

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` which (respectively) 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:

```
data ExtVal a = Ok a | Error
```

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:

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

So, in abstract terms, one may regard as *partial* every function of type

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

for some *A* and *B*¹.

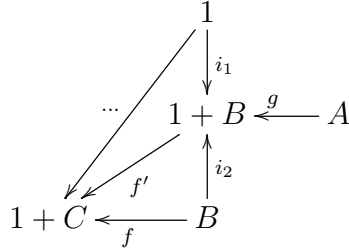
4.2 Putting partial functions together

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

$$\begin{array}{c} 1 + B \xleftarrow{g} A \\ \vdots \\ 1 + C \xleftarrow{f} B \end{array}$$

¹In 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.

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 (...) 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:

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

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

$$(f \bullet g)a = f'(ga)$$

where

$$\begin{aligned} f' \text{ Nothing} &= \text{Nothing} \\ f' (\text{Just } b) &= f \ b \end{aligned}$$

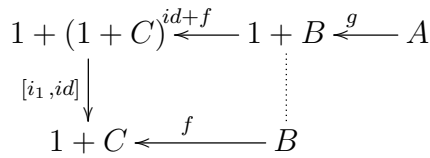
Altogether, we have the following Haskell pointwise expression for $f \bullet g$:

$$\begin{aligned} \lambda a \rightarrow f' (g \ a) \text{ where} \\ f' \text{ Nothing} &= \text{Nothing} \\ f' (\text{Just } b) &= f \ b \end{aligned}$$

Note that the adopted extension of f can be decomposed — by reverse +-absorption (2.41) — into

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

as displayed in diagram



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

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

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 $1 + A \xleftarrow{u} A$. Let us *solve* (4.2) for u :

$$\begin{aligned} & f \bullet u = f = u \bullet f \\ \equiv & \quad \{ \text{substitution} \} \\ & [i_1, f] \cdot u = f = [i_1, u] \cdot f \\ \Leftarrow & \quad \{ \text{let } u = i_2 \} \\ & [i_1, f] \cdot i_2 = f = [i_1, i_2] \cdot f \wedge u = i_2 \\ \equiv & \quad \{ \text{by } +\text{-cancellation (2.38) and } +\text{-reflection (2.39)} \} \\ & f = f = id \cdot f \wedge u = i_2 \\ \Leftarrow & \quad \{ \text{identity} \} \\ & u = i_2 \end{aligned}$$

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

Exercise 4.1. *Prove that property*

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

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:

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

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):

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

To complete the process, one has to *flatten* the nested-sequence output in $(C^*)^*$ via the obvious list-catamorphism $C^* \xleftarrow{\text{concat}} (C^*)^*$, where $\text{concat} \stackrel{\text{def}}{=} ([[], ++])$. In summary:

$$f \bullet g \stackrel{\text{def}}{=} \text{concat} \cdot f^* \cdot g \quad (4.3)$$

as captured in the following diagram:

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

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

□

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: $T X = 1 + X$ in case of (4.1), and $T X = X^*$ in case of (4.3). Then a composition scheme is devised for such functions, which is displayed in

$$\begin{array}{ccc} T(T C) & \xleftarrow{T f} & T B \xleftarrow{g} A \\ \mu \downarrow & & \vdots \\ T C & \xleftarrow{f} & B \end{array}$$

and is given by

$$f \bullet g \stackrel{\text{def}}{=} \mu \cdot T f \cdot g \quad (4.4)$$

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

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

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

datatype T will form a so-called *monad* type, of which $(1+)$ and $(-)^*$ are the two examples seen above. Arrow $\mu \cdot T f$ 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 (4.4) is referred to as *Kleisli composition*.

A *monadic arrow* $T B \xleftarrow{f} A$ 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 $T B \xleftarrow{u} B$ 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 \mathbb{T} to be regarded as a monad:

Multiplication :

$$\begin{array}{ccc}
 \mathbb{T}^2 A & \xleftarrow{\mu} & \mathbb{T}^3 A \\
 \mu \downarrow & & \downarrow \mathbb{T} \mu \\
 \mathbb{T} A & \xleftarrow{\mu} & \mathbb{T}^2 A
 \end{array}
 \qquad \mu \cdot \mu = \mu \cdot \mathbb{T} \mu \qquad (4.5)$$

Unit :

$$\begin{array}{ccc}
 \mathbb{T}^2 A & \xleftarrow{u} & \mathbb{T} A \\
 \mu \downarrow & \swarrow id & \downarrow \mathbb{T} u \\
 \mathbb{T} A & \xleftarrow{\mu} & \mathbb{T}^2 A
 \end{array}
 \qquad \mu \cdot u = \mu \cdot \mathbb{T} u = id \qquad (4.6)$$

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 (4.6) 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 (4.5) and (4.6), the different instances of μ and u are differently typed, as these are polymorphic and exhibit natural properties:

μ -natural :

$$\begin{array}{ccc}
 A & \mathbb{T} A \xleftarrow{\mu} \mathbb{T}^2 A & \\
 f \downarrow & \mathbb{T} f \downarrow \quad \downarrow \mathbb{T}^2 f & \\
 B & \mathbb{T} B \xleftarrow{\mu} \mathbb{T}^2 B &
 \end{array}
 \qquad \mathbb{T} f \cdot \mu = \mu \cdot \mathbb{T}^2 f \qquad (4.7)$$

u -natural :

$$\begin{array}{ccc}
 A & \mathbb{T} A \xleftarrow{u} A & \\
 f \downarrow & \mathbb{T} f \downarrow \quad \downarrow f & \\
 B & \mathbb{T} B \xleftarrow{u} B &
 \end{array}
 \qquad \mathbb{T} f \cdot u = u \cdot f \qquad (4.8)$$

The simplest of all monads is the *identity monad* $\mathbb{T} X \stackrel{\text{def}}{=} X$, 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. Such is the main aim of this chapter.

