Chapter 4

Why Monads Matter

In this chapter we present a powerful device in state-of-the-art functional programming, that of a *monad*. The monad concept is nowadays of primary importance in computing science because it makes it possible to describe computational effects as disparate as input/output, comprehension notation, state variable updating, probabilistic behaviour, context dependence, partial behaviour *etc.* in an elegant and uniform way.

Our motivation to this concept will start from a well-known problem in functional programming (and computing as a whole) — that of coping with undefined computations.

4.1 Partial functions

Recall the function head that yields the first element of a finite list. Clearly, head x is undefined for $x = [\,]$ because the empty list has no elements at all. As expected, the HASKELL output for head $[\,]$ is just "panic":

```
*Main> head []
*** Exception: Prelude.head: empty list
*Main>
```

Functions such as head are called *partial functions* because they cannot be applied to all of their (well-typed) inputs, *i.e.*, they diverge for some of such inputs. Partial functions are very common in mathematics or programming — for other examples think of *e.g.* tail, and so on.

Panic is very dangerous in programming. In order to avoid this kind of behaviour one has two alternatives, either (a) ensuring that every call to head x

is *protected* — *i.e.*, the contexts which wrap up such calls ensure *pre-condition* $x \neq [\,]$, or (b) *raising* exceptions, *i.e.* explicit error values, as above. In the former case, mathematical proofs need to be carried out in order to ensure *safety* (that is, *pre-condition* compliance). The overall effect is that of *restricting* the domain of the partial function. In the latter case one goes the other way round, by *extending* the co-domain (vulg. range) of the function so that it accommodates exceptional outputs. In this way one might define, in HASKELL:

$$\mathbf{data}\ ExtVal\ a = Ok\ a \mid Error$$

and then define the "extended" version of head:

$$\begin{aligned} &exthead :: [a] \to ExtVal\ a \\ &exthead\ [\,] = Error \\ &exthead\ x = Ok\ (\mathsf{head}\ x) \end{aligned}$$

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

$$\mathbf{data}\ Maybe\ a = \mathsf{Nothing} \mid \mathsf{Just}\ a$$

Clearly, the isomorphisms hold:

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

So, we might have written the more standard code

$$\begin{aligned} &exthead :: [a] \to Maybe\ a \\ &exthead\ [\,] = \mathsf{Nothing} \\ &exthead\ x = \mathsf{Just}\ (\mathsf{head}\ x) \end{aligned}$$

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

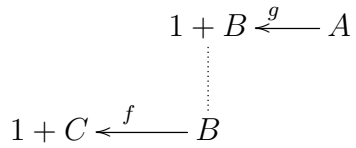
$$1 + A \xleftarrow{g} B$$

for some A and B [1].

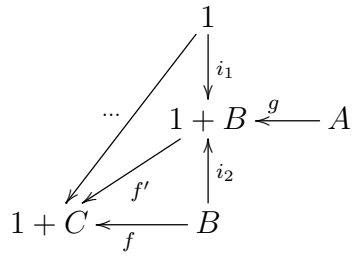
[1] In conventional programming, every function delivering a *pointer* as result — as in *e.g.* the C programming language — can be regarded as one of these functions.

4.2 Putting partial functions together

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



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



The most "obvious" instance of the ellipsis $(\ldots)$ in the diagram above is i_1 and this corresponds to what is called *strict* composition: an exception produced by the *producer* function g is propagated to the output of the *consumer* function f. We define:

$$f \bullet g \stackrel{\text{def}}{=} [i_1 , f] \cdot g \tag{4.1}$$

Expressed in terms of $Maybe$, composite function $f \bullet g$ works as follows:

$$(f \bullet g)a \quad = \quad f'(g\, a)$$

where

$$f'\ \mathsf{Nothing} = \mathsf{Nothing}$$
$$f'\ (\mathsf{Just}\ b) = f\ b$$

Altogether, we have the following Haskell pointwise expression for $f \bullet g$:

$$\lambda a \rightarrow f'\ (g\ a)\ \mathbf{where}$$
$$\quad f'\ \mathsf{Nothing} = \mathsf{Nothing}$$
$$\quad f'\ (\mathsf{Just}\ b) = f\ b$$

Note that the adopted extension of f can be decomposed — by reverse $+$-absorption (2.43) — into

$$f' \quad = \quad [i_1 , id] \cdot (id + f)$$

as displayed in diagram

$$\begin{array}{ccccc} 1 + (1 + C) & \xleftarrow{id+f} & 1 + B & \xleftarrow{g} & A \\ {\scriptstyle [i_1 , id]}\downarrow & & \vdots & & \\ 1 + C & \xleftarrow{\quad f \quad} & B & & \end{array}$$

All in all, we have the following version of (4.1):

$$f \bullet g \quad \stackrel{\text{def}}{=} \quad [i_1 , id] \cdot (id + f) \cdot g$$

Does this functional composition scheme have a unit, that is, is there a function u such that

$$f \bullet u = f = u \bullet f \tag{4.2}$$

holds? Clearly, if it exists, it must bear type $1 + A \xleftarrow{u} A$. $1 + A \xleftarrow{i_2} A$ has the same type, so $u = i_2$ is a very likely solution. Let us check it:

$$\begin{array}{ll} & f \bullet u = f = u \bullet f \\ \equiv & \qquad \{ \text{ substitutions } \} \\ & [i_1 , f] \cdot i_2 = f = [i_1 , i_2] \cdot f \\ \equiv & \qquad \{ \text{ by } +\text{-cancellation (2.40) and } +\text{-reflection (2.41) } \} \\ & f = f = id \cdot f \\ \equiv & \qquad \{ \text{ trivial } \} \\ & true \end{array}$$

So $f \bullet u = f = u \bullet f$ for $u = i_2$.

Exercise 4.1. *Prove that property*

$$f \bullet (g \bullet h) \quad = \quad (f \bullet g) \bullet h$$

holds, for $f \bullet g$ defined by (4.1).
□

4.3 Lists

In contrast to partial functions, which may produce *no* output, let us now consider functions which may deliver *too many* outputs, for instance, lists of output values:

$$\begin{array}{ccc} & & B^\star \\ & & \vdots \\ C^\star & \xleftarrow{f} & B \end{array} \qquad \begin{array}{c} \\ \xleftarrow{g} \\ \\ \end{array} \begin{array}{c} A \\ \\ \\ \end{array}$$

Functions f and g do not compose but, once again, one can think of extending the consumer function (f) by mapping it along the output of the producer function (g):

$$\begin{array}{ccc} (C^\star)^\star & \xleftarrow{f^\star} & B^\star \\ \vdots & & \vdots \\ C^\star & \xleftarrow{f} & B \end{array}$$

To complete the process, one has to *flatten* the nested-sequence output in $(C^\star)^\star$ via the obvious list-catamorphism $C^\star \xleftarrow{\mathsf{concat}} (C^\star)^\star$, recall $\mathsf{concat} = (\!|[\underline{[\,]}, \mathsf{conc}]|\!)$ where $\mathsf{conc}\ (x, y) = x +\!\!+ y$. In summary:

$$f \bullet g \stackrel{\text{def}}{=} \mathsf{concat} \cdot f^\star \cdot g \tag{4.3}$$

as captured in the following diagram:

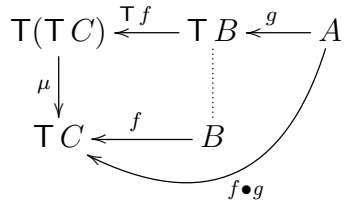
$$\begin{array}{ccccc} (C^\star)^\star & \xleftarrow{f^\star} & B^\star & \xleftarrow{g} & A \\ {\scriptstyle \mathsf{concat}}\downarrow & & \vdots & & \\ C^\star & \xleftarrow{f} & B & & \end{array}$$

Exercise 4.2. *Show that* $singl$ *(recall exercise 3.14) is the unit* u *of* $\bullet$ *as defined by (4.3) above.*
□

Exercise 4.3. *Encode in* HASKELL *a pointwise version of (4.3).* ***Hint****: start by applying (list) cata-absorption (3.77).*
□

4.4 Monads

Both function composition schemes (4.1) and (4.3) above share the same polytypic pattern: the output of the producer function g is *"T-times" more elaborate* than the input of the consumer function f, where T is some parametric datatype: $\mathsf{T}\ X = 1+X$ in case of (4.1), and $\mathsf{T}\ X = X^\star$ in case of (4.3). Then a composition scheme is devised for such functions, which is displayed in



(4.4)

and is given by

$$f \bullet g \stackrel{\text{def}}{=} \mu \cdot \mathsf{T}\ f \cdot g \tag{4.5}$$

where $\mathsf{T}\ A \xleftarrow{\mu} \mathsf{T}^2\ A$ is a suitable polymorphic function. (We have already seen $\mu = [i_1, id]$ in case (4.1), and $\mu = \mathsf{concat}$ in case (4.3).)

