Mónadas en Haskell

Garcia Ruiz, Eduardo Andrés Martinez, Pablo

January 19, 2016

Contents

1 Introducción

1.1 ¿Qué necesitamos saber para entender la clase Monad?

La clase Monad está diseñada para describir tipos/estructuras de datos en los que sea necesario manejar un *contexto* de forma transparente al usuario. El concepto de *contexto* y cómo es manejado (de una forma muy cómoda) por una mónada, es lo que hace a esta clase tan abstracta. Una vez captada esta idea, el siguiente paso será comprender la potencia que tiene trabajar con Monads para estos tipos de datos.

Cabe destacar que toda persona que haya trabajado alguna vez con Haskell habrá utilizado, probablemente sin saberlo, tipos monádicos: Maybe, Either, IO o List son todos Monads. Esto es, en sí mismo, una evidencia de la capacidad de transparencia que proporcionan las mónadas.

El presente documento intentará describir la funcionalidad de la clase Monad a través de varios ejemplos, empezando por el caso (prácticamente trivial) del tipo Maybe en Haskell. Poco a poco iremos introduciendo ejemplos más complejos hasta que lleguemos a trabajar con un tipo de datos bastante potente que, sin embargo, podrá ser utilizado por cualquier usuario sin necesidad de tener conocimiento sobre de una mónadas. Esperamos así que las ventajas que tienen las mónadas queden en evidencia.

No obstante, antes de empezar a presentar códigos en los que se implementen mónadas, el lector debe familiarizarse con la clase Monad y con su base teórica. Dado que saltar directamente a la abstracción que representan las mónadas es, probablemente, un reto demasiado ambicioso, introduciremos antes dos "type classes" de Haskell que nos ayudarán a empezar a entender el concepto de *contexto* del que se habló anteriormente. Estas clases son Functor y Applicative y, como se explicará mas adelante, deben entenderse como supersclases de Monad.

1.2 Pero, ¿qué es el *contexto*?

La siguiente estructura de datos simplemente envuelve el tipo de datos genérico t:

```
data Type t = Wrap t
```

Debe entenderse que, al envolverlo, le estamos dando un *contexto* al contenido de tipo t. Éste indicará dónde puede ser utilizado el valor y **de qué forma será manejado** (al instanciar distintas clases e implementar funciones que manejen Type). Así pues, toda estructura de datos está, de alguna forma, proporcionando un contexto (un entorno o un significado) al tipo o tipos de datos más básicos de los que está compuesta.

2 Functors

Empezamos por algo simple, la type class Functor. Ésta busca resolver la problemática más básica de trabajar con una estructura de datos: ¿cómo modificamos los valores que están contenidos en ella?

2.1 Functor type class

La definición de la clase es la siguiente:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

La función *fmap* toma como primer argumento una función que lleva de cualquier tipo genérico a a otro b y, como segundo argumento, un valor de tipo a dentro de un **envoltorio que le da contexto**. Dicho envoltorio f es la estructura de datos que instanciaremos como Functor. La salida de *fmap* será un valor del tipo b envuelto en la misma estructura de datos en la que se encontraba a, es decir, manteniendo el contexto.

Por tanto, cuando una estructura de datos se ha definido como Functor, sabemos que sobre ésta se podrán **aplicar funciones que transformen su contenido manteniendo el contexto**.

2.2 Leyes de la clase Functor

Según la definición de Functor que da la documentación de Haskell, toda instancia de esta clase debe cumplir las siguientes leyes:

```
fmap id = id
fmap (p . q) = (fmap p) . (fmap q)
```

Teniendo una estructura de datos instanciada como Functor:

- La **ley de la identidad** indica que aplicar la función identidad al contenido de la estructura de datos debe ser siempre equivalente a aplicar la función identidad directamente sobre la propia estructura. Esto implica que, cuando el contenido del envoltorio (la estructura de datos) se mantiene igual, el contexto que éste maneja tampoco puede cambiar.
- La **ley de la composición** indica que, al aplicar la función compuesta de otras dos cualesquiera q y p sobre el contenido de la estructura de datos, el resultado debe ser equivalente a transformar ésta aplicando primero q y después p sobre la salida de la anterior. Esto implica que las distintas transformaciones a realizar sobre la estructura de datos pueden componerse y efectuarse en una única llamada a *fmap*.

2.3 This Maybe a Functor!

Introducimos a continuación un conocido tipo de datos de Haskell, el tipo Maybe, y veremos cómo éste puede ser instanciado como Functor y cumple las leyes correspondientes:

```
data Maybe a = Just a | Nothing

instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```

La información que mantiene Maybe permite distinguir si el resultado de una computación ha devuelto un valor válido (*Just x*) o, por el contrario, no se ha podido evaluar (*Nothing*). Estamos, por tanto, manejando con Maybe el **contexto de las funciones parciales**.

Asimismo, podemos comprobar que cumple las leyes que debe verificar todo Functor:

fmap id = id

```
fmap id (Nothing) = Nothing = id (Nothing)
fmap id (Just x) = Just (id x) = Just x = id (Just x)
```

fmap (p . q) = (fmap p) . (fmap q)