4.4.1 Properties involving (Kleisli) composition

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

$$f \bullet (g \bullet h) = (f \bullet g) \bullet h \quad (4.9)$$

$$u \bullet f = f = f \bullet u \quad (4.10)$$

$$(f \bullet g) \cdot h = f \bullet (g \cdot h) \quad (4.11)$$

$$(f \cdot g) \bullet h = f \bullet (\mathbb{T} g \cdot h) \quad (4.12)$$

$$id \bullet id = \mu \quad (4.13)$$

Properties (4.9) and (4.10) 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 (4.10) holds for the list monad, recall (4.2). A general proof can be produced similarly. We select property (4.9) as an illustration of the rôle of the monadic properties:

$$\begin{aligned} & f \bullet (g \bullet h) \\ = & \quad \{ \text{definition (4.4) twice} \} \\ & \mu \cdot \mathbb{T} f \cdot (\mu \cdot \mathbb{T} g \cdot h) \\ = & \quad \{ \mu \text{ is natural (4.7)} \} \\ & \mu \cdot \mu \cdot \mathbb{T}^2 f \cdot \mathbb{T} g \cdot h \\ = & \quad \{ \text{monad property (4.5)} \} \\ & \mu \cdot \mathbb{T} \mu \cdot \mathbb{T}^2 f \cdot \mathbb{T} g \cdot h \\ = & \quad \{ \text{functor } \mathbb{T} \text{ (3.55)} \} \end{aligned}$$

$$\begin{aligned}
& \mu \cdot \mathbb{T} (\mu \cdot \mathbb{T} f \cdot g) \cdot h \\
= & \quad \{ \text{definition (4.4) twice} \} \\
& (f \bullet g) \bullet h
\end{aligned}$$

Clearly, this calculation generalizes that of exercise 4.1 to any monad \mathbb{T} .

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

$$(\mathbb{T} f) \cdot (h \bullet k) = (\mathbb{T} f \cdot h) \bullet k \quad (4.14)$$

$$(f \cdot g) \bullet h = f \bullet (\mathbb{T} g \cdot h) \quad (4.15)$$

where Kleisli composition trades with normal composition.

□

4.5 Monadic application (binding)

The monadic counterpart of functional application ap (2.73) is another operator ap' which is intended to be “tolerant” in face of any “ \mathbb{T} -inflated” argument x :²

$$\begin{aligned}
& (\mathbb{T} B)^A \times \mathbb{T} A \xrightarrow{ap'} \mathbb{T} B \\
ap'(f, x) &= f' x = (\mu \cdot \mathbb{T} f)x
\end{aligned} \quad (4.16)$$

If in curry/flipped format, monadic application is called *binding* and denoted by the symbol $\gg=$, looking very much like postfix functional application,

$$((\mathbb{T} B)^A)^{\mathbb{T} A} \xrightarrow{\gg=} \mathbb{T} B \quad (4.17)$$

that is:

$$x \gg= f \stackrel{\text{def}}{=} (\mu \cdot \mathbb{T} f)x \quad (4.18)$$

This operator will exhibit properties arising from its definition and the basic monadic properties, *e.g.*

$$x \gg= u$$

²Recall that notation A^B expresses the set of all functions $f : A \rightarrow B$.

$$\begin{aligned}
&\equiv \{ \text{definition (4.18)} \} \\
&\quad (\mu \cdot \top u)x \\
&\equiv \{ \text{law (4.6)} \} \\
&\quad (id)x \\
&\equiv \{ \text{identity function} \} \\
&\quad x
\end{aligned}$$

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

$$(((x \gg f_1) \gg f_2) \gg \dots f_{n-1}) \gg f_n$$

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

4.6 Sequencing and the do-notation

Given two monadic values x and y , it becomes possible to “sequence” them, thus obtaining another of such value, by defining the following operator

$$x \gg y \stackrel{\text{def}}{=} x \gg y \quad (4.19)$$

of type $(\gg) : \top A \rightarrow \top B \rightarrow \top B$. For instance, within the finite-list monad, one has

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

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

$$x_1 \gg x_2 \gg \dots \gg x_n$$

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

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

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

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

4.7 Generators and comprehensions

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

$$a \leftarrow x \quad (4.20)$$

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

$$\mathbf{do} \ a \leftarrow x_1; x_2; \dots; x_n \stackrel{\text{def}}{=} x_1 \gg \lambda a \cdot (\mathbf{do} \ x_2; \dots; x_n) \quad (4.21)$$

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)

$$\mathbf{do} \ a \leftarrow [1, 2, 3]; [a^2] \quad (4.22)$$

we mean

$$\begin{aligned} & [1, 2, 3] \gg \lambda a. [a^2] \\ &= \text{concat}((\lambda a. [a^2])^* [1, 2, 3]) \\ &= \text{concat}([1], [4], [9]) \\ &= [1, 4, 9] \end{aligned}$$

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,

$$[a^2 \mid a \leftarrow [1, 2, 3]] \quad (4.23)$$

as a “shorthand” of (4.22). This is an instance of the so-called *comprehension* notation, which can be defined in general as follows:

$$[e \mid a_1 \leftarrow x_1, \dots, a_n \leftarrow x_n] = \mathbf{do} \ \{a_1 \leftarrow x_1; \dots; a_n \leftarrow x_n; u(e)\} \quad (4.24)$$

where u is the monad’s unit (4.6,4.8).