Together with a unit function $\mathsf{T}\ A \xleftarrow{u} A$ and μ, that is

$$A \xrightarrow{u} \mathsf{T}\ A \xleftarrow{\mu} \mathsf{T}^2\ A$$

datatype T will form a so-called *monad* type, of which $(1+)$ and $(\_)^\star$ are the two examples seen above. Arrow $\mu \cdot \mathsf{T}\ f$ is called the *extension* of f. Functions μ and u are referred to as the monad's *multiplication* and *unit*, respectively. The monadic composition scheme (4.5) is referred to as *Kleisli composition*.

A *monadic arrow* $\mathsf{T}B \xleftarrow{f} A$ conveys the idea of a function which produces an output of "type" B "wrapped by T", where datatype T describes some kind of (computational) "effect". The monad's unit $\mathsf{T}B \xleftarrow{u} B$ is a primitive monadic arrow which injects (*i.e.* promotes, wraps) data *inside* such an effect.

The monad concept is nowadays of primary importance in computing science because it makes it possible to describe computational effects as disparate as input/output, state variable updating, context dependence, partial behaviour (seen above) *etc.* in an elegant, generic and uniform way. Moreover, the monad's operators exhibit notable properties which make it possible to *reason* about such computational effects.

The remainder of this section is devoted to such properties. First of all, the properties implicit in the following diagrams will be *required* for T to be regarded as a monad:

Multiplication :

$$\begin{array}{ccc} \mathsf{T}^2\,A & \xleftarrow{\mu} & \mathsf{T}^3\,A \\ {\scriptstyle\mu}\downarrow & & \downarrow{\scriptstyle\mathsf{T}\,\mu} \\ \mathsf{T}\,A & \xleftarrow[\mu]{} & \mathsf{T}^2\,A \end{array} \qquad \mu \cdot \mu = \mu \cdot \mathsf{T}\,\mu \tag{4.6}$$

Unit :

$$\begin{array}{ccc} \mathsf{T}^2\,A & \xleftarrow{u} & \mathsf{T}\,A \\ {\scriptstyle\mu}\downarrow & \swarrow{\scriptstyle id} & \downarrow{\scriptstyle\mathsf{T}\,u} \\ \mathsf{T}\,A & \xleftarrow[\mu]{} & \mathsf{T}^2\,A \end{array} \qquad \mu \cdot u = \mu \cdot \mathsf{T}\,u = id \tag{4.7}$$

The simple and beautiful symmetries apparent in these diagrams will make it easy to memorize their laws and check them for particular cases. For instance, for the $(1+)$ monad, law (4.7) will read as follows:

$$[i_1\,, id] \cdot i_2 = [i_1\,, id] \cdot (id + i_2) = id$$

These equalities are easy to check.

In laws (4.6) and (4.7), the different instances of μ and u are differently typed, as these are polymorphic and exhibit natural properties:

μ-natural :

$$\begin{array}{ccc} A & \mathsf{T}\,A & \xleftarrow{\mu} & \mathsf{T}^2\,A \\ {\scriptstyle f}\downarrow & {\scriptstyle\mathsf{T}\,f}\downarrow & & \downarrow{\scriptstyle\mathsf{T}^2\,f} \\ B & \mathsf{T}\,B & \xleftarrow[\mu]{} & \mathsf{T}^2\,B \end{array} \qquad \mathsf{T}\,f \cdot \mu = \mu \cdot \mathsf{T}^2\,f \tag{4.8}$$

u-natural :

$$\begin{array}{ccc} A & \mathsf{T}\,A & \xleftarrow{u} & A \\ {\scriptstyle f}\downarrow & {\scriptstyle\mathsf{T}\,f}\downarrow & & \downarrow{\scriptstyle f} \\ B & \mathsf{T}\,B & \xleftarrow[u]{} & B \end{array} \qquad \mathsf{T}\,f \cdot u = u \cdot f \tag{4.9}$$

The simplest of all monads is the *identity monad* $\mathsf{T}\,X \stackrel{\text{def}}{=} X$, which is such that $\mu = id$, $u = id$ and $f \bullet g = f \cdot g$. So — in a sense — the *whole functional discipline* studied thus far was already *monadic*, living inside the simplest of all monads: the identity one. Put in other words, such functional discipline can be framed into a wider discipline in which an arbitrary monad is present. Describing this is the main aim of the current chapter.

4.4.1 Properties involving (Kleisli) composition

The following properties arise from the definitions and monadic properties presented above:

$$f \bullet (g \bullet h) = (f \bullet g) \bullet h \tag{4.10}$$

$$u \bullet f = f = f \bullet u \tag{4.11}$$

$$(f \bullet g) \cdot h = f \bullet (g \cdot h) \tag{4.12}$$

$$(f \cdot g) \bullet h = f \bullet (\mathsf{T}\,g \cdot h) \tag{4.13}$$

$$id \bullet id = \mu \tag{4.14}$$

Properties (4.10) and (4.11) are the monadic counterparts of, respectively, (2.8) and (2.10), meaning that monadic composition preserves the properties of normal functional composition. In fact, for the identity monad, these properties coincide with each other.

Above we have shown that property (4.11) holds for the list monad, recall (4.2). A general proof can be produced similarly. We select property (4.10) as an illustration of the rôle of the monadic properties:

$$\begin{array}{ll} & f \bullet (g \bullet h) \\ = & \quad \{ \text{ definition (4.5) twice } \} \\ & \mu \cdot \mathsf{T}\,f \cdot (\mu \cdot \mathsf{T}\ g \cdot h) \\ = & \quad \{ \ \mu \text{ is natural (4.8) } \} \\ & \mu \cdot \mu \cdot \mathsf{T}^2\,f \cdot \mathsf{T}\ g \cdot h \\ = & \quad \{ \text{ monad property (4.6) } \} \\ & \mu \cdot \mathsf{T}\ \mu \cdot \mathsf{T}^2\,f \cdot \mathsf{T}\ g \cdot h \\ = & \quad \{ \text{ functor } \mathsf{T} \text{ (3.56) } \} \\ & \mu \cdot \mathsf{T}\ (\mu \cdot \mathsf{T}\,f \cdot g) \cdot h \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ definition (4.5) twice } \} \\
& (f \bullet g) \bullet h
\end{array}
$$

Clearly, this calculation generalizes that of exercise 4.1 to any monad T.