```
fmap (p . q) Nothing = Nothing = fmap p Nothing
                               = fmap p (fmap q Nothing)
                               = ((fmap p) . (fmap q)) Nothing
fmap (p . q) (Just x) = Just ((p . q) x)
                      = Just ((p (q x))
                      = fmap p (Just (q x))
                      = fmap p (fmap q (Just x))
                      = ((fmap p) . (fmap q)) (Just x)
```

3 Applicative Functors

Los applicative functors pueden ser concebidos como una mejora de los functors y en Haskell los podemos encontrar bajo la clase **Applicative** en el módulo **Control.Applicative**.

3.1 Límites de la type class Functor

Cuando hablabamos de **Functor** hablabamos de mapear funciones en un contexto, es decir, sacar el valor de dicho contexto, aplicar la función en cuestión sobre dicho valor y *envolver* el valor devuelto por dicha función en el *contexto* en el que se encontraba.

La función *fmap* aplica una función del tipo *a–>b*, esta función necesita un valor *a* como argumento, un ejemplo de estas funciones pueden ser:

- id, (3+), (++"foo"), (5*), (3:),...

Dicha función es mapeada sobre una sola estructura de datos que contiene un valor envuelto en su contexto, pero ¿qué debemos hacer si queremos mapear una función que necesita más de un parámetro? Dichas funciones podrían ser:

- (+), (*), (:), (++), ...

Usando fmap sobre este tipo de funciones obtendríamos una *función currificada envuelta* en el contexto de la estructura de datos. A través de fmap ya no podríamos volver a mapear dicha función currificada ya que, por como está definido fmap, no nos permite mapear funciones que están envueltas en un contexto, ni nos permite sacar la función del contexto.

3.2 Applicative type class

La definición de la clase es la siguiente:

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>)  :: f (a -> b) -> f a -> f b
```

Esta definición nos da bastante información:

- **Restricción de clase:** Si queremos que un tipo forme parte de la typeclass *Applicative* primero debe formar parte de *Functor*, por lo que podremos usar *fmap* en la instancia de dicho Applicative.
- **pure:** Esta función recibe un valor y devuelve un *applicative functor* con dicho valor dentro de él. Podemos decir que *pure* coge un valor y lo coloca en un *contexto por defecto*, un contexto mínimo que contiene dicho valor.
- **<*>:** La declaración que encontramos en la función <*> es muy parecida a la de la función *fmap*. Mientras que fmap coge una función, un functor con cierto valor, y aplica la función dentro del functor; <*> coge un functor que contiene una función en él, otro functor con cierto valor, y saca la función del primer functor para mapearla sobre el segundo. Permite así mapear funciones currificadas envueltas.

3.3 Cogiendo soltura con <*>

Un Functor mapea una función sobre un solo functor a través de fmap, sin embargo usando un contexto de applicative podemos operar sobre varios functors con una sola función de la siguiente manera:

```
pure g <*> f x_1 <*> f x_2 <*> ... <*> f x_N
```

Donde g es una función, f es el applicative y

$$fx_1, fx_2, ..., fx_N$$

son las diferentes estructuras sobre las que mapeamos la función. Observamos que gracias al operador <*> podemos mapear funciones N-arias envueltas en un contexto sobre N estructuras con el mismo contexto. Como el operador <*> es asociativo por la izquierda, sería lo mismo escribir:

```
((((pure g <*> f x_1) <*> f x_2) <*> ...) <*> f x_N)
```

Lo primero que se evalúa es la función *pure*, que coloca la funcion g dentro del aplicative de tipo *f*, obteniendo ...

```
((((f g <*> f x_1) <*> f x_2) <*> ...) <*> f x_N)
```

El siguiente paso sería mapear la función g sobre el primer envoltorio ...

```
(((f (g x_1) <*> f x_2) <*> ...) <*> f x_N)
```

Dónde (g x_1) debe de ser una función currificada a la espera de N-1 argumentos. Cómo el operador va paso-por-paso de izquierda a derecha, finalmente obtendríamos la siguiente expresión:

```
f (g x_1 x_2 ... x_N)
```

3.4 ¡Un nuevo operador! <$>

Una de las leyes de la clase *Applicative* es *pure g <*> x = fmap g x* , ya hablaremos de ella en el siguiente apartado. Por lo que escribir la siguiente expresión:

```
pure g <*> f x_1 <*> f x_2 <*> ... <*> f x_N
```

Es equivalente a escribir:

```
fmap g x_1 <*> f x_2 <*> ... <*> f x_N
```

Para mayor facilidad en la sintaxis, **Control.Applicative** exporta la función **<$>** que no es más que un *fmap* en notación infija. Su definición es la siguiente:

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

3.5 Leyes de la clase Applicative

Según la definición de Applicative que da la documentación de Haskell, toda instancia de esta clase debe cumplir las siguientes leyes:

```
pure f <*> x = fmap f x
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
```

- **Ley del fmap:** Indica que introducir una funcion f en un contexto y luego mapearla con el operador <*> sobre una estructura con cierto contexto, es equivalente a mapear la función sobre dicha estructura a través de *fmap*.
- **Ley de la identidad:** Indica que introducir la función identidad en un contexto y mapearla con el operador <*> sobre una estructura con contexto, no debe modificar dicha estructura.
- **Ley de la composición:** Nos informa de que componer las funciones u y v y mapear el resultado sobre una estructura, es igual que mapear primero v y luego u.
- **Ley de pure:** Introducir una función en el *contexto por defecto* y mapearlo sobre un valor, también envuelto en el contexto por defecto, debe ser igual que introducir en el *contexto por defecto* dicha función mapeada sobre el valor.
- **Ley de precedencia:** Nos dice que mapear una función sobre un *contexto por defecto* es equivalente a introducir un valor en un *contexto por defecto* al que le quitamos precedencia, por lo que en lugar de mapear la funcion de la izquierda sobre el contexto de la derecha, en este caso (como al contexto de la izquierda le ha quitado precedencia) la función de la derecha se mapea sobre el contexto de la izquierda.