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

$$[t] = u \ t \quad (4.25)$$

$$[f \ x \mid x \leftarrow l] = (\mathbb{T} f) l \quad (4.26)$$

$$[t \mid p, q] = \mu [[t \mid q] \mid p] \quad (4.27)$$

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

Exercise 4.5. Show that

$$(f \bullet g) a = \text{do } \{ b \leftarrow g a; f b \} \quad (4.28)$$

$$T f x = \text{do } \{ a \leftarrow x; u (f x) \} \quad (4.29)$$

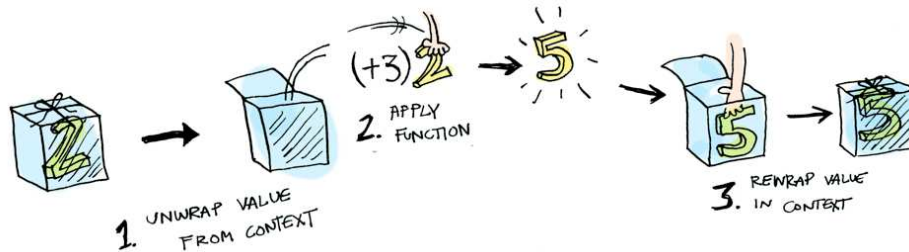
Note that the second **do** expression is equivalent to $x \gg= (u \cdot f)$.

□

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

□

Fact (4.29) is illustrated in the cartoon³



for the computation of $T (+3) x$, 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,

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

```
class Monad m where
  return :: a → m a
  (≫) :: m a → (a → m b) → m b
```

where `return` refers to the unit of m , on top of which the “sequence” operator

```
(≫) :: m a → m b → m b
fail :: String → m a
```

is defined by

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

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`:

```
join :: (Monad m) ⇒ m (m a) → m a
join x = x ≫= id
```

This is easily justified:

$$\begin{aligned}
 \text{join } x &= x \gg= id & (4.30) \\
 &= \{ \text{definition (4.18)} \} \\
 &\quad (\mu \cdot \top id)x \\
 &= \{ \text{functors commute with identity (3.54)} \} \\
 &\quad (\mu \cdot id)x \\
 &= \{ \text{law (2.10)} \} \\
 &\quad \mu x
 \end{aligned}$$

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

```
(•) :: Monad t ⇒ (b → t c) → (d → t b) → d → t c
(f • g) a = (g a) ≫= f
```

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 :: IO String` is a particular action. Parameter `String` refers to the fact that this action “delivers” — or extracts — a string from the environment. This meaning is clearly conveyed by the type `String` assigned to symbol `l` in

$$\text{do } l \leftarrow \text{getLine}; \dots l \dots$$

which is consistent with typing rule for generators (4.20). Sequencing corresponds to the “;” syntax in most programming languages (e.g. C) and the `do`-notation is particularly intuitive in the IO-context.

Examples of functions delivering actions are

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

and

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

— 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 intricacies of the underlying machine:

```
instance Monad IO where
  (≫=) = primbindIO
  return = primretIO
```

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

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

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

$$\begin{aligned} x \gg= (u \cdot f) &= (\mu \cdot \text{IO}(u \cdot f))x \\ &= \{ \text{functors commute with composition} \} \\ &= (\mu \cdot \text{IO } u \cdot \text{IO } f)x \end{aligned}$$

$$\begin{aligned}
&= \{ \text{law (4.6) for } T = IO \} \\
&\quad (IO\ f)x \\
&= \{ \text{definition of } fmap \} \\
&\quad (fmap\ f)x
\end{aligned}$$

For enjoyable reading on monadic input/output in HASKELL see [17], chapter 18.

Exercise 4.7. *Extend the Maybe monad to the following “error message” exception handling datatype:*

data Error a = Err String | Ok a **deriving** Show

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

□

Exercise 4.8. *Recalling section 3.13, show that any inductive type with base functor*

$$B(f, g) = f + F\ g$$

where F is an arbitrary functor, forms a monad for

$$\begin{aligned}
\mu &= ([id, in \cdot i_2]) \\
u &= in \cdot i_1.
\end{aligned}$$

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

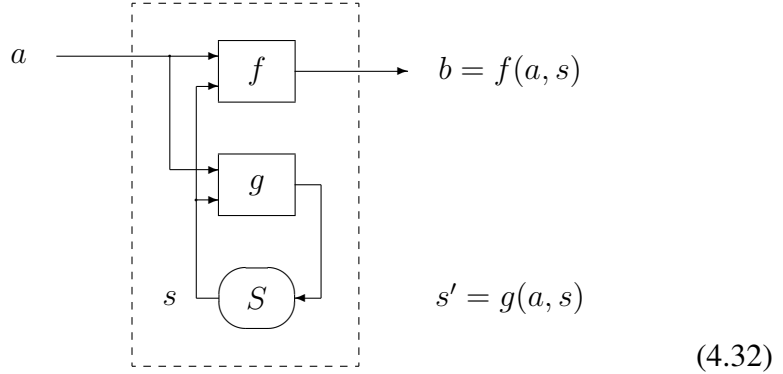
$$A \times S \xrightarrow{\delta} B \times S \quad (4.31)$$

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.