Exercise 4.4. *Prove the other laws above and also the following one,*

$$(\mathsf{T}\ f) \cdot (h \bullet k) = (\mathsf{T}\ f \cdot h) \bullet k \tag{4.15}$$

where Kleilsi composition again trades with normal composition.
□

4.5 Monadic application (binding)

We have seen above that, given a monad $A \xrightarrow{u} \mathsf{T}\ A \xleftarrow{\mu} \mathsf{T}^2\ A$, u is the unit of Kleisli composition, $f \bullet u = f$, recall (4.11). Now, what does happen in case we Kleisli compose f with the identity id of *standard* composition? Looking at diagram (4.4) for this case,

$$\begin{array}{ccccc}
\mathsf{T}(\mathsf{T}\ C) & \xleftarrow{\mathsf{T}\ f} & \mathsf{T}\ B & \xleftarrow{id} & \mathsf{T}\ B \\
{\scriptstyle\mu}\downarrow & & \vdots & & \\
\mathsf{T}\ C & \xleftarrow{f} & B & &
\end{array}$$

we realize that $f \bullet id$ accepts a value of type $\mathsf{T}\ B$ that is passed to $\mathsf{T}\ C \xleftarrow{f} B$, yielding an output of type $\mathsf{T}\ C$. This construction is called *binding* and denoted by $\gg\!= f$:

$$(\gg\!= f) = f \bullet id \tag{4.16}$$

Expressed pointwise, we get:[2]

$$x \gg\!= f \ \stackrel{\text{def}}{=}\ (\mu \cdot \mathsf{T}\ f)x \tag{4.17}$$

[2] In the case of the identity monad one has: $x \gg\!= f = f\ x$. So, $\gg\!=$ can be regarded as denoting *monadic function application*.

This operator exhibits properties that arise from its definition and the basic monadic properties, *e.g.*

$$
\begin{array}{ll}
x \gg\!= u & \\
\quad \equiv & \quad \{\ \text{definition (4.17)}\ \} \\
\quad & (\mu \cdot \mathsf{T}\, u)x \\
\quad \equiv & \quad \{\ \text{law (4.7)}\ \} \\
\quad & (id)x \\
\quad \equiv & \quad \{\ \text{identity function}\ \} \\
\quad & x
\end{array}
$$

At pointwise level, one may chain monadic compositions from left to right, *e.g.*

$$(((x \gg\!= f_1) \gg\!= f_2) \gg\!= \ldots f_{n-1}) \gg\!= f_n$$

for functions $A \xrightarrow{f_1} \mathsf{T}\, B_1$, $B_1 \xrightarrow{f_2} \mathsf{T}\, B_2$, $\ldots$ $B_{n-1} \xrightarrow{f_n} \mathsf{T}\, B_n$.

4.6 Sequencing and the do-notation

Recall from above that $x \gg\!= f$ is the monadic *generalization* of function application $f\ x$, since both coincide for the identity monad. Also recall that, for $f = \underline{y}$ (the "everywhere"-y constant function) one gets $\underline{y}\ x = y$.

What does the corresponding monadic generalization, $x \gg\!= \underline{y}$ mean? In the standard notation, this leads to another monadic operator,

$$x \gg y \ \stackrel{\text{def}}{=}\ x \gg\!= \underline{y} \tag{4.18}$$

of type

$$(\gg) : \mathsf{T}\ A \to \mathsf{T}\ B \to \mathsf{T}\ B$$

called "sequencing". For instance, within the finite-list monad, one has

$$[1,2] \gg [3,4] = (\mathsf{concat} \cdot \underline{[3,4]}^{\star})[1,2] = \mathsf{concat}[[3,4],[3,4]] = [3,4,3,4]$$

Because this operator is associative (prove this as an exercise), one may iterate it to more than two arguments and write, for instance,

$$x_1 \gg x_2 \gg \ldots \gg x_n$$

This leads to the popular "**do**-notation", which is another piece of (pointwise) notation which makes sense in a monadic context:

$$\mathbf{do}\ \{x_1; x_2; \ldots; x_n\} \stackrel{\text{def}}{=} x_1 \gg \mathbf{do}\ \{x_2; \ldots; x_n\}$$

for $n \geqslant 1$. For $n = 1$ one trivially has

$$\mathbf{do}\ x_1 \stackrel{\text{def}}{=} x_1$$

4.7 Generators and comprehensions

The monadic **do**-notation paves the way to a device that is very useful in (pointwise) monadic programming. As before, we consider its (non-monadic) counterpart first. Consider for instance the expression $x + \mathsf{sum}\ y$, where sum is some operator in some context, e.g. adding up all elements of a list. Nothing impedes us from "structuring" expression $x + \mathsf{sum}\ y$ in the following way:

$$\begin{array}{l}\mathbf{let}\ a = \mathsf{sum}\ y\\ \mathbf{in}\ x + a\end{array}$$

It turns out that the above is the same as the following monadic expression,

$$\begin{array}{l}\mathbf{do}\ \{\\ \quad a \leftarrow \mathsf{sum}\ y;\\ \quad u\ (x + a)\}\end{array}$$

provided the underlying monad is the *identity* monad. Now, what does the notation $a \leftarrow \ldots$ mean for an arbitrary monad $A a \mathrel{-}\!\!^{u}\!\!\longrightarrow \mathsf{T}\ A \xleftarrow{\mu} \mathsf{T}^2\ A$?

The **do**-notation accepts a variant in which the arguments of the $\gg$ operator are "generators" of the form

$$a \leftarrow x \tag{4.19}$$

where, for a of type A, x is an inhabitant of monadic type $\mathsf{T}\ A$. One may regard $a \leftarrow x$ as meaning "let a be taken from x". Then the **do**-notation unfolds as follows:

$$\mathbf{do}\ a \leftarrow x_1; x_2; \ldots; x_n \stackrel{\text{def}}{=} x_1 \gg\!= \lambda a \cdot (\mathbf{do}\ x_2; \ldots; x_n) \tag{4.20}$$

Of course, we should now allow for the x_i to range over terms involving variable a. For instance, by writing (again in the list-monad)

$$\mathbf{do}\ a \leftarrow [1,2,3]; [\,a^2\,] \tag{4.21}$$

we mean

$$\begin{array}{rcl} & [1,2,3] \gg\!= \lambda a.[a^2] \\ = & \mathsf{concat}((\lambda a.[a^2])^\star [1,2,3]) \\ = & \mathsf{concat}[[1],[4],[9]] \\ = & [1,4,9] \end{array}$$

The analogy with classical set-theoretic ZF-notation, whereby one might write $\{a^2 \mid a \in \{1,2,3\}\}$ to describe the set of the first three perfect squares, calls for the following notation,

$$[\ a^2 \mid a \leftarrow [1,2,3]\ ] \tag{4.22}$$

as a "shorthand" of (4.21). This is an instance of the so-called *comprehension* notation, which can be defined in general as follows:

$$[\ e \mid a_1 \leftarrow x_1, \ldots, a_n \leftarrow x_n\ ] \quad = \quad \mathbf{do}\ \{a_1 \leftarrow x_1; \ldots; an \leftarrow x_n; u\ e\} \tag{4.23}$$

where u is the monad's unit (4.7,4.9).

Alternatively, comprehensions can be defined as follows, where p, q stand for arbitrary generators:

$$[t] \quad = \quad u\,t \tag{4.24}$$

$$[\ f\,x \mid x \leftarrow l\ ] \quad = \quad (\mathsf{T}\,f)l \tag{4.25}$$

$$[\ t \mid p, q\ ] \quad = \quad \mu[\ [\ t \mid q\ ] \mid p\ ] \tag{4.26}$$

Note, however, that comprehensions are not restricted to lists or sets — they can be defined for any monad T thanks to the $\mathbf{do}$-notation.

Exercise 4.5. *Show that*

$$(f \bullet g)\ a \quad = \quad \mathbf{do}\ \{b \leftarrow g\ a; f\ b\} \tag{4.27}$$

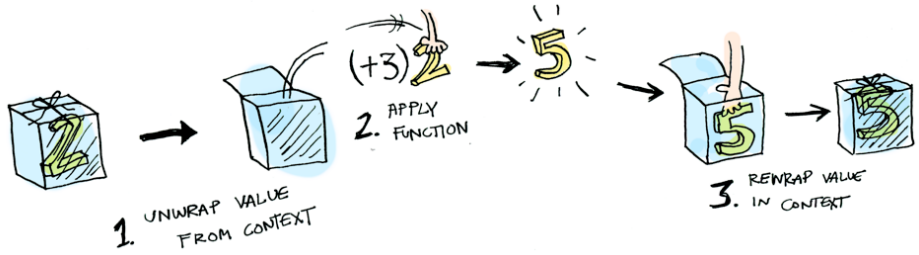
$$\mathsf{T}\,f\ x \quad = \quad \mathbf{do}\ \{a \leftarrow x; u\ (f\ x)\} \tag{4.28}$$

Note that the second $\mathbf{do}$ *expression is equivalent to* $x \gg\!= (u \cdot f)$.