3.6 This Maybe an Applicative!

Ahora instanciaremos Maybe como Applicative, utilizando para ello las funciones propias de Functor:

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> mx = fmap f mx
```

El contexto del **Maybe** es la posibilidad de que la computación en cuestión pueda dar lugar a fallo. El contexto mínimo por defecto del *Maybe* es *Just*, como podemos observar en la función *pure*. Observamos que por definición cuando mapeamos *Nothing* sobre lo que sea, siempre obtenemos *Nothing*, y que cuando la función a mapear está envuelta en un *Just*, la mapeamos directamente sobre el functor *mx* en cuestión.

Podemos comprobar que el tipo Maybe cumple las leyes que debe verificar todo Applicative:

pure f <*> x = fmap f x

```
Si x = Just y
pure f <*> (Just y) = (Just f y)
                    = fmap f (Just y)

Si x = Nothing
pure f <*> Nothing = Nothing
                   = fmap f Nothing
```

pure id <*> v = v

```
Si v = Just x
pure id <*> (Just x) = (Just id) <*> (Just x)
                     = fmap id (Just x)
                     = (Just id x)
                     = (Just x)
Si v = Nothing
pure id <*> Nothing = (Just id) <*> Nothing
                    = fmap id Nothing
                    = Nothing
```

pure (.) <*> u <*> v <*> w = u <*>(v <*>w)

```
Consideramos que [u = Just f] y [v = Just g]. En caso de que alguna
    fuese Nothing, el resultado seria Nothing y la demostracion es
    trivial.

Si w = Just x
pure (.) <*> u <*> v <*> (Just x) = Just (f.g) <*> (Just x)
                                  = fmap (f.g) (Just x)
                                  = (Just (f.g) x)
                                  = (Just f (g x))
                                  = fmap f (Just (g x))
                                  = (Just f) <*> (Just (v x))
                                  = (Just f) <*> fmap v (Just x)
                                  = u <*> ((Just g) <*> (Just x))
                                  = u <*> (v <*> (Just x))
Si w = Nothing
pure (.) <*> u <*> v <*> Nothing = Just (f.g) <*> Nothing
                                 = fmap (f.g) Nothing
                                 = Nothing
                                 = fmap f Nothing
                                 = (Just f) <*> Nothing
                                 = (Just f) <*> fmap v Nothing
                                 = (Just f) <*> ((Just g) <*> Nothing)
                                 = u <*> (v <*> Nothing)
```

pure f <*> pure x = pure (f x)

```
pure f <*> pure x = pure f <*> (Just x)
                  = (Just f) <*> (Just x)
                  = fmap f (Just x)
                  = (Just (f x))
                  = pure (f x)
```

u <*> pure y = pure ($ y) <*> u

```
Consideramos que [u = Just f]. En caso de que fuese Nothing, el
    resultado seria Nothing y la demostracion es trivial.

u <*> pure y = (Just f) <*> (Just y)
             = fmap f (Just y)
             = (Just (f y))
             = (Just f $ y)
             = fmap f (Just $ y)
             = (Just $ y) <*> (Just f)
             = pure ($ y) <*> u
```

4 Monads

Por fin llegamos al tema principal del documento. Aquí explicaremos qué es un Monad utilizando las herramientas que nos ha proporcionado definir las clases Functor y Applicative. Sin embargo, antes de continuar cabe destacar un interesante detalle histórico.

Cuando se busca información sobre la clase Monad se pueden encontrar distintas fuentes; algunas (normalmente las más antiguas) tratan esta clase como un elemento independiente en la jerarquía de clases de Haskell, sin ninguna superclase. No obstante, las más recientes suelen insistir en el hecho de que todo tipo instanciado como Monad debe siempre definirse también como Applicative.

Esto se debe a que Monad es una clase que lleva mucho tiempo existiendo en Haskell y, que desde sus inicios, ha servido para implementar tipos tan utilizados como Maybe, IO o List. Por el contrario, Applicative es una clase considerablemente reciente, que ni siquiera está incluida en el paquete básico de Prelude. Lo curioso es que todo tipo que instancie la clase Monad puede instanciar también Applicative y cumplir sus leyes, de donde se razona que todo Monad es un Applicative, es decir, Applicative es superclase de Monad.

Así pues, en versiones anteriores a GHC 7.10, un tipo puede instanciar la clase Monad y no instanciar Applicative. A partir de dicha versión, Applicative se considera superclase de Monad y, por tanto, instanciarla es obligatorio.

4.1 Límites de la type class Applicative

Hemos visto que Applicative permite realizar transformaciones sobre tipos de datos con cierto contexto de una forma más cómoda que Functor. En Applicative empieza a vislumbrarse el concepto fundamental de las mónadas: Las transformaciones que se realizan sobre la estructura de datos también pueden tener cierto contexto asociado; de ahí que la función <*> reciba como primer argumento una función encapsulada en un Functor. Este hecho permite aprovechar herramientas como la currificación de funciones sobre estructuras de datos que, con Functor, era imposible (como ya se vió en el apartado 3.3).

Sin embargo, hasta el momento no ha sido posible para el usuario controlar directamente el contexto de la estructura de datos. Por ejemplo, con la definición de Maybe como Applicative no es posible conseguir que *(-) <$> Just N <*> Just 1* devuelva *Nothing* cuando *N* sea un número negativo. Esto se debe a que *(-)* afecta al contenido de la estructura de datos, no a la estructura en sí, pues su tipo es *f (a ->b)*.

Con la clase Monad, las funciones que introduce el usuario no sólo tendrán asociado cierto contexto (como en el caso de Applicative, con sus correspondientes ventajas) sino que, además, tendrán la capacidad de **actualizar el contexto de la estructura de datos**, permitiendo al usuario controlar cómo evoluciona el contexto dependiendo del contenido de la estructura.

4.2 Monad type class

La definición de la clase es la siguiente:

```
class (Applicative f) => Monad f where
    return :: a -> f a
    (>>=) :: f a -> (a -> f b) -> f b
    fail :: String -> f a
```

- **Restricción de clase:** Si queremos que un tipo forme parte de la typeclass *Monad* primero debe formar parte de *Applicative*, por lo que podremos usar *pure* y <*> en la instancia de dicho *Monad*. Asimismo, dado que *Functor* es superclase de *Applicative*, también contaremos con *fmap*.

- **return:** Esta función recibe un valor y devuelve una *mónada* con dicho valor dentro de él. Así pues, realiza exáctamente la misma tarea que la función *pure* de *Applicative*, envolviendo el valor introducido como argumento dentro de un contexto por defecto.

- **>>=:** Esta función se conoce como *bind* y es la que da a la clase Monad su enorme potencial. Para ser capaces de entenderla debemos recurrir a observar su parecido con la función <*> de *Applicative*. Ambas devuelven como salida un *applicative functor* envolviendo a cierto tipo genérico *b*. Respecto a la entrada, las dos reciben argumentos similares (aunque en orden distinto): un tipo genérico *a*, también envuelto, y una función que, ignorando los envoltorios, lleva del tipo *a* al tipo *b*.

- **fail:** Esta función es utilizada internamente por Haskell para manejar errores (indicados como String y dados como argumento), de forma que se devuelva el Monad con contexto y contenido más adecuado. No es obligatorio implementar esta función cuando se instancia la clase Monad, pero es muy recomendable hacerlo, sobre todo si vamos a trabajar con la *do notation* (explicada más adelante).

4.3 Cogiendo soltura con *bind*

Como hemos introducido en el apartado anterior, el operador *bind* (>>=) se diferencia del operador <*> esencialmente en cómo debe especificar el usuario la función que lleva del tipo genérico *a* al tipo genérico *b*.

Comentábamos en la introducción de esta sección que los Monad iban a permitir al usuario **actualizar el contexto de la estructura de datos** a su antojo. Esto no era posible en los Applicative porque la función que se daba como argumento al utilizar <*> era del tipo *f (a -> b)*, que no era más que una **transformación del contenido** de la estructura de datos, aunque englobada en su mismo contexto para facilitar ciertas operaciones.

Sin embargo, en el caso de Monad, la función a dar como argumento en *bind* es del tipo *a ->f b*. ¡Toma el contenido de la estructura de datos y devuelve un resultado envuelto con cierto contexto! He aquí lo que hace especial al operador *bind*: en función del contenido actual de la estructura, el usuario puede especificar no sólo un nuevo contenido, sino una actualización del contexto de la estructura de datos.

Así pues, la función *bind* debe encargarse de extraer el contenido de la estructura de datos que recibe como primer argumento, aplicarle a éste la función que introduzca el usuario como segundo argumento (que indicará la actualización del contenido y el contexto de la estructura) y, finalmente, dar como salida una estructura de datos coherente a la trasnformación requerida por el usuario.

Debe destacarse que el resultado de la operación *bind* no es necesariamente la estructura de datos que el usuario ha indicado como salida de la función que ha introducido, sino que ésta se aplicará de forma consecuente al contexto que tuviese el Monad dado como primer argumento del *bind*. Se entiende por tanto que, al usar *bind*, el usuario delega en dicho operador la tarea de actualizar el contexto, siguiendo las pautas que haya definido al introducir su función y manteniéndolo siempre coherente a la estructura de datos.

De esta forma, *Nothing* >>= *f* seguirá devolviendo *Nothing* como lo haría en un Applicative pero, *Just x* >>= *f* podría devolver un *Just y* o un *Nothing* en función de cómo haya construido el usuario la función *f*.

Por otro lado, el usuario no siempre querrá realizar distintas transformaciones de la estructura de datos según el valor que ésta contenga. Existirán casos en los que el usuario simplemente quiera realizar una actualización de la estructura de datos, sin importar qué contenido tenga ésta. En tal caso, puede utilizarse el sinónimo (>>) definido automáticamente por Haskell del siguiente modo:

```
(>>) :: (Monad m) => m a -> m b -> m b
(>>) m1 m2 = m1 >>= (\_ -> m2)
```

Debe entenderse que en caso de utilizar este operador, aunque el usuario no necesite tener en cuenta el contenido del Monad *m1* para hacer la transformación que desea, el operador *bind* sigue teniendo que manejar el contexto de *m1*. Por ello, no debemos caer en la equivocación (muy común) de creer que la salida del operador (>>) será siempre *m2*. La salida real será *m1* con el contexto actualizado para corresponder con aquél de *m2*. Todo esto quedará mucho más claro cuando empecemos a trabajar con ejemplos, sobre todo cuando lleguemos al *State Monad*, donde se utiliza el operador (>>) con bastante frecuencia.

4.4 Leyes de la clase Monad

Según la definición de Monad que da la documentación de Haskell, toda instancia de esta clase debe cumplir las siguientes leyes:

```
return x >>= f = f x
m >>= return = m
(m >>= f) >>= g = m >>= (\y -> f y >>= g)
```

- La **ley de la identidad izquierda** indica que aplicar un *bind* sobre un monad cuyo contexto sea el dado por defecto por *return*, deberá dar siempre el mismo resultado que aplicar directamente la función dada por el usuario sobre el contenido de dicho monad. Debe destacarse que esto no es algo que pueda asumirse para cualquier otro monad que no se haya construido por *return*, dado que un contexto distinto al de por defecto podría suponer que la salida no fuese *f a*.

- La **ley de la identidad derecha** indica que hacer, sobre un monad cualquiera, un *bind* con la función *return* debe devolver siempre el monad original. Esto implica que la función *return* nunca cambiará ni el contenido ni el contexto del monad.

- La **ley de la asociatividad** indica que, de hacer dos *bind* consecutivos sobre un monad, el resultado debe ser el mismo independientemente de qué bind se compute antes, si el izquierdo o el derecho. La única diferencia es que, de computar primero el bind derecho, debemos considerar todos los posibles Monad que hayan podido resultar del primer bind.

4.5 This Maybe a Monad!

Ahora instanciaremos Maybe como Monad, utilizando para ello las funciones propias de Applicative y Functor:

```
instance Monad Maybe where
    return k = Just k
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
    fail _ = Nothing
```

En este caso es muy fácil entender por qué el contexto por defecto está asociado a Just (computación efectuada con éxito) y Nothing está asociado al contexto manejado por la función *fail* en caso de darse cualquier error. Asimismo, vemos que *bind* tiene en cuenta el contexto del monad que recibe como argumento: si la computación ha fallado (Nothing), el resultado tras aplicar la función indicada por el usuario sigue siendo una notificación de que en algún punto ha ocurrido un error (siempre devuelve Nothing), mientras que si la computación se ha efectuado correctamente hasta el momento (Just x), el resultado será el que devuelva la función indicada por el usuario, pudiendo ser ésta Nothing o Just y.

Podemos comprobar que el tipo Maybe cumple las leyes que debe verificar todo Monad:

return x >>= f = f x

```
return x >>= f = (Just x) >>= f = f x
```

m >>= return = m

```
Si m = Nothing
Nothing >>= return = Nothing
Si m = Just x
(Just x) >>= return = return x = Just x
```

(m >>= f) >>= g = m >>= (\y -> f y >>= g)

```
Si m = Nothing
(Nothing >>= f) >>= g = Nothing >>= g = Nothing = Nothing >>= (\y -> f
    y >>= g)
Si m = Just x
(Just x >>= f) >>= g = f x >>= g = Just x >>= (\y -> f y >>= g)
```

4.6 La sintáxis *do notation*

Hemos visto que *bind* es una operación muy potente pero, si hubiésemos implementado algún Monad más complejo e intentado utilizarlo, habríamos sufrido lo engorroso que es trabajar con un operador que recibe como segundo argumento una función (que en muchos casos se indicará mendiante lamda-expresiones) y que, normalmente, se utiliza de forma reiterada sobre el mismo monad. Un código que realice varios binds consecutivos, con sus correspondientes funciones como lambda-expresiones, llega a ser prácticamente ilegible.

Para solucionar este problema surge la *do notation*, una sintáxis que permite utilizar el operador *bind* de una forma cómoda y transparente:

```
do v <- m1
   m2

Equivale a:

m1 >>= (\x -> case x of v -> m2
                        _ -> fail "s")
```

De esta forma, la primera línea de la *do notation* toma el contenido del monad *m1* y comprueba que coicida con el patrón *v*. Si lo hace, realiza un *bind* con *m2*; en otro caso lo hace con el resultado de *fail*.

Incluyendo más líneas pueden realizarse sucesivos *bindings* cómodamente, permitiendo construir códigos **sospechosamente imperativos**:

```
restaUno :: Maybe Int -> Maybe Int
restaUno m = do x <- m
                True <- return (x > 0)
                return (x-1)
```

En caso de introducir como argumento *Nothing*, la función devolverá *Nothing*, debido a la definición del *bind* sobre dicho valor del Monad. Si se le introduce un *Just n* y devolverá *Just (n-1)* cuando *n* es mayor que cero y *Nothing* cuando es menor.

Este código implementado utilizando el operador (>>=) sería prácticamente ilegible, sin embargo con *do notation* es bastante intuitivo: Primero se extrae el valor *x* de *m = Just x*, después se comprueba que evaluar *x >0* devuelva *True* y, en tal caso, devuelve *Just (x-1)*. De darse el caso de que *x >0* devolviese *False*, el patrón de la segunda sentencia no se satisfacería, por lo que se llamaría a *fail*, obteniendo como resultado *Nothing*, que se propagaría hasta la salida debido a la definición de *bind* para este valor del Monad.

De igual forma que existe el operador (>>) para cuando el contenido del monad no es de interés, hay un equivalente en *do notation*:

```
do m1
   m2

Equivale a:

m1 >> m2 = m1 >>= (\_ -> m2)
```

4.7 Monad y sus superclases

Llegados a este punto, el lector ya debe estar mínimamente familiarizado con las tres *typeclasses* que han sido definidas, y sus correspondientes funciones. Durante el documento se ha insistido en que Functor es superclase de Applicative y éste superclase de Monad, cada uno suponiendo una mayor potencia para manejar contextos que el anterior.

En Haskell, decir que una clase A es superclase de otra B implica que todo tipo que instancie la clase A debe instanciar también la clase B. A continuación, vamos a mostrar cómo cualquier tipo definido como Monad puede ser, automáticamente, definido como Applicative y como Functor, demostrando así que son superclases de Monad.

Dado un Monad m siempre podremos instanciar Applicative:

```
instance Applicative m where
    pure = return
    mf <*> mx = do f <- mf
                   x <- mx
                   return (f x)
```

Como ya se discutió anteriormente, las funciones *pure* y *return* son idénticas. Por otro lado, el operador $<*>$ simplemente debe: primero extraer la función f del envoltorio, después extraer el valor x del envoltorio y, finalmente, envolver el resultado de aplicar $f\ x$ en el contexto por defecto.

Dado un Applicative m siempre podemos instanciar Functor:

```
instance Functor m where
    fmap f mx = (pure f) <*> mx
```

La única diferencia de *fmap* respecto al operador $<*>$ es que, como ya se comentó, el segundo requería que la función dada como argumento estuviese encapsulada en cierto contexto. Así pues, *fmap* se puede implementar utilizando $<*>$, aplicando previamente *pure* a la función para darle el contexto por defecto.

5 Un Monad más interesante: List

Ahora instanciaremos List como Monad, utilizando para ello las funciones propias de Applicative y Functor:

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (fmap f xs)
    fail _ = []
```

En esta instancia de Monad el contexto mínimo por defecto es una lista con un valor, por lo que el contexto mínimo es un contexto donde aún no hay *no-determinismo*. El **bind** de esta instancia mapea la función *f* elemento por elemento de la lista y los va concatenando conforme va obteniendo los resultados (recordemos que *f :: a -> [a]*).

El Monad List nos abstrae un contexto bastante conocido en el ámbito de la computación como es el **no-determinismo**. Mientras que **5** es un valor **determinista**, **[3,6,15]** es un valor **no-determinista**, ese contexto es el que nos propone Haskell cuando hablamos del tipo List.

