T \rangle \ = \ T[k := e] \tag{A.6}$$

Nesting:

$$\langle \forall\, a, b \ : \ R \wedge S : \ T \rangle \ = \ \langle \forall\, a \ : \ R : \ \langle \forall\, b \ : \ S : \ T \rangle \rangle \tag{A.7}$$

$$\langle \exists\, a, b \ : \ R \wedge S : \ T \rangle \ = \ \langle \exists\, a \ : \ R : \ \langle \exists\, b \ : \ S : \ T \rangle \rangle \tag{A.8}$$

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

What about other facts valid for numeric exponentials such as $a^0 = 1$ and $1^a = 1$? The reader is invited to go back to section 2.11 and recall what 0 and 1 mean as datatypes: the empty (void) and singleton datatypes, respectively. Our counterpart to $a^0 = 1$ then is

$$A^0 \cong 1 \tag{2.98}$$

where A^0 denotes the set of all functions from the empty set to some A. What does (2.98) mean? It simply tells us that there is only one function in such a set — the empty function mapping "no" value at all. This fact confirms our choice of notation once again (compare with $a^0 = 1$ in a numeric context).

Next, we may wonder about facts

$$1^A \cong 1 \tag{2.99}$$

$$A^1 \cong A \tag{2.100}$$

which are the functional exponentiation counterparts of $1^a = 1$ and $a^1 = a$. Fact (2.99) is valid: it means that there is only one function mapping A to some singleton set $\{c\}$ — the constant function $\underline{c}$. There is no room for another function in 1^A because only c is available as output value. Our standard denotation for such a unique function is given by (2.58).

Fact (2.100) is also valid: all functions in A^1 are (single valued) constant functions and there are as many constant functions in such a set as there are elements in A. These functions are often called (abstract) "points" because of the 1-to-1 mapping between A^1 and the elements (points) in A.

***Exercise** 2.26. Relate the isomorphism involving generic elementary type* 2

$$A \times A \cong A^2 \tag{2.101}$$

to the expression $\lambda f \rightarrow (f\ \mathsf{True}, f\ \mathsf{False})$ *written in* HASKELL *syntax.*
□

***Exercise** 2.27. Consider the witnesses of isomorphism (2.97)*

$$\begin{tikzcd}
(B \times C)^A \arrow[rr, bend left, "\mathsf{unpair}"] & \cong & B^A \times C^A \arrow[ll, bend left, "\mathsf{pair}"]
\end{tikzcd}$$

defined by:

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

think of the generic situation of a function $D \xleftarrow{f} C$ which observes datatype C:

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "C1"'] & A+B \arrow[d, "{[C1,C2]}"] & B \arrow[l, "i_2"'] \arrow[dl, "C2"] \\
& C \arrow[d, "f"] & \\
& D &
\end{tikzcd}$$

This is an opportunity for $+$-*fusion* (2.42), whereby we obtain

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

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

$$\begin{array}{l}
f : C \rightarrow D \\
f(C1\,a) = \ldots \\
f(C2\,b) = \ldots
\end{array}$$

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

$$\begin{array}{ll}
& inv \cdot [C1, C2] = id \\
\equiv & \qquad \{ \text{ by } +\textit{-fusion} \text{ (2.42) } \} \\
& [inv \cdot C1, inv \cdot C2] = id \\
\equiv & \qquad \{ \text{ by } +\textit{-reflexion} \text{ (2.41) } \} \\
& [inv \cdot C1, inv \cdot C2] = [i_1, i_2] \\
\equiv & \qquad \{ \textit{ either} \text{ structural equality (2.66) } \} \\
& inv \cdot C1 = i_1 \wedge inv \cdot C2 = i_2
\end{array}$$

Therefore:

$$\begin{array}{l}
inv : C \rightarrow A + B \\
inv(C1\,a) = i_1\,a \\
inv(C2\,b) = i_2\,b
\end{array}$$

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

$$C \xleftarrow{[C1,C2]} A + B \qquad (2.111)$$

$[C1, C2]$ is called the *algebra* of datatype C and its inverse *inv* is called the *coalgebra* of C. The algebra contains the constructors C1 and C2 of *find g and h such that $ap \cdot [g \times id, h \times id] = id$. Conclude that*

$$\overline{\mathsf{distl}} = [\overline{i_1}, \overline{i_2}] \tag{2.118}$$

Draw the type diagram of $\overline{\mathsf{distl}}$.