□

Exercise 4.6. *Show that* $x \gg\!= f$ = **do** $\{ a \leftarrow x; f\ a \}$ *and then that* $(x \gg\!= g) \gg\!= f$ *is the same as* $x \gg\!= f \bullet g$.
□

Fact (4.28) is illustrated in the cartoon[3]



for the computation of $\mathsf{T}\ (+3)\ x$, where $x = u\ 2$ is the T-monadic object containing number 2.

4.8 Monads in HASKELL

In the *Standard Prelude* for HASKELL, one finds the following minimal definition of the $Monad$ class,

$$\begin{array}{l}\textbf{class}\ Monad\ m\ \textbf{where} \\ \quad \mathsf{return} :: a \to m\ a \\ \quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b\end{array}$$

where return refers to the unit of m, on top of which the "sequence" operator

$$\begin{array}{l}(\gg) :: m\ a \to m\ b \to m\ b \\ fail\ :: String \to m\ a\end{array}$$

is defined by

[3]Credits: see this and other helpful, artistic illustrations in
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html.

$$p \gg q = p \gg\!= \lambda\_ \rightarrow q$$

as expected. This class is instantiated for finite sequences ($[\,]$), $Maybe$ and IO, among others.

The μ multiplication operator is function join in module `Monad.hs`:

$$\begin{aligned}&\mathsf{join} :: (Monad\ m) \Rightarrow m\ (m\ a) \rightarrow m\ a\\ &\mathsf{join}\ x = x \gg\!= id\end{aligned}$$

This is easily justified:

$$\begin{aligned}\mathsf{join}\ x &= x \gg\!= id \qquad (4.29)\\ &= \quad \{\ \text{definition (4.17)}\ \}\\ &\quad (\mu \cdot \mathsf{T}\ id)x\\ &= \quad \{\ \text{functors commute with identity (3.55)}\ \}\\ &\quad (\mu \cdot id)x\\ &= \quad \{\ \text{law (2.10)}\ \}\\ &\quad \mu\, x\end{aligned}$$

The following infix notation for (Kleisli) monadic composition in HASKELL uses the binding operator:

$$\begin{aligned}&(\bullet) :: Monad\ t \Rightarrow (b \rightarrow t\ c) \rightarrow (d \rightarrow t\ b) \rightarrow d \rightarrow t\ c\\ &(f \bullet g)\ a = (g\ a) \gg\!= f\end{aligned}$$

Exercise 4.7. *Consider the* HASKELL *function*

$$\begin{aligned}&discollect :: [(a,[b])] \rightarrow [(a,b)]\\ &discollect\ [\,] = [\,]\\ &discollect\ ((a,x) : y) = [(a,b) \mid b \leftarrow x] +\!\!+ discollect\ y\end{aligned}$$

Knowing that finite lists form a monad where $\mu = \mathsf{concat} = (\!|[\mathsf{nil}, \mathsf{conc}]|\!)$ *and* $\mathsf{conc}\ (x,y) = x +\!\!+ y$*, derive the above pointfree code from the definition*

$$discollect = lstr \bullet id \qquad (4.30)$$

where $lstr\ (a,x) = [(a,b) \mid b \leftarrow x]$.

□

4.8.1 Monadic I/O

IO, a parametric datatype whose inhabitants are special values called *actions* or *commands*, is a most relevant monad. Actions perform the interconnection between HASKELL and the environment (file system, operating system). For instance,

$$getLine :: \mathsf{IO}\ String$$

is a particular such action. Parameter $String$ refers to the fact that this action "delivers" — or extracts — a string from the environent. This meaning is clearly conveyed by the type $String$ assigned to symbol l in

$$\mathbf{do}\ l \leftarrow getLine; \ldots l \ldots$$

which is consistent with typing rule for generators (4.19). Sequencing corresponds to the ";" syntax in most programming languages (*e.g.* C) and the **do**-notation is particulary intuitive in the IO-context.

Examples of functions delivering actions are

$$FilePath \xrightarrow{readFile} \mathsf{IO}\ String$$

and

$$Char \xrightarrow{putChar} \mathsf{IO}\ ()$$

— both produce I/O commands as result.

As is to be expected, the implementation of the IO monad in HASKELL — available from the *Standard Prelude* — is not totally visible, for it is bound to deal with the intrincacies of the underlying machine:

$$\begin{array}{l}\mathbf{instance}\ Monad\ \mathsf{IO}\ \mathbf{where}\\ \quad (\gg\!=) = primbindIO\\ \quad \mathsf{return} = primretIO\end{array}$$

Rather interesting is the way IO is regarded as a functor:

$$fmap\ f\ x = x \gg\!= (\mathsf{return} \cdot f)$$

This goes the other way round, the monadic structure "helping" in defining the functor structure, everything consistent with the underlying theory:

$$\begin{aligned} x \gg\!= (u \cdot f) &= (\mu \cdot \mathsf{IO}(u \cdot f))x \\ &= \quad \{ \text{ functors commute with composition } \} \\ &\quad (\mu \cdot \mathsf{IO}\, u \cdot \mathsf{IO}\, f)x \\ &= \quad \{ \text{ law (4.7) for } \mathsf{T} = \mathsf{IO} \ \} \\ &\quad (\mathsf{IO}\, f)x \\ &= \quad \{ \text{ definition of } fmap \ \} \\ &\quad (fmap\ f)\ x \end{aligned}$$

For enjoyable reading on monadic input/output in HASKELL see [18], chapter 18.

Exercise 4.8. *Extend the Maybe monad to the following "error message" exception handling datatype:*

$$\mathbf{data}\ Error\ a = Err\ String \mid Ok\ a\ \mathbf{deriving}\ Show$$

In case of several error messages issued in a `do` *sequence, how many turn up on the screen? Which ones?*
□

Exercise 4.9. *Recalling section 3.13, show that any inductive type with base functor*

$$\mathsf{B}\ (f, g) = f + \mathsf{F}\ g$$

where F *is an arbitrary functor, forms a monad for*

$$\mu = (\!|[id, \mathsf{in} \cdot i_2]|\!)$$
$$u = \mathsf{in} \cdot i_1.$$

Identify F *for known monads such as eg.* Maybe, LTree *and (non-empty) lists.*
□

4.9 The state monad

The so-called *state monad* is a monad whose inhabitants are state-transitions encoding a particular brand of state-based automata known as *Mealy machines*. Given a set A (input alphabet), a set B (output alphabet) and a set of states S, a deterministic Mealy machine (DMM) is specified by a transition function of type

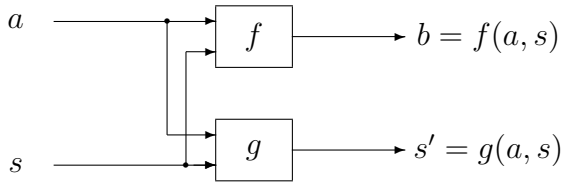
$$A \times S \xrightarrow{\delta} B \times S \tag{4.31}$$

Wherever $(b, s') = \delta(a, s)$, we say that the machine has transition

$$s \xrightarrow{a|b} s'$$

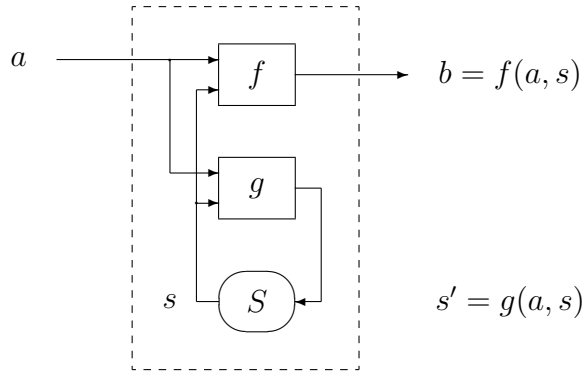
and refer to s as the *before* state, and to s' as the *after* state. Many programs that one writes in conventional programming languages such as C or Java can be regarded as DMMs.