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 diagram:



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. Such an abstraction leads to the *state monad* in the following way: taking (4.31) and recalling (2.81), we simply transpose (ie. *curry*) δ and obtain

$$A \xrightarrow{\bar{\delta}} \underbrace{(B \times S)^S}_{(\text{St } S) \text{ } B} \quad (4.33)$$

thus “shifting” the input state to the output. In this way, $\bar{\delta} a$ is a function capturing all state-transitions (and corresponding outputs) for input a . For instance, the function which *appends* a new element to the back of a queue,

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

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

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

Action *enqueue* performs *enq* on the state while acknowledging it by issuing an output of type 1.

Unit and multiplication. Let us show that

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

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 2.14 is recommended at this point.

Notation \widehat{f} will be used to abbreviate *uncurry* f , enabling the following variant of universal law (2.73),

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

whose cancellation

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

is written pointwise as follows:

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

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

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

on objects and

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

on functions, where $(-)^S$ is the exponential functor (2.77).

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

$$\begin{aligned} u &: A \rightarrow (A \times S)^S \\ u &= \overline{id} \end{aligned} \quad (4.41)$$

Let us see what this means:

$$\begin{aligned} u &= \overline{id} \\ \equiv & \{ (2.73) \} \\ & ap \cdot (u \times id) = id \\ \equiv & \{ \text{introducing variables} \} \\ & ap(u \ a, s) = (a, s) \\ \equiv & \{ \text{definition of } ap \} \\ & (u \ a)s = (a, s) \end{aligned}$$

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 (2.77)) for x of type $((A \times S)^S \times S) \xrightarrow{x} (A \times S)$. This, on its turn, is easily recognized as an instance of the ap polymorphic function (2.73), which is such that $ap = \widehat{id}$, recall (2.75). Altogether, we define

$$\mu = ap^S \quad (4.42)$$

Let us inspect the behaviour of μ by checking the meaning of applying it to an action expressed as in diagram (2.81):

$$\begin{aligned} \mu \langle f, g \rangle &= ap^S \langle f, g \rangle \\ \equiv & \{ (2.77) \} \\ & \mu \langle f, g \rangle = ap \cdot \langle f, g \rangle \\ \equiv & \{ \text{extensional equality (2.5)} \} \\ & \mu \langle f, g \rangle s = ap(f \ s, g \ s) \\ \equiv & \{ \text{definition of } ap \} \\ & \mu \langle f, g \rangle s = (f \ s)(g \ s) \end{aligned}$$

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 (4.6) is made in two parts, checking $\mu \cdot u = id$ first,

$$\begin{aligned}
 & \mu \cdot u \\
 = & \quad \{ \text{definitions} \} \\
 & ap^S \cdot \overline{id} \\
 = & \quad \{ \text{exponentials absorption (2.78)} \} \\
 & \overline{ap \cdot id} \\
 = & \quad \{ \text{reflection (2.75)} \} \\
 & id
 \end{aligned}$$

and then checking $\mu \cdot (\top u) = id$:

$$\begin{aligned}
 & \mu \cdot (\top u) \\
 = & \quad \{ (4.42, 4.40) \} \\
 & ap^S \cdot (\overline{id} \times id)^S \\
 = & \quad \{ \text{functor} \} \\
 & (ap \cdot (\overline{id} \times id))^S \\
 = & \quad \{ \text{cancellation (2.74)} \} \\
 & id^S \\
 = & \quad \{ \text{functor} \} \\
 & id
 \end{aligned}$$

The proof of (4.5) is also not difficult once supported by the laws of exponentials.

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

$$\begin{aligned}
 & f \bullet g \\
 = & \quad \{ (4.4) \}
 \end{aligned}$$

$$\begin{aligned}
& \mu \cdot \top f \cdot g \\
= & \{ (4.42) ; (4.40) \} \\
& ap^S \cdot (f \times id)^S \cdot g \\
= & \{ (-)^S \text{ is a functor} \} \\
& (ap \cdot (f \times id))^S \cdot g \\
= & \{ (4.36) \} \\
& \widehat{f}^S \cdot g \\
= & \{ \text{cancellation} \} \\
& \widehat{f}^S \cdot \widehat{\bar{g}} \\
= & \{ \text{absorption (2.78)} \} \\
& \overline{\widehat{f} \cdot \widehat{g}}
\end{aligned}$$

In summary, we have:

$$f \bullet g = \overline{\widehat{f} \cdot \widehat{g}} \quad (4.43)$$

which can be written alternatively as

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

Let us use this in calculating law

$$pop \bullet push = u \quad (4.44)$$

where *push* and *pop* are such that

$$\begin{aligned}
push & : A \rightarrow (1 \times S)^S \\
\widehat{push} & \stackrel{\text{def}}{=} \langle !, (\cdot) \rangle
\end{aligned} \quad (4.45)$$

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

for *S* the datatype of finite lists. We reason:

$$pop \bullet push$$

$$\begin{aligned}
&= \{ (4.43) \} \\
&\quad \overline{\widehat{pop} \cdot \widehat{push}} \\
&= \{ (4.45, 4.46) \} \\
&\quad \overline{\langle head, tail \rangle \cdot \pi_2 \cdot \langle !, \widehat{(\cdot)} \rangle} \\
&= \{ (2.20, 2.24) \} \\
&\quad \overline{\langle head, tail \rangle \cdot \widehat{(\cdot)}} \\
&= \{ out \cdot in = id \text{ (lists)} \} \\
&\quad \overline{id} \\
&= \{ (4.41) \} \\
&\quad u
\end{aligned}$$

