16 Sequencing and Monads

In Haskell, the notion of a *Monad* abstracts from a common program structure like that of

```
> return :: a -> Parser a
> (>>=) :: Parser a -> (a -> Parser b) -> Parser b
```

There is a predefined type class (roughly)

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  m >> k = m >>= const k
```

which it makes sense to define only if the components satisfy the laws

$$\begin{aligned} \mathit{return}\ x \gg\!= k &= k\ x \\ m \gg\!= \mathit{return} &= m \\ m \gg\!= (\lambda x \to (k\ x \gg\!= h)) &= (m \gg\!= k) \gg\!= h \end{aligned}$$

These laws express that *return* is the left and right unit of bind, an the third law is an associative law.

The type of *Parser* can be made an instance, although we need a name for the type function, and a type constructor to identify that type:

```
> newtype Parser a = Parser { parse :: (String -> [(a, String)]) }

> instance Monad Parser where
>   return x = Parser (\xs -> [(x,xs)])
>   Parser p >>= f  = Parser (\xs ->
>           [ (v,zs) | (a,ys) <- p xs, (v,zs) <- f a `parse` ys ])
```

It might not be immediately obvious, but this satisfies the monad laws.

There are many other instances of the *Monad* class which may be illuminating, for example

```
instance Monad []  where
  return x = [ x ]
  xs >>= f = [ y | x <- xs, y <- f x ]
```

In this case $xs \gg\!= f = \mathit{concatMap}\ f\ xs = \mathit{concat}\ (\mathit{map}\ f\ xs)$. It is perhaps easier to check that the monad laws hold here. Similarly with

```
instance Monad Maybe where
  return = Just
  Nothing >>= f = Nothing
  Just x  >>= f = f x
```

16.1 Kleisli composition

It might not be obvious that

$$m \gg= (\lambda x \rightarrow (k\ x \gg= h)) \quad = \quad (m \gg= k) \gg= h$$

is an associative law: what exactly is associative? Define the *Kleisli composition* by

```
> (>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
> (f >=> g) x = f x >>= g
```

then the monad laws can be expressed as properties of $(\gt\!\!=\!\!\gt)$

$$\begin{aligned} \mathit{return} \gt\!\!=\!\!\gt k &= k \\ h \gt\!\!=\!\!\gt \mathit{return} &= h \\ j \gt\!\!=\!\!\gt (k \gt\!\!=\!\!\gt h) &= (j \gt\!\!=\!\!\gt k) \gt\!\!=\!\!\gt h \end{aligned}$$

The Kleisli composition in the list monad is

$$\begin{aligned} & (f \gt\!\!=\!\!\gt g)\ x \\ = \quad & f\ x \gg= g \\ = \quad & [z \mid y \leftarrow f\ x,\ z \leftarrow g\ y] \end{aligned}$$

16.2 **do** notation

Haskell provides a special notation for Monad-valued expressions.

$$\begin{aligned} \mathbf{do}\ \{x \leftarrow m; \mathit{stuff}\} &= m \gg= (\lambda x \rightarrow \mathbf{do}\ \{\mathit{stuff}\}) \\ \mathbf{do}\ \{m; \mathit{stuff}\} &= m \gg \mathbf{do}\ \{\mathit{stuff}\} \\ \mathbf{do}\ \{m\} &= m \end{aligned}$$

As with other constructs, the braces and semicolons are usually omitted when the **do** expressions are laid out on several lines, using the offside rule.

In the case of the Monad of lists,

$$
\begin{array}{rl}
& \mathbf{do}\ \{x \leftarrow xs; y \leftarrow f\ x; return\ (g\ x\ y)\} \\
= & xs \gg\!= (\lambda x \rightarrow f\ x \gg\!= (\lambda y \rightarrow return\ (g\ x\ y))) \\
= & concat\ (map\ (\lambda x \rightarrow f\ x \gg\!= (\lambda y \rightarrow return\ (g\ x\ y)))\ xs) \\
= & concat\ (map\ (\lambda x \rightarrow concat\ (map\ (\lambda y \rightarrow return\ (g\ x\ y))\ (f\ x)))\ \ xs) \\
= & concat\ (map\ (\lambda x \rightarrow concat\ (map\ (\lambda y \rightarrow [g\ x\ y])\ (f\ x)))\ \ xs) \\
= & concat\ (map\ (\lambda x \rightarrow concat\ [[g\ x\ y] \mid y \leftarrow f\ x])\ \ xs) \\
= & concat\ (map\ (\lambda x \rightarrow [g\ x\ y \mid y \leftarrow f\ x])\ \ xs) \\
= & concat\ [[g\ x\ y \mid y \leftarrow f\ x] \mid x \leftarrow xs] \\
= & [g\ x\ y \mid x \leftarrow xs, y \leftarrow f\ x]
\end{array}
$$

so the similarity between **do**-notation and list comprehension is deliberate. (You might wonder why list comprehenion notation is not used for comprehenions in other monads, but that way madness lies.)

In terms of **do** notation, the Monad laws are

$$
\left.\begin{array}{l}
\mathbf{do} \\
\quad y \leftarrow return\ x \\
\quad k\ y
\end{array}\right\} \quad = \quad k\ x
$$

$$
\left.\begin{array}{l}
\mathbf{do} \\
\quad x \leftarrow m \\
\quad return\ x
\end{array}\right\} \quad = \quad m
$$

$$
\left.\begin{array}{l}
\mathbf{do} \\
\quad x \leftarrow m \\
\quad \mathbf{do} \\
\quad\quad y \leftarrow k\ x \\
\quad\quad h\ y
\end{array}\right\} \quad = \quad \left\{\begin{array}{l}
\mathbf{do} \\
\quad y \leftarrow \ \mathbf{do} \\
\quad\quad\quad\quad x \leftarrow m \\
\quad\quad\quad\quad k\ x \\
\quad h\ y
\end{array}\right.
$$

The associative law justifies writing both sides as

$$
\begin{array}{l}
\mathbf{do} \\
\quad x \leftarrow m \\
\quad y \leftarrow k\ x \\
\quad h\ y
\end{array}
$$

and so on, and the unit laws allow for unnecessary *return* calls to be removed.

The beauty of *do* notation is that the plumbing in