Por tanto, podemos de hablar de "posibles valores". Si tenemos "posibles funciones" y las mapeamos sobre"posibles valores" tendremos una serie de "posibles resultados" que recogerán todas las posibilidades de esta computación *no-determinista*. Veamos como sería usando List como un applicative:

```
ghci> (*) <$> [1,2,3] <*> [10,100,1000]
[10,100,1000,20,200,2000,30,300,3000]
```

Ahora veamos como sería usando List como un Monad, viendo la posibilidad de usar *do notation* y usando el *bind*:

```
ghci> do f <- [1,2,3]; v <- [10,100,1000]; return (f*v)
[10,100,1000,20,200,2000,30,300,3000]

ghci> [1,2,3]>>= (\x -> [10,100,1000] >>= (\f -> return(f*x)))
[10,100,1000,20,200,2000,30,300,3000]
```

5.1 *do notation* with List Monad

Tenemos el siguiente trozo de código:

```
do a<-[1,2,3]
   b<-[10,100,1000]
   return (a*b)

[10,100,1000,20,200,2000,30,300,3000]
```

Hasta ahora este tipo de calculos era posible de la siguiente manera:

```
ghci> [ a*b | a<-[1,2,3], b<-[10,100,1000]]

[10,100,1000,20,200,2000,30,300,3000]
```

¡¡Sí!! ¡¡Estamos realizando con *do notation* lo que antes realizabamos con *List Comprehension*!!

6 El State Monad

Ya hemos definido la clase Monad y el lector ha podido obtener experiencia en el uso e implementación de algunos tipos básicos de Haskell que seguramente había utilizado antes sin saber que instanciaban la clase Monad. Ahora es el momento de dar un paso más allá y explorar uno de los tipos monádicos más potentes: el State Monad.

Dicho Monad puede encontrarse en el paquete Control.Monad.State, sin embargo, nos encargaremos de realizar nuestra propia implementación en esta sección y explicarla detalladamente. Los ejemplos que requieran un State Monad utilizarán también nuestra implementación.

Lo más importante es, como hemos insistido a lo largo de todo el documento, qué representa el *contexto* en cada caso y cómo se maneja. Así pues, debe entenderse que el State Monad es utilizado para representar **computaciones que actualicen reiteradamente un estado**. Dicho estado debe ser almacenado de algún modo, y manejado por el Monad de una forma transparente al usuario. Por tanto, el estado de la computación es, precisamente, el contexto de este tipo monádico.

Un lector perspicaz se dará cuenta de que poder mantener y actualizar, de una forma transparente, el estado de una computación es algo que parece opuesto al paradigma funcional. De hecho, la existencia de estados mutables es precisamente una de las características principales de los lenguajes imperativos. Si a esto le añadimos la *do notation*, podremos vislumbrar el enorme potencial de State Monad para definir, de forma muy cómoda, comportamientos puramente imperativos dentro de un lenguaje que se mantiene puramente funcional.

6.1 Implementando el State Monad

Lo primero que debe discutirse es qué tipo tiene State Monad:

```
data MS s a = C (s -> (a,s))
```

Puede parecer muy complejo a simple vista, pero intentemos entenderlo paso por paso:

- El tipo *MS* está compuesto por otros dos tipos genéricos. El primero, *s*, corresponderá al tipo del estado almacenado por el State Monad, pudiendo ser desde un Int, un Char, un String, hasta la propia función State definida en los apuntes de la asignatura. El segundo es el tipo sobre el que trabajará el State Monad y que podrá manejar el usuario; es muy similar al tipo genérico de *Maybe a* o de *List a*.
- Todo elemento de este tipo es una función encapsulada, que recibe como entrada un valor del tipo estado que se haya elegido y devuelve un par contenido un valor de salida del tipo *a* y un nuevo estado. Por tanto, todo valor de este tipo **es una computación** que lleva de cualquier estado a un nuevo estado y devuelve cierto valor de salida.

- Será necesario implementar una función ***run*** que extraiga la computación de su envoltorio. Una vez extraída, se le proporcionará un *estado inicial* y el State Monad se encargará de manejarlo a lo largo de la computación, devolviendo finalmente la salida y el estado final.

- Cabe destacar que la clase Monad sólo permite que el tipo instanciado actúe como envoltorio de un único tipo genérico. Dicho de otro modo, nunca podríamos encontrar un *monad :: (Monad m) =>m s a*. Por tanto, el tipo instanciado como Monad no será *MS*, sino *(MS s)*, algo que tiene mucho sentido si se considera que *s* indica el estado del State Monad y, siendo éste su contexto, ¿qué Monad iba a poderse definir sin especificar claramente el contexto sobre el que opera?

A continuación, mostramos la implementación del State Monad:

```
instance Monad (MS s) where
    (>>=) (C c1) fc2 = C (\xs -> let
                                    (r,xs') = c1 xs
                                    (C c2) = fc2 r
                                in c2 xs')
    return k = C (\xs -> (k,xs))
```

El *return*, como de costumbre, es fácil de entender; simplemente se construye una función que, tomando cualquier estado, devuelve el mismo (contexto por defecto: función identidad) y da como salida el valor especificado en el argumento. Más complejo es seguir el razonamiento para entender cómo se ha realizado el *bind*:

- Primero, recibe como argumento un State Monad que envuelve una computación *c1*. Además, recibe una función *fc2* que indica cómo quiere el usuario que *c1* sea actualizada (no sólo qué devolverá, también qué nuevo contexto/estado debe tener).

- Para realizar el *bind*, se construye una nueva computación que, dado cualquier estado, ejecuta *c1*, obteniendo la salida *r* y el nuevo estado *xs'*. A continuación extrae de *fc2* información sobre qué computación ha requerido el usuario en caso de que la salida de *c1* fuese *r*. Teniendo dicha computación *c2*, sólo queda aplicarla sobre el estado *xs'* devuelto por *c1*.

