Chapter 3

A Look at Monads

In this chapter we present a powerful device in state-of-the-art 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, 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.

3.1 Partial functions

Consider function

$$g\,x \quad \stackrel{\text{def}}{=} \quad 1/x$$

defined on $\mathbb{R}$. Clearly, this function is undefined for $x = 0$ because $g\,0 = 1/0$ is so big a real number that it cannot be properly evaluated. In fact, the HASKELL output for $g\,0 = 1/0$ is just "panic":

```
Main> g 0

Program error: {primDivDouble 1.0 0.0}

Main>
```

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

Panic is very dangerous in programming. In order to avoid this kind of behaviour one has two alternatives, either ensuring that every call to $g\,x$ is *protected* — *i.e.*, the

contexts which wrap up such calls ensure *pre-condition* $x \neq 0$, or one *raises* exceptions, *i.e.* explicit error values. In the former case, mathematical proofs need to be carried out in order to guarantee *safety* (that is, *pre-condition* compliance). The overall effect is to *restrict* 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 ExtReal = Ok Real | Error
```

and then redefine

```
g :: Real -> ExtReal
g 0 = Error
g n = Ok 1/n
```

In general, one might define parametric type

```
data Ext a = Ok a | Error
```

in order to extend an arbitrary data type `a` with its (polymorphic) exception (or error value). Clearly, one has

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

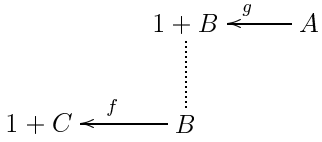
So, in abstract terms, one may regard as *partial* every function of signature

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

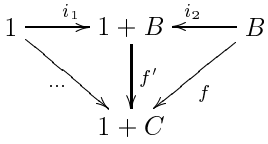
for some A and B [1].

3.2 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 $1 + B$:



[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.

The most "obvious" instance of ... 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:

$$f \bullet g \quad \stackrel{\text{def}}{=} \quad [\, i_1, f \,] \cdot g \tag{3.1}$$

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

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

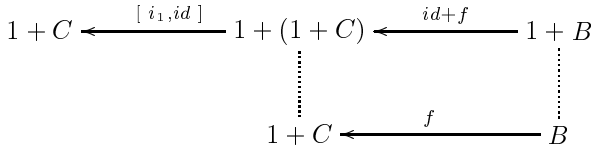
where

$$\begin{array}{rcl} f'\,\mathtt{Error} & = & \mathtt{Error} \\ f'\,(\mathtt{Ok}\, a) & = & f\, a \end{array}$$

Note that the adopted extension of f can be decomposed — by reverse $+$-absorption (1.41) — into

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

as displayed in diagram



All in all, we have the following version of (3.1):

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

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

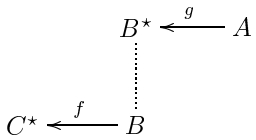
$$f \bullet u = f = u \bullet f \tag{3.2}$$

holds? Whe have to solve (3.2) for u:

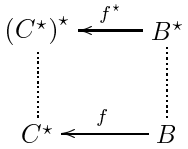
$$\begin{array}{ll} f \bullet u = f = u \bullet f & \\ \quad \Leftrightarrow & \quad \{\text{ substitution}\} \\ \quad & \quad [\, i_1, f \,] \cdot u = f = [\, i_1, u \,] \cdot f \\ \quad \Leftarrow & \quad \{\text{ let } u = i_2\} \\ \quad & \quad [\, i_1, f \,] \cdot i_2 = f = [\, i_1, i_2 \,] \cdot f \wedge u = i_2 \\ \quad \Leftrightarrow & \quad \{\text{ by } +\text{-cancellation (1.38) and } +\text{-reflection (1.39)}\} \\ \quad & \quad f = f = id \cdot f \wedge u = i_2 \\ \quad \Leftarrow & \quad \{\text{ identity}\} \\ \quad & \quad u = i_2 \end{array}$$

3.3 Lists

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



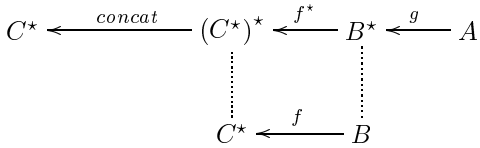
Functions f and g do not compose but once again one can think of extending the consumer function (f) by mapping it along the output of the producer function (g):



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

$$f \bullet g \quad \stackrel{\text{def}}{=} \quad concat \cdot f^\star \cdot g \tag{3.3}$$

as captured in the following diagram:



Exercise 3.1 Show that *singl* (recall exercise 2.8) is the unit u of $\bullet$ in the context of (3.3).
□

Exercise 3.2 Encode in HASKELL a pointwise version of (3.3). **Hint**: first apply (list) cata-absorption (2.67).
□

3.4 Monads

Both function composition schemes (3.1) and (3.3) above share the same polytypic pattern: the output of the producer function is "F-times" more elaborate than the input

of the consumer function, where F is some parametric datatype — $\mathsf{F}\,X = 1 + X$ in case of (3.1), and $\mathsf{F}\,X = X^\star$ in case of (3.3). Then a composition scheme is devised for such functions, which is displayed in

$$\begin{array}{ccccccc} \mathsf{F}\,C & \xleftarrow{\mu} & \mathsf{F}(\mathsf{F}\,C) & \xleftarrow{\mathsf{F}\,f} & \mathsf{F}\,B & \xleftarrow{g} & A \\ & & \vdots & & \vdots & & \\ & & \mathsf{F}\,C & \xleftarrow{f} & B & & \end{array}$$

and is given by

$$f \bullet g \quad \stackrel{\text{def}}{=} \quad \mu \bullet \mathsf{F}\,f \bullet g \tag{3.4}$$

where $\mathsf{F}\,A \xleftarrow{\mu} \mathsf{F}^2\,A$ is a suitable polymorphic function. Together with a unit function $\mathsf{F}\,A \xleftarrow{u} A$ and μ, datatype F will form a so-called *monad* type, of which $1 + \_\_$ and $(\_\_)^\star$ are the two examples seen above.

Arrow $\mu \bullet \mathsf{F}\,f$ 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 (3.4) is called *Kleisli composition*.

A *monadic arrow* $\mathsf{F}B \xleftarrow{f} A$ conveys the idea of a function which produces an output of "type" B "wrapped by F", where datatype F describes some kind of (computational) "effect". The monad's unit $\mathsf{F}B \xleftarrow{u} B$ is a primitive monadic arrow which produces (*i.e.* promotes, injects, wraps) data *together with* 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 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 F to be regarded as a monad:

Multiplication :

$$\begin{array}{ccc} \mathsf{F}^2\,A & \xleftarrow{\mu} & \mathsf{F}^3\,A \\ {\scriptstyle\mu}\downarrow & & \downarrow{\scriptstyle\mathsf{F}\,\mu} \\ \mathsf{F}\,A & \xleftarrow[\mu]{} & \mathsf{F}^2\,A \end{array} \qquad\qquad \mu \bullet \mu = \mu \bullet \mathsf{F}\,\mu \tag{3.5}$$

Unit :

$$\begin{array}{ccc} \mathsf{F}^2\,A & \xleftarrow{u} & \mathsf{F}\,A \\ {\scriptstyle\mu}\downarrow & \swarrow_{id} & \downarrow{\scriptstyle\mathsf{F}\,u} \\ \mathsf{F}\,A & \xleftarrow[\mu]{} & \mathsf{F}^2\,A \end{array} \qquad\qquad \mu \bullet u = \mu \bullet \mathsf{F}\,u = id \tag{3.6}$$

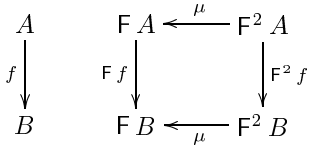
Simple but beautiful symmetries apparent in these diagrams make it easy to memorize their laws and check them for particular cases. For instance, for the $(1 + \_)$ monad, law (3.6) will read as follows:

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

These equalities are easy to check.

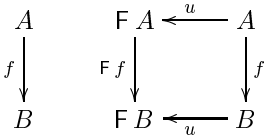
In laws (3.5) and (3.6), the different instances of μ and u are differently typed, as these are polymorphic and exhibit natural properties:

μ-natural :



$$\mathsf{F}\, f \cdot \mu = \mu \cdot \mathsf{F}^2\, f \tag{3.7}$$

u-natural :



$$\mathsf{F}\, f \cdot u = u \cdot f \tag{3.8}$$

The simplest of all monads is the *identity monad* $\mathsf{F}\, X \stackrel{\text{def}}{=} X$, which is such that $\mu = id$, $u = id$ and $f \bullet g = f \cdot g$. So — in a sense — one may be think of all the functional discipline studied so far as a particular case of a wider discipline in which an arbitrary monad is present.

3.4.1 Properties involving (Kleisli) composition

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

$$f \bullet (g \bullet h) = (f \bullet g) \bullet h \tag{3.9}$$
$$u \bullet f = f = f \bullet u \tag{3.10}$$
$$(f \bullet g) \cdot h = f \bullet (g \cdot h) \tag{3.11}$$
$$(f \cdot g) \bullet h = f \bullet (\mathsf{F}\, g \cdot h) \tag{3.12}$$
$$id \bullet id = \mu \tag{3.13}$$