```
> some :: Parser a -> Parser [a]
> some p = p >>= ps
>          where ps a = many p >>= done
>                       where done as = return (a:as)
```

or

```
> some p = p >>= (\a -> many p >>= (\as -> return (a:as)))
```

is much easier to express as

```
> some p = do
>            a <- p
>            as <- many p
>            return (a:as)
```

where of course the **do** expression in this example is a value in the *Parser* monad.

16.3 Monadic Input and Output

The reason that monads first became such an important part of Haskell is that they capture the idea of sequencing effects, and this gives a way of sequencing the effects of input and output without leaving the functional programming language.

In the *Parser* monad the effects being sequenced are the extent to which each parser consumes the input string. Parsers which appear in sequence in a *do* expression consume (if any) parts of the input string which appear in the same sequence in the string.

In the same way the effects in the *IO* monad are interactions with the real world, which happen in the order described by their sequence in a *do*. A thing of type *IO* a is an interaction with the real world which yields a value of type a, so for example

```
readFile :: FilePath -> IO String
```

so when applied to the name of a file *readFile* produces an *IO* value from which you can get a *String* containing the sequence of characters in the file.

Simple output operations like

```
putStr :: String -> IO ()
```

have nothing significant to return, so produce an *IO* value from which you can get () which is the only value of the type () of null-tuples.

16.4 Applicatives and Functors

In any monad, you can define an operation (called *ap* or *apply*) that looks like application of a monadic function to a monadic argument:

```
(<*>) :: Monad m => m (a -> b) -> m a -> m b
fs <*> xs = do { f <- fs; x <- xs ; return (f x) }
```

(Notice the parentheses in *return* $(f\ x)$, because *return* here is a function, not a syntactic component of the *do* construct.)

It is convenient for now also to have a different name for

```
pure :: Monad m => a -> m a
pure = return
```

In the case of the list monad, *pure* makes a singleton list and *apply* would do all of the applications of a function to an argument that you would find in the Cartesian product of a list of functions and a list of arguments. In the case of the Parser monad *pure* makes a parser that successfully returns a given value without consuming anything from the string, and *apply* would parse a function followed by an argument, and the result is a parse in which the two bits of input are both consumed and the result is the result of applying the function to the argument.

This *apply* operation is well behaved in many ways:

$$\begin{aligned} \mathit{pure}\ \mathit{id} <\!\!*\!\!> v &= v \\ \mathit{pure}\ (\cdot) <\!\!*\!\!> u <\!\!*\!\!> v <\!\!*\!\!> w &= u <\!\!*\!\!> (v <\!\!*\!\!> w) \\ \mathit{pure}\ f <\!\!*\!\!> \mathit{pure}\ x &= \mathit{pure}\ (f\ x) \\ u <\!\!*\!\!> \mathit{pure}\ y &= \mathit{pure}\ (\$\ y) <\!\!*\!\!> u \end{aligned}$$

These are the qualifications for being an instance of the *Applicative* type class:

```
class Applicative m where
  pure :: a -> ma
  (<*>) :: m (a -> b) -> m a -> m b
```

so every monad is necessarily an *applicative*. (If you are bothered that *applicative* is not a noun, you are right: it turns out to be an *applicative functor*.)

Given an applicative functor m (in particular, given any monad m) it is possible to define

```
(<$>) :: Applicative m => (a -> b) -> (m a -> m b)
f <$> xs = pure f <*> xs
```

which obeys the laws for *map*

$$\begin{aligned} \mathit{id} <\!\!\$\!\!> \mathit{xs} &= \mathit{xs} \\ (f \cdot g) <\!\!\$\!\!> \mathit{xs} &= f <\!\!\$\!\!> (g <\!\!\$\!\!> \mathit{xs}) \\ &= ((f <\!\!\$\!\!>) \cdot (g <\!\!\$\!\!>))\ \mathit{xs} \end{aligned}$$

so every monad is a legitimate instance of the class

```
class Functor m where
  fmap :: (a -> b) -> m a -> m b
(<$>) = fmap
```

In fact, the three classes are defined by headings that say

```
class Functor m where ...
class Functor m => Applicative m where ...
class Applicative m => Monad m where ...
```

which obliges you, when defining an instance of *Monad* first to define the instance of *Applicative* and before that the instance of *Functor*.

16.5 Monadic Join

In the list monad, $xs \gg= f = concat\ (map\ f\ xs)$. In every monad, it turns out, this same factorisation is possible. The equivalent of *map* is of course $(<\$>)$, and the equivalent of *concat* is

```
> join :: Monad m => m (m a) -> m a
> join m = m >>= id
```

so in particular in the list monad

$$\begin{aligned} & join\ m \\ = \;& m \gg= id \\ = \;& concat\ (map\ id\ m) \\ = \;& concat\ m \end{aligned}$$

and in general

$$m \gg= f \quad = \quad join\ (f <\$> m)$$

This means that it is possible to define the operations on a monad by giving not *return* and $(\gg=)$ but *return*, $(<\$>)$ and *join*. Sometimes that is more intuitive.