- El resultado final es una computación que toma cualquier estado y devuelve el par (salida, estado actualizado) correspondiente a haber aplicado *c1* y después la correspondiente *c2* en función de la salida anterior.

Por tanto, debe entenderse que el operador *bind* actúa en el State Monad como un secuenciador de computaciones, haciendo la composición de la que encuentra a su izquierda con la que proporciona el usuario dentro de la función, a su derecha.

Las instancias correspondientes a Applicative y Functor pueden implementarse de la misma forma que se especificó en la sección 4.7. Por ello, no se incluye el código en esta sección, pero el lector debe asumir que ambas clases han sido también instanciadas.

Aparte del operador *bind* y la función *return*, tres funciones más son fundamentales para trabajar con el State Monad:

```
-- Gives the state of the MS
get :: MS s s
get = C (\xs -> (xs,xs))

-- Puts a state into the MS
put :: s -> MS s ()
put xs = C (\_ -> ((),xs))

-- Runs the computation stored in the MS
run :: MS s a -> (s -> (a,s))
run (C c) = c --Just unwraps the function out of the data type
```

- La función **get** describe una computación que no cambia su estado y devuelve como salida éste. Es útil cuando el usuario desea recibir información sobre el estado de la computación.
- La función **put** recibe como argumento un estado y devuelve una computación que ignora cualquier estado anterior y lo reemplaza por el especificado como argumento.
- La función **run** permite extraer la computación encapsulada en un State Monad. De esta forma, ejecutar *run (C c) s* devolvería la salida de la computación *c* asociada al estado inicial *s*.

6.2 Un simple Stack usando State Monad

Un Stack puede ser definido como un State Monad si consideramos que su estado interno es el conjunto ordenado de los valores almacenados actualmente en la pila (una lista). Se definirán para trabajar con ella las dos funciones típicas *pop* y *push* que, en el primer caso, devolverá como salida cierto valor y, en el segundo, dará un valor vacío.

```
type Stack t a = MS [t] a
```

De esta forma tan sencilla queda definida una forma de manejar un Stack de cualquier tipo genérico *t*. Sin embargo, para poder darle un uso cómodo, necesitamos implementar las dos funciones mencionadas anteriormente:

```
pop :: Stack t (Maybe t)
pop = C (\xs -> case xs of
                  [] -> (Nothing,[])
                  x:xs' -> (Just x,xs'))

push :: t -> Stack t ()
push x = C (\xs -> ((),x:xs))
```

De nuevo, con las herramientas que nos proporciona State Monad, esta tarea es sencilla:

- La función **pop** devuelve una computación que, tomando el estado anterior de la pila (y si la pila tiene algún elemento), extrae el primer elemento de ella y lo muestra como salida. En caso de que la pila esté vacía, no realiza ninguna acción sobre ella y devuelve *Nothing*.
- La función **push** recibe un valor y actualiza el estado de la pila, añadiéndolo como elemento cabeza de la lista. Esta computación no devuelve ningún resultado como salida.

Pueden definirse otras funciones básicas, que nos permitirán tener más libertad a la hora de utilizar nuestro Stack. Una muy interesante es *stackLength*:

```
stackLength :: Stack t Int
stackLength = C (\xs -> let l = foldr (\_ c -> c+1) 0 xs in (l, xs))
```

Finalmente, una vez contamos con una estructura de datos y unas funciones básicas que nos permiten manipularla, es hora de escribir programas más complejos:

```
doblePop :: Stack t (Maybe t)
doblePop = pop >> pop

clearStack :: Stack t (Maybe t)
clearStack = pop >>= (\r -> case r of
                              Just _ -> clearStack
                              otherwise -> C (\xs -> (Nothing, xs)))

pushLength :: Stack Int ()
pushLength = do l <- stackLength
                push l
```

Como podemos observar, todas estas funciones devuelven un State Monad (bajo el sinónimo Stack), por lo que pueden ser utilizadas para definir computaciones más complejas. Asimismo, la computación envuelta en cualquiera de estos State Monads puede ser ejecutada mediante el comando *run state_monad initial_stack*.

6.3 Operando en Polaca con nuestro Stack

El lector conocerá, probablemente, la Notación Polaca por la que cualquier operación aritmética puede expresarse, conservando la precedencia, sin necesidad de paréntesis. En ella, una expresión como *5*(4+3)* quedaría **5+43*.

Esta notación suele ser cómodamente evaluada con la ayuda de un Stack. Por ello, la utilizaremos para ejemplificar un código más complejo utilizando el Stack que acabamos de definir.

Nuestro programa recibirá una lista ordenada de tipo *[Either Char Int]* en la que se especifique la operación en Polaca que se desea resolver, por ejemplo: *[Left '*', Right 5, Left '+', Right 4, Right 3*. El Stack que usaremos almacenará elementos también del tipo *Either Char Int*. El código es el siguiente:

```
polaca :: [Either Char Int] -> Stack (Either Char Int) ()
polaca ((Left op):exp) = push (Left op) >> polaca exp
polaca ((Right n):exp) = do (Just x) <- pop
                            evalPop x n
                            polaca exp
polaca [] = return ()
```

En caso de que se haya terminado de evaluar la expresión (la lista sea vacía), se devuelve directamente el Stack. En caso de que la lista no esté vacía, se extrae el primer elemento y, en función de si es una operación o un número, se procede de forma distinta:

- Si es **una operación**, se almacena en la pila y se llama recursivamente a nuestra función, para evaluar el resto de la expresión.