It is clear from (4.31) that δ can be expressed as the *split* of two functions f and g — $\delta = \langle f, g \rangle$ — as depicted in the following drawing:



Note, however, that the information recorded in the state of a DMM is either meaningless to the user of the machine (as in eg. the case of states represented by numbers) or too complex to be perceived and handled explicitly (as is the case of eg. the data kept in a large database). So, it is convenient to *abstract* from it, via the "encapsulation" suggested by the following, transformed, version of the

previous drawing,



(4.32)

in which the state is no longer accessible from the outside.

Such an abstraction is nicely captured by the so-called *state monad*, in the following way: taking (4.31) and recalling (2.91), we simply transpose (ie. *curry*) δ and obtain

$$A \xrightarrow{\overline{\delta}} \underbrace{(B \times S)^S}_{(\mathsf{St}\ S)\ B} \tag{4.33}$$

thus "shifting" the *input* state to the *output*. In this way, $\overline{\delta}\, a$ is a function capturing all state-transitions (and corresponding outputs) for input a. For instance, the function that *appends* a new element at the rear of a queue,

$$enq(a, s) \stackrel{\text{def}}{=} s +\!\!+ [a]$$

can be converted into a DMM by adding to it a dummy output of type 1 and then transposing:

$$\begin{array}{rcl} enqueue & : & A \to (1 \times S)^S \\ enqueue\ a & \stackrel{\text{def}}{=} & \langle !, (+\!\!+[a]) \rangle \end{array} \tag{4.34}$$

Action $enqueue$ performs enq on the state while acknowledging it by issuing an output of type 1.[4]

[4] A kind of "done!" message.

Unit and multiplication. Let us now show that

$$(\mathsf{St}\ S)\ A \quad \cong \quad (A \times S)^S \tag{4.35}$$

forms a monad. As we shall see, the fact that the *values* of this monad are functions brings the theory of exponentiation to the forefront. Thus, a review of section 2.15 is recommended at this point.

Notation $\widehat{f}$ will be used to abbreviate $uncurry\ f$, enabling the following variant of universal law (2.83),

$$\widehat{k} = f \quad \Leftrightarrow \quad f = ap \cdot (k \times id) \tag{4.36}$$

whose cancellation

$$\widehat{k} = ap \cdot (k \times id) \tag{4.37}$$

is written pointwise as follows:

$$\widehat{k}(c, a) \quad = \quad (k\ c)a \tag{4.38}$$

First of all, what is the functor behind (4.35)? Fixing the state space S, we obtain

$$\mathsf{T}X \quad \stackrel{\text{def}}{=} \quad (X \times S)^S \tag{4.39}$$

on objects and

$$\mathsf{T}f \quad \stackrel{\text{def}}{=} \quad (f \times id)^S \tag{4.40}$$

on functions, where $(\_)^S$ is the exponential functor (2.87).

The unit of this monad is the transpose of the simplest of all Mealy machines — the identity:

$$\begin{array}{lcl} u & : & A \to (A \times S)^S \\ u & = & \overline{id} \end{array} \tag{4.41}$$

Let us see what this means:

$$\begin{array}{ll} & u = \overline{id} \\ \equiv & \quad \{\ (2.83)\ \} \\ & ap \cdot (u \times id) = id \\ \equiv & \quad \{\ \text{introducing variables}\ \} \\ & ap(u\ a, s) = (a, s) \\ \equiv & \quad \{\ \text{definition of } ap\ \} \\ & (u\ a)s = (a, s) \end{array}$$

So, action $u\ a$ performed on state s keeps s unchanged and outputs a.

From the type of μ, for this monad,

$$((A \times S)^S \times S)^S \xrightarrow{\mu} (A \times S)^S$$

one figures out $\mu = x^S$ (recalling the exponential functor as defined by (2.87)) for some $((A \times S)^S \times S) \xrightarrow{x} (A \times S)$. This, on its turn, is easily recognized as an instance of the ap polymorphic function (2.83), which is such that $ap = \widehat{id}$, recall (2.85). Altogether, we define

$$\mu \ = \ ap^S \tag{4.42}$$

Let us inspect the behaviour of μ by checking the meaning of applying it to an action expressed as in diagram (2.91):

$$\begin{array}{ll} & \mu\langle f, g\rangle = ap^S\langle f, g\rangle \\ \equiv & \quad \{\ (2.87)\ \} \\ & \mu\langle f, g\rangle = ap \cdot \langle f, g\rangle \\ \equiv & \quad \{\ \text{extensional equality (2.5)}\ \} \\ & \mu\langle f, g\rangle s = ap(f\ s, g\ s) \\ \equiv & \quad \{\ \text{definition of } ap\ \} \\ & \mu\langle f, g\rangle s = (f\ s)(g\ s) \end{array}$$

We find out that μ "unnests" the action inside f by applying it to the state delivered by g.

Checking the monadic laws. The calculation of (4.7) is made in two parts, checking $\mu \cdot u = id$ first,

$$\begin{array}{ll} & \mu \cdot u \\ = & \quad \{\ \text{definitions}\ \} \\ & ap^S \cdot \overline{id} \\ = & \quad \{\ \text{exponentials absorption (2.88)}\ \} \\ & \overline{ap \cdot id} \end{array}$$

$$
\begin{array}{ll}
= & \quad \{ \text{ reflection (2.85) } \} \\
& id \\
\Box &
\end{array}
$$

and then checking $\mu \cdot (\mathsf{T}u) = id$:

$$
\begin{array}{ll}
& \mu \cdot (\mathsf{T}u) \\
= & \quad \{ \text{ (4.42,4.40) } \} \\
& ap^S \cdot (\overline{id} \times id)^S \\
= & \quad \{ \text{ functor } \} \\
& (ap \cdot (\overline{id} \times id))^S \\
= & \quad \{ \text{ cancellation (2.84) } \} \\
& id^S \\
= & \quad \{ \text{ functor } \} \\
& id \\
\Box &
\end{array}
$$

The proof of (4.6) is also not difficult once supported by the laws of exponentials.

Kleisli composition. Let us calculate $f \bullet g$ for this monad:

$$
\begin{array}{ll}
& f \bullet g \\
= & \quad \{ \text{ (4.5) } \} \\
& \mu \cdot \mathsf{T}\, f \cdot g \\
= & \quad \{ \text{ (4.42) ; (4.40) } \} \\
& ap^S \cdot (f \times id)^S \cdot g \\
= & \quad \{ \text{ } (\_)^S \text{ is a functor } \} \\
& (ap \cdot (f \times id))^S \cdot g \\
= & \quad \{ \text{ (4.36) } \} \\
& \widehat{f}^S \cdot g \\
= & \quad \{ \text{ cancellation } \}
\end{array}
$$

$$
\begin{array}{ll}
 & \widehat{f}^S \cdot \overline{\widehat{g}} \\
= & \quad \{ \text{ absorption (2.88) } \} \\
 & \overline{\widehat{f} \cdot \widehat{g}}
\end{array}
$$

In summary, we have:

$$f \bullet g \;=\; \overline{\widehat{f} \cdot \widehat{g}} \tag{4.43}$$

which can be written alternatively as

$$\widehat{f \bullet g} \;=\; \widehat{f} \cdot \widehat{g}$$

Let us use this in calculating law

$$pop \bullet push \;=\; u \tag{4.44}$$

where $push$ and pop are such that

$$\begin{array}{lcl} push & : & A \to (1 \times S)^S \\ \widehat{push} & \stackrel{\text{def}}{=} & \langle !, \widehat{(:)} \rangle \end{array} \tag{4.45}$$

$$\begin{array}{lcl} pop & : & 1 \to (A \times S)^S \\ \widehat{pop} & \stackrel{\text{def}}{=} & \langle head, tail \rangle \cdot \pi_2 \end{array} \tag{4.46}$$

for S the datatype of finite lists. We reason:

$$
\begin{array}{ll}
 & pop \bullet push \\
= & \quad \{ \text{ (4.43) } \} \\
 & \overline{\widehat{pop} \cdot \widehat{push}} \\
= & \quad \{ \text{ (4.45, 4.46) } \} \\
 & \overline{\langle head, tail \rangle \cdot \pi_2 \cdot \langle !, \widehat{(:)} \rangle} \\
= & \quad \{ \text{ (2.22, 2.26) } \} \\
 & \overline{\langle head, tail \rangle \cdot \widehat{(:)}} \\
= & \quad \{ \text{ } out \cdot in = id \text{ (lists) } \} \\
 & \overline{id} \\
= & \quad \{ \text{ (4.41) } \} \\
 & u
\end{array}
$$