Properties (3.9) and (3.10) are the monadic counterparts of, respectively, (1.8) and (1.10), 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 (3.10) holds for the list monad, recall (3.2). A general proof can be produced similarly. We select property (3.9) as an illustration of the rôle of the monadic properties:

$$
\begin{array}{ll}
f \bullet (g \bullet h) & \\
= & \quad \{ \text{ definition (3.4) twice} \} \\
& \mu \cdot \mathsf{F}\, f \cdot (\mu \cdot \mathsf{F}\, g \cdot h) \\
= & \quad \{ \mu \text{ is natural (3.7)} \} \\
& \mu \cdot \mu \cdot \mathsf{F}(\mathsf{F}\, f) \cdot \mathsf{F}\, g \cdot h \\
= & \quad \{ \text{functor } \mathsf{F} \} \\
& \mu \cdot \mu \cdot \mathsf{F}(\mathsf{F}\, f \cdot g) \cdot h \\
= & \quad \{ \text{definition (3.4)} \} \\
& \mu \cdot (\mathsf{F}\, f \cdot g) \bullet h \\
= & \quad \{ \text{definition (3.4)} \} \\
& (f \bullet g) \bullet h
\end{array}
$$

Exercise 3.3 Check the other laws above.
□

3.5 Monadic application (binding)

The monadic extension of functional application ap (1.65) is another operator ap' which is intended to be "tolerant" in face of any F'ed argument x:

$$
\begin{array}{l}
(\mathsf{F}\, B)^A \times \mathsf{F}\, A \xrightarrow{ap'} B \\
ap'(f, x) = f'\, x = (\mu \cdot \mathsf{F}\, f)x
\end{array}
\tag{3.14}
$$

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

$$
((\mathsf{F}\, B)^A)^{\mathsf{F}\, A} \xrightarrow{>>=} \mathsf{F}\, B \tag{3.15}
$$

that is:

$$
x >>= f \quad \stackrel{\text{def}}{=} \quad (\mu \cdot \mathsf{F}\, f)x \tag{3.16}
$$

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

$$\begin{array}{ll} x >>= u & \\ \Leftrightarrow & \{ \text{ definition (3.16) } \} \\ (\mu \bullet \mathsf{F}\, u)x & \\ \Leftrightarrow & \{ \text{ law (3.6) } \} \\ (id)x & \\ \Leftrightarrow & \{ \text{ identity function} \} \\ x & \end{array}$$

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

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

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

3.6 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:

$$x >> y \quad \stackrel{\text{def}}{=} \quad x >>= \underline{y}$$

For instance, within the finite-list monad, one has

$$[1,2] >> [3,4] = (concat \bullet \underline{[3,4]}^{\star})[1,2] = concat[[3,4],[3,4]] = [3,4,3,4]$$

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

$$x_1 >> x_2 >> \ldots >> x_n$$

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

$$\mathsf{do}\ x_1; x_2; \ldots; x_n \quad \stackrel{\text{def}}{=} \quad x_1 >> \mathsf{do}\ x_2; \ldots; x_n$$

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

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

3.7 Generators and comprehensions

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

$$a \leftarrow x \tag{3.17}$$

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

$$\mathsf{do}\ a \leftarrow x_1; x_2; \ldots; x_n \quad \stackrel{\text{def}}{=} \quad x_1 >>= \lambda a.(\mathsf{do}\ x_2; \ldots; x_n) \tag{3.18}$$

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

$$\mathsf{do}\ a \leftarrow [1,2,3]; [a^2] \tag{3.19}$$

we mean

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

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,

$$[\, a^2 \mid a \leftarrow [1,2,3]\,] \tag{3.20}$$

as a "shorthand" of (3.19). This is an instance of the so-called *comprehension* notation, which can be defined in general as follows:

$$[\, e \mid a_1 \leftarrow x_1, \ldots, a_n \leftarrow x_n\,] \quad = \quad \mathsf{do}\ a_1 \leftarrow x_1; \ldots; a_n \leftarrow x_n; u(e) \tag{3.21}$$

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

$$[t] \quad = \quad u\,t \tag{3.22}$$

$$[\, f\,x \mid x \leftarrow l\,] \quad = \quad (\mathsf{F}\,f)l \tag{3.23}$$

$$[\, t \mid p, q\,] \quad = \quad \mu[\,[\, t \mid q\,] \mid p\,] \tag{3.24}$$

Note, however, that comprehensions are not restricted to lists or sets — they can be defined for any monad F.

3.8 Monads in HASKELL

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