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

$$\begin{aligned}
&x \gg h \\
&= \{ (4.18) \} \\
&\quad (\mu \cdot \top h)x \\
&= \{ (4.42) \text{ and } (4.40) \} \\
&\quad (ap^S \cdot (h \times id)^S)x \\
&= \{ (-)^S \text{ is a functor} \} \\
&\quad (ap \cdot (h \times id))^S x \\
&= \{ \text{cancellation (4.37)} \} \\
&\quad \widehat{h}^S x \\
&= \{ \text{exponential functor (2.77)} \} \\
&\quad \widehat{h} \cdot x
\end{aligned}$$

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

$$\langle f, g \rangle \gg h = \widehat{h} \cdot \langle f, g \rangle$$

$$\begin{aligned}
&\equiv \{ \text{go pointwise} \} \\
&\quad (\langle f, g \rangle \gg h) s = \widehat{h}(\langle f, g \rangle s) \\
&\equiv \{ (2.18) \} \\
&\quad (\langle f, g \rangle \gg h) s = \widehat{h}(f\ s, g\ s) \\
&\equiv \{ (4.38) \} \\
&\quad (\langle f, g \rangle \gg h) s = h(f\ s)(g\ s)
\end{aligned}$$

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:

$$get \stackrel{\text{def}}{=} \langle id, id \rangle \quad (4.47)$$

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

Action g retrieves the data stored in the state without changing it, while put — which can also be written

$$put\ s = \langle !, \underline{s} \rangle \quad (4.49)$$

or even as

$$put\ s = modify\ \underline{s} \quad (4.50)$$

where

$$modify\ f \stackrel{\text{def}}{=} \langle !, f \rangle \quad (4.51)$$

updates the state via state-to-state function f — stores a particular value in the state.

The following is an example, in Haskell, of the standard use of *get/put* in managing context data, in this case a counter. The function decorates each node of a *BTree* (recall this datatype from page 106) with its position in the tree:

```

decBTree Empty = return Empty
decBTree (Node (a, (t1, t2))) = do {

```

```

n ← get;
put (n + 1);
x ← decBTree t1;
y ← decBTree t2;
return (Node ((a, n), (x, y)))
}

```

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 which adds all numbers found in its input:

```

sum [] = 0
sum (h : t) = h + sum t

```

Notice that this code could have been written as follows

```

sum [] = id 0
sum (h : t) = let x = sum t in id (h + x)

```

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*:

```
msum [] = return 0
msum (h : t) = do { x <- msum t; return (h + x) }
```

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 $Id\ X = X$. It is very easy to show that inside this monad return— is the identity and do— means the same as let— — 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:

```
msum' [] = return 0
msum' (h : t) =
  do { x <- msum' t;
      print ("x= " ++ show x);
      return (h + x) }
```

Thus one obtains traces 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

```
decBTree Empty = return Empty
decBTree (Node (a, (t1, t2))) =
  do
    x ← decBTree t1;
    y ← decBTree t2;
    return (Node (a, (x, y)))
```

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

```
decBTree Empty = Empty
decBTree (Node (a, (t1, t2))) =
  let
    x = decBTree t1
    y = decBTree t2
  in Node (a, (x, y))
```

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 identity⁴. The same could be said about imperative code of a particular class — the *recursive descent* one — much used in 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,

⁴We have seen the same kind of “inspiration” before in building type functors (3.76) which, for $f = id$, boil down to the identity.

```

getmin [a] = a
getmin (h : t) = min h (getmin t)

```

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 definition we first go monadic, as we did before,

```

mgetmin [a] = return a
mgetmin (h : t) = do {x ← mgetmin t; return (min h x)}
λend {verbatim}

```

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

```

mgetmin [] = Nothing
mgetmin [a] = return a
mgetmin (h : t) = do {x ← mgetmin t; return (min h x)}

```

Alternatively, we might have written

```

mgetmin [] = Error "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. 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

```

map f [] = []
map f (h : t) = (f h) : map f t

```

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,