□

***Exercise** 2.37. The arithmetic law $(a+b)(c+d) = (ac+ad)+(bc+bd)$ corresponds to the isomorphism*

$$\begin{tikzcd}(A+B) \times (C+D) \arrow[r, phantom, "\cong"] & (A \times C + A \times D) + (B \times C + B \times D) \arrow[l, bend left, "h = [[i_1 \times i_1, i_1 \times i_2], [i_2 \times i_1, i_2 \times i_2]]"]\end{tikzcd}$$

From universal property (2.65) infer the following definition of function h, written in Haskell syntax:

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

□

***Exercise** 2.38. Every C programmer knows that a struct of pointers*

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

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

$$\begin{tikzcd}(A+1) \times (B+1) & ((A+1) \times B) + ((A+1) \times 1) \arrow[l, "i_1"'] \\ & (A \times B + 1 \times B) + (A \times 1 + 1 \times 1) \arrow[u, "i_2"'] \\ & (A \times B + B) + (A+1) \arrow[u, "i_3"'] \\ (A \times B + (B+A)) + 1 \arrow[r, "i_5"'] & A \times B + (B + (A+1)) \arrow[u, "i_4"']\end{tikzcd}$$

which witness the observation above.

□

***Exercise** 2.39. Prove the following property of McCarthy conditionals:*

$$p \rightarrow f \cdot g\ ,\ h \cdot k \quad = \quad [f, h] \cdot (p \rightarrow i_1 \cdot g\ ,\ i_2 \cdot k) \tag{2.119}$$

□

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

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

More knowledge about for-loops can be extracted from (3.7). Later on we will show that these constructs are special cases of a more general concept termed *catamorphism*.[3] In the usual *"banana-bracket"* notation of catamorphisms, to be introduced later, the for-combinator will be written:

$$\mathsf{for}\, g\, k = (\!|\, [\underline{k}, g] \,|\!) \tag{3.9}$$

In the sequel, we shall study the (more general) theory of catamorphisms and come back to for-loops as an instantiation. Then we will understand how more interesting for-loops can be synthesized, for instance those handling more than one "global variable", thanks to catamorphism theory (for instance, the mutual recursion laws).

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

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

$$(a+) \quad = \quad \mathsf{for}\ \mathsf{succ}\ a \tag{3.10}$$

□

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

$$h \cdot (\mathsf{for}\, g\, k) = \mathsf{for}\, j\, (h\, k) \quad \Leftarrow \quad h \cdot g = j \cdot h \tag{3.11}$$

can be derived from universal-property (3.7) [4]*. Since $(a+) \cdot id = (a+)$, provide an alternative derivation of (3.10) using the fusion-law above.*

□

***Exercise** 3.3. From (3.4) and fusion-law (3.11) infer: $(a*) \cdot \mathsf{succ} = \mathsf{for}\ a\ (a+)$.*

□

[3] See eg. section 3.6.

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

would have been perfectly acceptable, generating another solution for the equation under algebra $[\underline{Stop}, Join]$. It is easy to check that (3.28) is but a renaming of Nil to $Stop$ and of $Cons$ to $Join$. Therefore, both datatypes are isomorphic, or "abstractly the same".

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

We wrote "inductively" above because the set of all expressions (trees) which inhabit the type is defined by induction. Such types are called *inductive* and we shall have a lot more to say about them in chapter 8 .

***Exercise** 3.6. Obviously,*

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

does not work as a HASKELL *realization of the mediating arrow in diagram (3.20). What do you need to write instead?*

□

3.4 OBSERVING AN INDUCTIVE DATATYPE

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

$$add : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$$
$$add(x, y) \overset{\mathrm{def}}{=} x + y$$

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 $\underline{0}$,

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

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

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