```
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 >> q  = p >>= \ _ -> q
```

as expected. This class is instantiated for finite sequences (`[]`), `Maybe` and `IO`.

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:

$$\begin{aligned} join\, x &= x >>= id \\ &= \quad \{ \text{ definition (3.16) } \} \\ &\quad (\mu \bullet \mathsf{F}\, id)x \\ &= \quad \{ \text{ functors commute with identity (2.44)} \} \\ &\quad (\mu \bullet id)x \\ &= \quad \{ \text{ law (1.10)} \} \\ &\quad \mu\, x \end{aligned}$$

In `Mpi.hs` we define (Kleisli) monadic composition in terms of the binding operator:

```
(.!) :: Monad a => (b -> a c) -> (d -> a b) -> d -> a c
(f .! g) a = (g a) >>= f
```

3.8.1 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 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

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

which is consistent with typing rule for generators (3.17). 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:

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

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

```
    fmap f x = x >>= (return . f)
```

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

$$\begin{array}{lcl} x >>= (u \bullet f) & = & (\mu \bullet \mathsf{IO}(u \bullet f))x \\ & = & \quad \{ \text{ functors commute with composition} \} \\ & & (\mu \bullet \mathsf{IO}\ u \bullet \mathsf{IO}\ f)x \\ & = & \quad \{ \text{ law (3.6) for } \mathsf{F} = \mathsf{IO} \ \} \\ & & (\mathsf{IO}\ f)x \\ & = & \quad \{ \text{ definition of } fmap \ \} \\ & & (fmap\ f)x \end{array}$$

For a very enjoyable reading on monadic input/output in HASKELL see [Hud00], chapter 18.

Exercise 3.4 Use the `do`-notation and the comprehension notation to output the following truth-table, in HASKELL:

p / q	False	True
False	False	False
True	False	True

□

Exercise 3.5 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?
□

Bibliography

[Hud00] P. Hudak. *The Haskell School of Expression - Learning Functional Programming Through Multimedia* . Cambridge University Press, 1st edition, 2000. ISBN 0-521-64408-9.