```

map f [] = []
map f (h : t) = let

```

```

b = f h
x = map f t in b : x

```

we go monadic in the usual way,

```

mmap f [] = return []
mmap f (h : t) = do { b ← f h; x ← mmap f t; return (b : x) }

```

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

$$\text{mmap} :: (\text{Monad } m) \Rightarrow (a \rightarrow m \, b) \rightarrow [a] \rightarrow m \, [b]$$

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.

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 $[m \, a] \rightarrow m \, [a]$, 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 `[Just 1, Nothing, Just 2]` the output will be `Nothing`.

There is much more one could say about monadic programming, but this is enough to see *where such a programming style comes from*.

Exercise 4.9. *Use the monadification technique to encode monadic function*

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

which generalizes the list-based `filter` function.

□

Exercise 4.10. “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 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 (2.73):

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

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

$$k = \bar{f} \Leftrightarrow f = \underbrace{ap \cdot G k}_{\hat{k}}$$

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

$$G C \rightarrow B \cong C \rightarrow F B \quad (4.52)$$

assuming F and G as defined above.

Note how (4.52) expresses a kind of “shunting rule” at type level: G s on the input side can be “shunted” to the output if replaced by F s. 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, $ap = \hat{id}$ – recall (2.75)
- Cancellation: $\hat{id} \cdot G \bar{f} = f$ – recall (2.74)

- Fusion: $\bar{h} \cdot g = \overline{h \cdot G g}$ — recall (2.76)
- Absorption: $(F g) \cdot \bar{h} = \overline{g \cdot h}$ — recall (2.78)
- Naturality: $h \cdot \bar{id} = \bar{id} \cdot G (F h)$
- Functor: $F h = \overline{h \cdot ap}$
- Closed definitions: $\hat{k} = ap \cdot (G k)$ and $\bar{g} = (F g) \cdot \bar{id}$, the latter following from absorption.

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

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

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

$$\begin{aligned}
 k = \bar{f} &\Leftrightarrow ap \cdot G k = f \\
 \equiv &\quad \{ \text{see above} \} \\
 k = (F f) \cdot \bar{id} &\Leftrightarrow f = \hat{k} \\
 \equiv &\quad \{ \text{swapping variables } k \text{ and } f, \text{ to match the starting diagram} \} \\
 f = (F k) \cdot \bar{id} &\Leftrightarrow k = \hat{f} \\
 \square
 \end{aligned}$$

that is,

$$\begin{array}{ccc}
 k = \hat{f} \Leftrightarrow f = \underbrace{F k \cdot \eta}_{\bar{k}} & \begin{array}{c} G B \\ \downarrow k=\hat{f} \\ C \end{array} & \begin{array}{ccc} F (G B) & \xleftarrow{\eta} & B \\ \downarrow F k & \swarrow f & \\ F A & & \end{array}
 \end{array}$$

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

$$\mu = F \hat{id} \tag{4.53}$$

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

Adjunctions

The reasoning we have made above for exponentials and the state monad generalizes for any other monad. In general, isomorphisms of shape (4.52) are called an *adjunction* of the two functors F and G , which are said to be *adjoint* to each other. One writes $G \dashv F$ 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

$$\begin{array}{ccc} G C \rightarrow B & \begin{array}{c} \xrightarrow{\lceil - \rceil} \\ \cong \\ \xleftarrow{\lfloor - \rfloor} \end{array} & C \rightarrow F B \end{array} \quad (4.54)$$

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

Let us see another example of a monad arising from an adjunction (4.54). Recall exercise 2.23, on page 43, where *pair* / *unpair* witness an isomorphism similar to that of *curry*/*uncurry*, for *pair* $(f, g) = \langle f, g \rangle$ and *unpair* $k = (\pi_1 \cdot k, \pi_2 \cdot k)$. This can be cast into an adjunction as follows

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

where $G k = (k, k)$. 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 $f : A \rightarrow B$ and $g : C \rightarrow D$, the pair (f, g) has type $(A \rightarrow B) \times (C \rightarrow D)$. However, we shall abuse of notation again and declare the type $(f, g) : (A, C) \rightarrow (B, D)$.⁵ In the opposite direction, $F(f, g) = f \times g$:

$$\begin{array}{ccc} B \times A & (B \times A, B \times A) & \xrightarrow{(\pi_1, \pi_2)} (B, A) \\ \uparrow k = \text{pair } (f, g) & \uparrow (k, k) & \nearrow (f, g) \\ C & (C, C) & \end{array}$$

This is but another way of writing the universal property of products (2.55), since $(f, g) = (h, k) \Leftrightarrow f = h \wedge g = k$ and $\text{pair } (f, g) = \langle f, g \rangle$, recall exercise 2.23.

⁵Strictly 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 [24].

What is, then, the monad behind this *pairing* adjunction? It is the *pairing monad* $(F \cdot G) X = F (G X) = F (X, X) = X \times X$, 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

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

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

```
data P a = P (a, a) deriving Show
instance Functor P where
  fmap f (P (a, b)) = P (f a, f b)
instance Monad P where
   $x \gg= f = (\mu P \cdot \text{fmap } f) x$ 
  return a = P (a, a)
   $\mu P :: P (P a) \rightarrow P a$ 
   $\mu P (P (P (a, b), P (c, d))) = P (a, d)$ 
```

Exercise 4.11. What is the vectorial operation expressed by the definition

```
 $op\ k\ v = \text{do } \{ x \leftarrow v; \text{return } (k \times x) \}$ 
```

in the pairing monad?

□

4.12 Bibliography notes

The use of monads in computer science started with Moggi [29], who had the idea that monads should supply the extra semantic information needed to implement the lambda-calculus theory. Haskell [22] is among the computer languages which make systematic use of monads for implementing effects and imperative constructs in an otherwise purely functional language.

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 [34] 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 [18], component-oriented programming [4], strategic programming [23], multimedia [17] and probabilistic programming [8]. Adjunctions play a major role in [15].

Part II

Moving Away From (Pure) Functions

not given in the current version of this textbook

Appendix A

Appendix

A.1 Haskell support library