□

Bind. The effect of binding a state transition x to a state-monadic function h is calculated in a similar way:

$$
\begin{array}{ll}
& x \gg\!= h \\
= & \quad \{ \text{ (4.17) } \} \\
& (\mu \cdot \mathsf{T}h)x \\
= & \quad \{ \text{ (4.42) and (4.40) } \} \\
& (ap^S \cdot (h \times id)^S)x \\
= & \quad \{ \ (\_)^S \text{ is a functor } \} \\
& (ap \cdot (h \times id))^S x \\
= & \quad \{ \text{ cancellation (4.37) } \} \\
& \widehat{h}^S x \\
= & \quad \{ \text{ exponential functor (2.87) } \} \\
& \widehat{h} \cdot x
\end{array}
$$

Let us unfold $\widehat{h} \cdot x$ by splitting x into its components two components f and g:

$$
\begin{array}{ll}
& \langle f, g\rangle \gg\!= h \ = \ \widehat{h} \cdot \langle f, g\rangle \\
\equiv & \quad \{ \text{ go pointwise } \} \\
& (\langle f, g\rangle \gg\!= h)s \ = \ \widehat{h}(\langle f, g\rangle s) \\
\equiv & \quad \{ \text{ (2.20) } \} \\
& (\langle f, g\rangle \gg\!= h)s \ = \ \widehat{h}(f\ s, g\ s) \\
\equiv & \quad \{ \text{ (4.38) } \} \\
& (\langle f, g\rangle \gg\!= h)s \ = \ h(f\ s)(g\ s)
\end{array}
$$

In summary, for a given "before state" s, $g\ s$ is the intermediate state upon which $f\ s$ runs and yields the output and (final) "after state".

Two prototypical inhabitants of the state monad: get **and** put**.** These generic actions are defined as follows, in the PF-style:

$$get \stackrel{\text{def}}{=} \langle id, id\rangle \tag{4.47}$$

$$put \stackrel{\text{def}}{=} \overline{\langle !, \pi_1 \rangle} \tag{4.48}$$

Action g retrieves the data stored in the state without changing it, while put stores a particular value in the state. Note that put can also be written

$$put\ s = \langle !, \underline{s} \rangle \tag{4.49}$$

or even as

$$put\ s = update\ \underline{s} \tag{4.50}$$

where

$$update\ f \stackrel{\text{def}}{=} \langle !, f \rangle \tag{4.51}$$

updates the state via state-to-state function f.

The following is an example, written in Haskell, of the standard use of get/put in managing context data, in this case a counter. Function $decBTree$ decorates each node of a $BTree$ (recall this datatype from page 108) with its position in the tree:

$$\begin{array}{l} decBTree\ Empty = \mathsf{return}\ Empty \\ decBTree\ (Node\ (a, (t1, t2))) = \mathbf{do}\ \{ \\ \quad n \leftarrow get; \\ \quad put\ (n+1); \\ \quad x \leftarrow decBTree\ t1; \\ \quad y \leftarrow decBTree\ t2; \\ \quad \mathsf{return}\ (Node\ ((a, n), (x, y))) \\ \quad \} \end{array}$$

To close the chapter, we will present a strategy for deriving this kind of monadic functions.

4.10 'Monadification' of Haskell code made easy

There is an easy roadmap for "monadification" of Haskell code. What do we mean by *monadification*? Well, in a sense — as we shall soon see — every piece of code is monadic: we don't notice this because the underlying monad is *invisible* (the *identity* monad). We are going to see how to make it visible taking advantage of monadic `do` notation and leaving it open for instantiation. This will bridge the

gap between monads' theory and its application to handling particular effects in concrete programming situations.

Let us take as starting point the pointwise version of sum, the list catamorphism that adds all numbers found in its input:

$$\begin{array}{l}\mathsf{sum}\ [\,] = 0\\ \mathsf{sum}\ (h:t) = h + \mathsf{sum}\ t\end{array}$$

Notice that this code could have been written as follows

$$\begin{array}{l}\mathsf{sum}\ [\,] = id\ 0\\ \mathsf{sum}\ (h:t) = \mathbf{let}\ x = \mathsf{sum}\ t\ \mathbf{in}\ id\ (h + x)\end{array}$$

using let notation and two instances of the identity function. Question: what is the point of such a "baroque" version of the starting, so simple piece of code? Answer:

- The let ... in ... notation stresses the fact that recursive call happens *earlier* than the delivery of the result.
- The id functions signal the exit points of the algorithm, that is, the points where it *returns* something to the caller.

Next, let us

- re-write id into return—;
- re-write let x = ... in ...— into `do { x <- ... ; ... }`

One will obtain the following version of sum:

$$\begin{array}{l}msum\ [\,] = \mathsf{return}\ 0\\ msum\ (h:t) = \mathbf{do}\ \{x \leftarrow msum\ t; \mathsf{return}\ (h + x)\}\end{array}$$

Typewise, while sum has type $(Num\ a) \Rightarrow [a] \rightarrow a$, $msum$ has type

$$(Monad\ m, Num\ a) \Rightarrow [a] \rightarrow m\ a$$

That is, $msum$ is monadic — parametric on monad m — while sum is not.

There is a particular monad for which sum and $msum$ coincide: the **identity** monad $\mathsf{Id}\ X = X$. It is very easy to show that inside this monad return is the identity and $\mathbf{do}\ x \leftarrow \ldots$ means the same as $\mathbf{let}\ x = \ldots$, as already mentioned —

enough for the pointwise versions of the two functions to be the same. Thus the "invisible" monad mentioned earlier is the identity monad.

In summary, the monadic version is *generic* in the sense that it runs on whatever monad you like, enabling you to perform *side effects* while the code runs. If you don't need any effects then you get the "non-monadic" version as special case, as seen above. Otherwise, Haskell will automatically switch to the effects you want, depending on the monad you choose (often determined by context).

For each particular monad we may decide to add specific monadic code like `get` and `put` in the `decBTree` example, where we want to take advantage of the state monad. As another example, check the following enrichment of $msum$ with state-monadic code helping you to trace the execution of your program:

$$
\begin{array}{l}
msum'\ [\,] = \mathsf{return}\ 0 \\
msum'\ (h : t) = \\
\quad \mathbf{do}\ \{x \leftarrow msum'\ t; \\
\qquad print\ (\texttt{"x= "} +\!\!+ show\ x); \\
\qquad \mathsf{return}\ (h + x)\}
\end{array}
$$

Thus one obtains traces of the code in the way prescribed by the particular usage of the $print$ (state monadic) function:

```
*Main> msum' [3,5,1,3,4]
"x= 0"
"x= 4"
"x= 7"
"x= 8"
"x= 13"
*Main>
```

In the reverse direction, one may try and see what happens to monadic code upon removing all monad-specific functions and going into the identity monad once it gets monad generic. In the case of `decBTree`, for instance, we will get

$$
\begin{array}{l}
decBTree\ Empty = \mathsf{return}\ Empty \\
decBTree\ (Node\ (a, (t1, t2))) = \\
\quad \mathbf{do} \\
\qquad x \leftarrow decBTree\ t1; \\
\qquad y \leftarrow decBTree\ t2; \\
\qquad \mathsf{return}\ (Node\ (a, (x, y)))
\end{array}
$$

once `get` and `put` are removed (and therefore all instances of `n`), and then

$$
\begin{array}{l}
decBTree\ Empty = Empty \\
decBTree\ (Node\ (a, (t1, t2))) = \\
\quad \textbf{let} \\
\qquad x = decBTree\ t1 \\
\qquad y = decBTree\ t2 \\
\quad \textbf{in}\ Node\ (a, (x, y))
\end{array}
$$

This is the identity function on type BTree, recall the cata-reflection law (3.68). So, the *archetype* of (inspiration for) much monadic code is the most basic of all tree traversal functions — the identity [5]. The same could be said about imperative code of a particular class — the *recursive descent* one — much used in compiler construction, for instance.