```
infix 5 ×  
infix 4 +
```

Products

$$\begin{aligned}\langle \cdot, \cdot \rangle &:: (a \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow a \rightarrow (b, c) \\ \langle f, g \rangle x &= (f\ x, g\ x) \\ (\times) &:: (a \rightarrow b) \rightarrow (c \rightarrow d) \rightarrow (a, c) \rightarrow (b, d) \\ f \times g &= \langle f \cdot \pi_1, g \cdot \pi_2 \rangle\end{aligned}$$

The 0-adic split is the unique function of its type

$$\begin{aligned}(!) &:: a \rightarrow () \\ (!) &= \underline{\quad}\end{aligned}$$

Renamings:

$$\begin{aligned}\pi_1 &= \text{fst} \\ \pi_2 &= \text{snd}\end{aligned}$$

Coproduct

Renamings:

$$\begin{aligned} i_1 &= i_1 \\ i_2 &= i_2 \end{aligned}$$

Either is predefined:

$$\begin{aligned} (+) &:: (a \rightarrow b) \rightarrow (c \rightarrow d) \rightarrow a + c \rightarrow b + d \\ f + g &= [i_1 \cdot f, i_2 \cdot g] \end{aligned}$$

McCarthy's conditional:

$$p \rightarrow f, g = [f, g] \cdot p?$$

Exponentiation

Curry is predefined.

$$\begin{aligned} ap &:: (a \rightarrow b, a) \rightarrow b \\ ap &= (\$) \end{aligned}$$

Functor:

$$\begin{aligned} \cdot &:: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c \\ f \cdot &= \overline{f \cdot ap} \end{aligned}$$

Pair to predicate isomorphism (2.95):

$$\begin{aligned} p2p &:: (b, b) \rightarrow \mathbb{B} \rightarrow b \\ p2p \ p \ b &= \mathbf{if} \ b \ \mathbf{then} \ (\mathbf{snd} \ p) \ \mathbf{else} \ (\mathbf{fst} \ p) \end{aligned}$$

The exponentiation functor is $(a \rightarrow)$ predefined:

$$\begin{aligned} \mathbf{instance} \ &Functor \ ((\rightarrow) \ s) \ \mathbf{where} \\ \mathbf{fmap} \ f \ g &= f \cdot g \end{aligned}$$

Others

$_ :: a \rightarrow b \rightarrow a$ such that $_ x = a$ is predefined. Guards:

$$\begin{aligned} \cdot? &:: (a \rightarrow \mathbb{B}) \rightarrow a \rightarrow a + a \\ p? \ x &= \mathbf{if} \ p \ x \ \mathbf{then} \ i_1 \ x \ \mathbf{else} \ i_2 \ x \end{aligned}$$

Natural isomorphisms

```

swap :: (a, b) → (b, a)
swap = ⟨π2, π1⟩

assocr :: ((a, b), c) → (a, (b, c))
assocr = ⟨π1 · π1, snd × id⟩

assocl :: (a, (b, c)) → ((a, b), c)
assocl = ⟨id × π1, π2 · π2⟩

undistr :: (a, b) + (a, c) → (a, b + c)
undistr = [id × i1, id × i2]

undistl :: (b, c) + (a, c) → (b + a, c)
undistl = [i1 × id, i2 × id]

coswap :: a + b → b + a
coswap = [i2, i1]

coassocr :: (a + b) + c → a + (b + c)
coassocr = [id + i1, i2 · i2]

coassocl :: b + (a + c) → (b + a) + c
coassocl = [i1 · i1, i2 + id]

distr =  $\widehat{f}$  where f a = (g a) + (g a) where g a x = (a, x)
distl = (swap + swap) · distr · swap

flatr :: (a, (b, c)) → (a, b, c)
flatr (a, (b, c)) = (a, b, c)

flatl :: ((a, b), c) → (a, b, c)
flatl ((b, c), d) = (b, c, d)

br = ⟨id, !⟩
bl = swap · br

```

Class bifunctor

```

class BiFunctor f where
    bmap :: (a → b) → (c → d) → (f a c → f b d)

instance BiFunctor · + · where
    bmap f g = f + g

instance BiFunctor (, ) where
    bmap f g = f × g

```

Monads

Kleisli monadic composition:

infix 4 •

$$(\bullet) :: \text{Monad } a \Rightarrow (b \rightarrow a \ c) \rightarrow (d \rightarrow a \ b) \rightarrow d \rightarrow a \ c$$

$$(f \bullet g) \ a = (g \ a) \gg\!\!= f$$

Multiplication, also known as join:

$$\text{mult} :: (\text{Monad } m) \Rightarrow m \ (m \ b) \rightarrow m \ b$$

$$\text{mult} = (\gg\!\!=id)$$

Monadic binding:

$$\text{ap}' :: (\text{Monad } m) \Rightarrow (a \rightarrow m \ b, m \ a) \rightarrow m \ b$$

$$\text{ap}' = \widehat{\text{flip } (\gg\!\!=)}$$

List monad:

$$\text{singl} :: a \rightarrow [a]$$

$$\text{singl} = \text{return}$$

Strong monads:

```
class (Functor f, Monad f)  $\Rightarrow$  Strong f where
  rstr :: (f a, b)  $\rightarrow$  f (a, b)
  rstr (x, b) = do a  $\leftarrow$  x; return (a, b)
  lstr :: (b, f a)  $\rightarrow$  f (b, a)
  lstr (b, x) = do a  $\leftarrow$  x; return (b, a)

instance Strong IO
instance Strong []
instance Strong Maybe
```

Double strength:

$$\text{dstr} :: \text{Strong } m \Rightarrow (m \ a, m \ b) \rightarrow m \ (a, b)$$

$$\text{dstr} = \text{rstr} \bullet \text{lstr}$$

Exercise 4.8.13 in Jacobs' "Introduction to Coalgebra" [19]:

$$\text{splitm} :: \text{Strong } ff \Rightarrow ff \ (a \rightarrow b) \rightarrow a \rightarrow ff \ b$$

$$\text{splitm} = \overline{\text{fmap } \text{ap} \cdot \text{rstr}}$$

Monad transformers:

```
class (Monad m, Monad (t m))  $\Rightarrow$  MT t m where    -- monad transformer class
    lift :: m a  $\rightarrow$  t m a
```

Nested lifting:

```
dlift :: (MT t (t1 m), MT t1 m)  $\Rightarrow$  m a  $\rightarrow$  t (t1 m) a
dlift = lift  $\cdot$  lift
```

Basic functions, abbreviations

```
zero = 0
one = 1
nil = []
cons =  $\widehat{\cdot}$ 
add =  $\widehat{+}$ 
mul =  $\widehat{*}$ 
conc =  $\widehat{++}$ 
inMaybe :: () + a  $\rightarrow$  Maybe a
inMaybe = [Nothing, Just]
```

More advanced

```
class (Functor f)  $\Rightarrow$  Unzipable f where
    unzp :: f (a, b)  $\rightarrow$  (f a, f b)
    unzp =  $\langle$ fmap  $\pi_1$ , fmap  $\pi_2$  $\rangle$ 
class Functor g  $\Rightarrow$  DistL g where
     $\lambda$  :: Monad m  $\Rightarrow$  g (m a)  $\rightarrow$  m (g a)
instance DistL [] where  $\lambda$  = sequence
instance DistL Maybe where
     $\lambda$  Nothing = return Nothing
     $\lambda$  (Just a) = mp Just a where mp f = (return  $\cdot$  f)  $\bullet$  id
```

Convert Monad into Applicative:

```
aap :: Monad m  $\Rightarrow$  m (a  $\rightarrow$  b)  $\rightarrow$  m a  $\rightarrow$  m b
aap mf mx = do { f  $\leftarrow$  mf; x  $\leftarrow$  mx; return (f x) }
```


Bibliography

- [1] C. Aarts, R.C. Backhouse, P. Hoogendijk, E.Voermans, and J. van der Woude. A relational theory of datatypes, December 1992. Available from `www.cs.nott.ac.uk/~rcb`.
- [2] R.C. Backhouse. *Mathematics of Program Construction*. Univ. of Nottingham, 2004. Draft of book in preparation. 608 pages.
- [3] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *CACM*, 21(8):613–639, August 1978.
- [4] L.S. Barbosa. *Components as Coalgebras*. University of Minho, December 2001. Ph. D. thesis.
- [5] R. Bird. Introduction to Functional Programming. Series in Computer Science. Prentice-Hall International, 2nd edition, 1998. C.A.R. Hoare, series editor.
- [6] R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall, 1997.
- [7] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, January 1977.
- [8] M. Erwig and S. Kollmannsberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16:21–34, January 2006.
- [9] R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19, pages 19–32. American Mathematical Society, 1967. Proc. Symposia in Applied Mathematics.
- [10] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.
- [11] J. Gibbons. Kernels, in a nut shell. *JLAMP*, 85(5, Part 2):921–930, 2016.

- [12] J. Gibbons and R. Hinze. Just do it: simple monadic equational reasoning. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP'11, pages 2–14, New York, NY, USA, 2011. ACM.
- [13] Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch. When is a function a fold or an unfold?, 2001. WGP, July 2001 (slides).
- [14] Ralf Hinze. Adjoint folds and unfolds — an extended study. *Science of Computer Programming*, 78(11):2108–2159, 2013.
- [15] Ralf Hinze. Adjoint folds and unfolds — an extended study. *Science of Computer Programming*, 78(11):2108–2159, 2013.
- [16] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Conjugate hylomorphisms – or: The mother of all structured recursion schemes. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 527–538, New York, NY, USA, 2015. ACM.
- [17] P. Hudak. The Haskell School of Expression - Learning Functional Programming Through Multimedia. Cambridge University Press, 1st edition, 2000. ISBN 0-521-64408-9.
- [18] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), 1993.
- [19] B. Jacobs. *Introduction to Coalgebra. Towards Mathematics of States and Observations*. Cambridge University Press, 2016.
- [20] P. Jansson and J. Jeuring. Polylib — a library of polytypic functions. In *Workshop on Generic Programming (WGP'98)*, Marstrand, Sweden, 1998.
- [21] J. Jeuring and P. Jansson. Polytypic programming. In *Advanced Functional Programming*, number 1129 in LNCS, pages 68–114. Springer-Verlag, 1996.
- [22] S.L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003. Also published as a Special Issue of the Journal of Functional Programming, 13(1) Jan. 2003.
- [23] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P.L. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of LNCS, pages 357–375. Springer-Verlag, January 2003.
- [24] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

- [25] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
- [26] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986. D. Gries, series editor.
- [27] J. McCarthy. Towards a mathematical science of computation. In C.M. Popplewell, editor, *Proc. of IFIP 62*, pages 21–28, Amsterdam-London, 1963. North-Holland Pub. Company.
- [28] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In S. Peyton Jones, editor, *Proceedings of Functional Programming Languages and Computer Architecture (FPCA95)*, 1995.
- [29] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.
- [30] D. Murta and J.N. Oliveira. A study of risk-aware program transformation. *SCP*, 110:51–77, 2015.
- [31] P. Naur and B. Randell, editors. *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968*. Scientific Affairs Division, NATO, 1969.
- [32] M.S. Paterson and C.E. Hewitt. Comparative schematology. In *Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 119–127, August 1970.
- [33] G. Villavicencio and J.N. Oliveira. *Reverse Program Calculation Supported by Code Slicing*. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE 2001) 2-5 October 2001, Stuttgart, Germany*, pages 35–46. IEEE Computer Society, 2001.
- [34] P.L. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, pages 347–359, London, Sep. 1989. ACM.