Playing with effects

As it may seem from the previous examples, adding effects to produce monadic code is far from arbitrary. This can be further appreciated by defining the function that yields the smallest element of a list,

$$
\begin{array}{l}
getmin\ [a] = a \\
getmin\ (h : t) = min\ h\ (getmin\ t)
\end{array}
$$

which is incomplete in the sense that it does not specify the meaning of $getmin\ [\,]$. As this is mathematically undefined, it should be expressed "outside the maths", that is, as an effect. Thus, to complete the defintion we first go monadic, as we did before,

$$
\begin{array}{l}
mgetmin\ [a] = \mathsf{return}\ a \\
mgetmin\ (h : t) = \textbf{do}\ \{x \leftarrow mgetmin\ t; \mathsf{return}\ (min\ h\ x)\}
\end{array}
$$

and then chose a monad in which to express the meaning of $getmin\ [\,]$, for instance the `Maybe` monad

$$
\begin{array}{l}
mgetmin\ [\,] = \mathsf{Nothing} \\
mgetmin\ [a] = \mathsf{return}\ a \\
mgetmin\ (h : t) = \textbf{do}\ \{x \leftarrow mgetmin\ t; \mathsf{return}\ (min\ h\ x)\}
\end{array}
$$

[5] We have seen the same kind of "inspiration" before in building type functors (3.76) which, for $f = id$, boil down to the identity.

Alternatively, we might have written

$$mgetmin\ [\,] = \mathit{Error}\ \texttt{"Empty input"}$$

going into the `Error` monad, or even the simpler (yet interesting) $mgetmin\ [\,] = [\,]$, which shifts the code into the list monad, yielding singleton lists in the success case, otherwise the empty list.

Function $getmin$ above is an example of a partial function, that is, a function which is undefined for some of its inputs.[6] These functions cause much interference in functional programming, which monads help us to keep under control.

Let us see how such interference is coped with in the case of higher order functions, taking map as example

$$\begin{aligned}&\mathsf{map}\ f\ [\,] = [\,]\\ &\mathsf{map}\ f\ (h : t) = (f\ h) : \mathsf{map}\ f\ t\end{aligned}$$

and supposing f is not a total function. How do we cope with erring evaluations of $f\ h$? As before, we first "letify" the code,

$$\begin{aligned}&\mathsf{map}\ f\ [\,] = [\,]\\ &\mathsf{map}\ f\ (h : t) = \mathbf{let}\\ &\quad b = f\ h\\ &\quad x = \mathsf{map}\ f\ t\ \mathbf{in}\ b : x\end{aligned}$$

we go monadic in the usual way,

$$\begin{aligned}&mmap\ f\ [\,] = \mathsf{return}\ [\,]\\ &mmap\ f\ (h : t) = \mathbf{do}\ \{\, b \leftarrow f\ h; x \leftarrow mmap\ f\ t; \mathsf{return}\ (b : x)\,\}\end{aligned}$$

and everything goes smoothly — as can be checked, the function thus built is of the expected (monadic) type:

$$mmap :: (Monad\ \mathsf{T}) \Rightarrow (a \to \mathsf{T}\ b) \to [a] \to \mathsf{T}\ [b] \tag{4.52}$$

Run $mmap$ Just $[1, 2, 3, 4]$, for instance: you will obtain Just $[1, 2, 3, 4]$. Now run $mmap\ print\ [1, 2, 3, 4]$. You will see the items in the sequence printed sequentially.

[6] Recall that function partiality was our motivation for studying monads right from the beginning of this chapter.

One may wonder about the behaviour of the $mmap$ for f the identity function: will we get an error? No, we get a well-typed function of type $[m\ a] \to m\ [a]$, for m a monad. We thus obtain the well-known monadic function sequence which evaluates each *action* in the input sequence, from left to right, collecting the results. For instance, applying this function to input sequence $[\mathsf{Just}\ 1, \mathsf{Nothing}, \mathsf{Just}\ 2]$ the output will be Nothing.

Exercise 4.10. *Use the* monadification *technique to encode monadic function*

$$filterM :: Monad\ m \Rightarrow (a \to m\ \mathbb{B}) \to [a] \to m\ [a]$$

which generalizes the list-based filter function.
□

Exercise 4.11. *"Reverse" the following monadic code into its non-monadic archetype:*

```
f :: (Monad m) ⇒ (a → m B) → [a] → m [a]
f p [] = return []
f p (h : t) = do {
   b ← p h;
   t' ← f p t;
   return (if b then h : t' else [])
   }
```

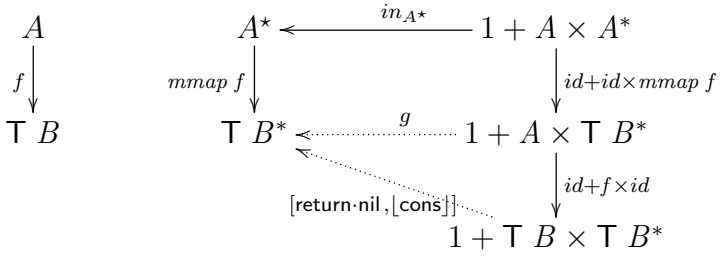
Which function of the Haskell Prelude do you get by such reverse monadification*?*
□

4.11 Monadic recursion

There is much more one could say about monadic recursive programming. In particular, one can express the code "monadification" strategies of the previous section in terms of catamorphisms. As an example, recall (4.52):

$$\begin{array}{ccc} A & & A^\star \xleftarrow{in_{A^\star}} 1 + A \times A^* \\ {\scriptstyle f}\downarrow & & {\scriptstyle mmap\ f}\downarrow \qquad\qquad \downarrow{\scriptstyle id+id\times mmap\ f} \\ \mathsf{T}\ B & & \mathsf{T}\ B^* \xleftarrow{\quad g \quad} 1 + A \times \mathsf{T}\ B^* \end{array}$$

How do we build g? Clearly, the recipe given by (3.76) needs to be adapted:



where

$$\lfloor f \rfloor\ (x, y) = \mathbf{do}\ \{\, a \leftarrow x; b \leftarrow y; \mathsf{return}\ (f\ (a, b))\,\}$$

By defining

$$\begin{aligned}&(\!|g|\!)^\flat = (\!|[\mathsf{return} \cdot f\,, \lfloor h \rfloor]|\!)\ \mathbf{where}\\ &\quad f = (g \cdot i_1)\\ &\quad h = (g \cdot i_2)\end{aligned}$$

we can write

$$mmap\ f = (\!|(\mathsf{in} \cdot (id + f \times id))|\!)^\flat \tag{4.53}$$

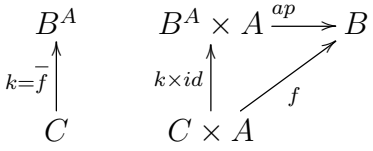
where (recall) $\mathsf{in} = [\mathsf{nil}, \mathsf{cons}]$.

Handling monadic recursion in full generality calls for technical ingredients called *commutative laws* which fall outside the current scope of this chapter.

4.12 Where do monads come from?

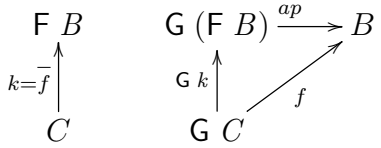
In the current context, a good way to find an answer this question is to recall the universal property of exponentials (2.83):

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



Let us re-draw this diagram by unfolding $B^A \times A$ into the composition of two functors $\mathsf{G}\ (\mathsf{F}\ B)$ where $\mathsf{F}\ X = X^A$ and $\mathsf{G}\ X = X \times A$:

$$k = \overline{f}\ \Leftrightarrow\ f = \underbrace{ap \cdot \mathsf{G}\ k}_{\widehat{k}}$$



As we already know, this establishes the $(curry/uncurry)$ isomorphism

$$\mathsf{G}\ C \to B \ \cong\ C \to \mathsf{F}\ B \tag{4.54}$$

assuming F and G as defined above.

Note how (4.54) expresses a kind of "shunting rule" at type level: Gs on the input side can be "shunted" to the output if replaced by Fs. This is exactly what $curry$ and $uncurry$ do typewise. The corollaries of the universal property can also be expressed in terms of F and G:

- Reflection: $\overline{ap} = id$, that is, $ap = \widehat{id}$ – recall (2.85)
- Cancellation: $\widehat{id} \cdot \mathsf{G}\ \overline{f} = f$ – recall (2.84)
- Fusion: $\overline{h} \cdot g = \overline{h \cdot \mathsf{G}\ g}$ — recall (2.86)
- Absorption: $(\mathsf{F}\ g) \cdot \overline{h} = \overline{g \cdot h}$ — recall (2.88)
- Naturality: $h \cdot \overline{id} = \overline{id} \cdot \mathsf{G}\ (\mathsf{F}\ h)$
- Functor: $\mathsf{F}\ h = \overline{h \cdot ap}$
- Closed definitions: $\widehat{k} = ap \cdot (\mathsf{G}\ k)$ and $\overline{g} = (\mathsf{F}\ g) \cdot \overline{id}$, the latter following from absorption.

Now observe what happens if the functor composition $\mathsf{G}\cdot\mathsf{F}$ is swapped: $\mathsf{F}\ (\mathsf{G}\ X) = (X \times A)^A$. We get the *state monad* out of this construction,

$$(\mathsf{G} \cdot \mathsf{F})\ X = (X \times A)^A = St\ A\ X$$

— recall (4.35). Interestingly, the same universal property can be expressed in terms of such a monad structure, as the simple calculation shows:

$$\begin{array}{ll} & k = \overline{f}\ \Leftrightarrow\ ap \cdot \mathsf{G}\ k = f \\ \equiv & \quad \{\ \text{see above}\ \} \\ & k = (\mathsf{F}\ f) \cdot \overline{id}\ \Leftrightarrow\ f = \widehat{k} \\ \equiv & \quad \{\ \text{swapping variables } k \text{ and } f\text{, to match the starting diagram}\ \} \\ & f = (\mathsf{F}\ k) \cdot \overline{id}\ \Leftrightarrow\ k = \widehat{f} \\ \Box & \end{array}$$

That is,

$$k = \widehat{f} \;\Leftrightarrow\; f = \underbrace{\mathsf{F}\ k \cdot \eta}_{\overline{k}} \qquad \begin{array}{c} \mathsf{G}\ B \\ {\scriptstyle k=\widehat{f}}\downarrow \\ C \end{array} \qquad \begin{array}{ccc} \mathsf{F}\ (\mathsf{G}\ B) & \xleftarrow{\eta} & B \\ {\scriptstyle \mathsf{F}\ k}\downarrow & \swarrow_{f} & \\ \mathsf{F}\ A & & \end{array}$$

for $\eta = \overline{id}$, the unit of the monad $\mathsf{T} = \mathsf{F} \cdot \mathsf{G}$. To complete the definition of the T monad in this way, we recall (4.42)

$$\mu = \mathsf{F}\ \widehat{id} \tag{4.55}$$

with type $(\mathsf{T} \cdot \mathsf{T})\ X \xrightarrow{\mu} \mathsf{T}\ X$, where $id : \mathsf{T}\ X \to \mathsf{T}\ X$.

Adjunctions

The reasoning we have made above for exponentials and the state monad generalizes for any other monad. In general, isomorphisms of shape (4.54) are called an *adjunction* of the two functors F and G, which are said to be *adjoint* to each other. One writes $\mathsf{G} \dashv \mathsf{F}$ and says that G is *left* adjoint and that F is *right* adjoint. Using notation $\lfloor k \rfloor$ and $\lceil k \rceil$ for the generic witnesses of the isomorphism we write

$$\mathsf{G}\ C \to B \quad \underset{\lfloor - \rfloor}{\overset{\lceil - \rceil}{\cong}} \quad C \to \mathsf{F}\ B \tag{4.56}$$

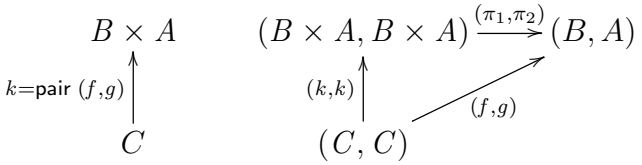
From this a monad $\mathsf{T} = \mathsf{F} \cdot \mathsf{G}$ arises defined by $\eta = \lceil id \rceil$ and $\mu = \mathsf{F} \lfloor id \rfloor$.

Let us see another example of a monad arising from one such adjunction (4.56). Recall exercise 2.25, on page 49, 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

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

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, $\mathsf{F}\ (f, g) = f \times g$:



This is but another way of writing the universal property of products (2.63), since $(f, g) = (h, k) \Leftrightarrow f = h \wedge g = k$ and $\mathsf{pair}\ (f, g) = \langle f, g \rangle$, recall exercise 2.25.

What is, then, the monad behind this *pairing* adjunction? It is the *pairing monad* $(\mathsf{F} \cdot \mathsf{G})\ X = \mathsf{F}\ (\mathsf{G}\ X) = \mathsf{F}\ (X, X) = X \times X$, where $\eta = \langle id, id \rangle$ and $\mu = \pi_1 \times \pi_2$. This monad allows us to work with pairs regarded as 2-dimensional *vectors* (y, x). For instance, the **do**-expression

$$\mathbf{do}\ \{x \leftarrow (2, 3); y \leftarrow (4, 5); \mathsf{return}\ (x + y)\}$$

yields $(6, 8)$ as result in this monad — the *vectorial* sum of vectors $(2, 3)$ and $(4, 5)$. A simple encoding of this monad in Haskell is:

$$\mathbf{data}\ P\ a = P\ (a, a)\ \mathbf{deriving}\ \mathit{Show}$$

$$\mathbf{instance}\ \mathit{Functor}\ P\ \mathbf{where}$$
$$\quad \mathit{fmap}\ f\ (P\ (a, b)) = P\ (f\ a, f\ b)$$

$$\mathbf{instance}\ \mathit{Monad}\ P\ \mathbf{where}$$
$$\quad x \gg\!= f = (\mu \cdot \mathit{fmap}\ f)\ x$$
$$\quad \mathsf{return}\ a = P\ (a, a)$$

$$\mu :: P\ (P\ a) \to P\ a$$
$$\mu\ (P\ (P\ (a, b), P\ (c, d))) = P\ (a, d)$$

Exercise 4.12. *What is the vectorial operation expressed by the definition*

$$\mathit{op}\ k\ v = \mathbf{do}\ \{x \leftarrow v; \mathsf{return}\ (k \times x)\}$$

in the pairing monad?

□

---

[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 [25].

4.13 Bibliography notes

The use of monads in computer science started with Moggi [30], who had the idea that monads should supply the extra semantic information needed to implement the lambda-calculus theory. Haskell [23] is among the computer languages which make systematic use of monads for implementing effects and imperative constructs in a purely functional style.

Category theorists invented monads in the 1960's to concisely express certain aspects of universal algebra. Functional programmers invented list comprehensions in the 1970's to concisely express certain programs involving lists. Philip Wadler [41] made a great contribution to the field by showing that list comprehensions could be generalised to arbitrary monads and unify with imperative "do"-notation in case of the monad which explains imperative computations.

Monads are nowadays an essential feature of functional programming and are used in fields as diverse as language parsing [19], component-oriented programming [4], strategic programming [24], multimedia [18] and probabilistic programming [8]. Adjunctions play a major role in [16].