- Si es **un número**, debe comprobarse si el *top* de la Stack es también un número, por lo que ya tendríamos los dos argumentos para aplicar una operación, o por el contrario es un operador y debemos seguir evaluando la expresión para obtener un segundo número. Este comportamiento es resuelto por la función *evalPop* que explicamos a continuación. Finalmente vuelve a llamarse recursivamente a *polaca* para evaluar el resto de la expresión.

```
evalPop :: Either Char Int -> Int -> Stack (Either Char Int) ()
evalPop (Left op) n = push (Left op) >> push (Right n)
evalPop (Right n1) n2 = ...
    ... = do Just(Left op) <- pop
             stackLength >>= (\l -> case l of
                                      0 -> push (Right res)
                                      otherwise -> do (Just x) <- pop
                                                      evalPop x res)
    [donde res = operar op n1 n2]
```

En caso de que el *top* de la Stack fuese un operador, necesitamos aún otro número para realizar la operación, por lo que introducimos éste y el número que nos llega por argumento en la Stack.
Por otro lado, en caso de el *top* de la Stack fuese un número, ya tenemos los dos argumentos que harán falta para resolver el último operador que habíamos introducido en la pila. Por ello, extraemos éste y realizamos la operación con la función *operar* cuyo código veremos más adelante. El siguiente paso depende de si la Stack está vacía (no queda ninguna operación pendiente) o si no lo está:

- Si la pila está vacía, introducimos el resultado final en ella.
- Si la pila tiene algún contenido, extraemos el *top* y volvemos a ejecutar la función *evalPop*.

Debe tenerse en cuenta que el código mostrado en el cuadro anterior no puede ser interpretado por Haskell. Se ha modificado ligeramente para que pudiese caber con un buen formato. Para utilizarlo en Haskell deben suprimirse los puntos suspensivos y sustituir *res* por el cálculo indicado abajo (una cláusula *where* no puede utilizarse en este caso, y *let in* empeoraría la legibilidad del código).

La función *operar* es muy simple:

```
operar :: Char -> Int -> Int -> Int
operar '+' n1 n2 = n1+n2
operar '*' n1 n2 = n1*n2
operar '-' n1 n2 = n1-n2
```

7 ¡Qué monada de While!

El lenguaje WHILE puede ser definido como un State Monad, donde consideramos que el estado interno es el State (Var -> Z visto en clase). Para modificar este estado interno del lenguaje WHILE vamos a definir cinco funciones:

- **assW:** Modifica el estado interno añadiendo una nueva relación Var -> Z o modificándo una ya existente.
- **skipW:** No modifica el estado interno.
- **compW:** No modifica el estado interno *explicítamente* pero establece un orden entre las diferentes sentencias del lenguaje WHILE.
- **ifW:** Como la propia sentencia del lenguaje WHILE, **ifW** realiza una biburcación en el programa según una condición.
- **whileW:** Al igual que en el **ifW**, **whileW** hace exactamente lo que define la semántica del *while*.

Con estas funciones vamos a implementar un intérprete del lenguaje WHILE y para ello vamos a definir 3 funciones auxiliares que nos ayudarán a realizarlo:

- **aValW:** Evalúa expresiones aritméticas, devuelve un entero.
- **bValW:** Evalúa expresiones booleanas, devuelve un valor booleano.
- **update:** Actualizará el estado interno inherente a nuestro programa WHILE.

7.1 Implementando una monada de WHILE

Partiendo de la definición del State Monad:

```
data MS s a = C (s -> (a,s))
```

Definimos el WHILE como un State Monad:

```
type While a = MS State a
```

Como dijimos al principio del apartado, este tipo *While* guarda un **State** como estado interno y como tipo de Salida un tipo genérico ***a***.

A continuación mostramos la implementación de las funciones auxiliares:
update:

```
update :: Var -> Z -> While ()
update x v = C (\s -> let s' y
                          | x == y = v
                          | otherwise = s y
                      in ((),s'))
```

aValW:

```
aValW :: Aexp -> While Z
aValW (N n) = C(\s -> (n,s))
aValW (V var) = C(\s -> (s var,s))
aValW (Add a1 a2) = (+) <$> aValW a1 <*> aValW a2
aValW (Mult a1 a2) = (*) <$> aValW a1 <*> aValW a2
aValW (Sub a1 a2) = (-) <$> aValW a1 <*> aValW a2
```

bValW:

```
bValW :: Bexp -> While T
bValW TRUE = C(\s -> (True, s))
bValW FALSE = C(\s -> (False, s))
bValW (Eq a1 a2) = (==) <$> aValW a1 <*> aValW a2
bValW (Le a1 a2) = (<=) <$> aValW a1 <*> aValW a2
bValW (Neg bexp) = (not) <$> bValW bexp
bValW (And b1 b2) = (&&) <$> bValW b1 <*> bValW b2
```

En la función **aValW** manejamos un WHILE que tiene como salida un entero, **Z**, y en **bValW** manejamos un WHILE que tiene como salida un booleano, **T**. Ambos evalúan dicha expresión a través de la mutación de su *State*. Para operar sobre estos contextos usamos el operador *fmap* en notación infija, <$> .

Una vez definidas estas funciones, podemos implementar las funciones principales de nuestro intérprete WHILE:

```
assW :: Var -> Aexp -> While ()
assW x a = aValW a >>= (\n -> update x n)
```

Como definimos al principio de este apartado, **assW** modifica el *State* del WHILE *actualizándolo* con una nueva relación Var -> Z, que recibe como argumentos, y esto lo realiza a través de la función *update*.

```
skipW :: While ()
skipW = return ()
```

En el caso de **skipW** se devuelve el mismo estado interno que ya tiene el contexto WHILE.

```
compW :: While () -> While () -> While ()
compW = (>>)
```

¡Eh! ¡La función **compW** es exactamente el operador *bind* que define Monad! Usamos (>>) ya que no importa qué salida tuviera la computación, pues siempre ejecutaremos la siguiente computación de la misma manera.

```
ifW :: Bexp -> While () -> While () -> While ()
ifW bexp ws1 ws2 = bValW bexp >>= (\b -> if b then ws1 else ws2)
```

En esta función vemos claramente la potencia del operador (>>=) ya que dependiendo de qué salida nos dé la función *bValW* a la hora de evaluar la condición, iremos a una rama del programa WHILE o a otra.

```
whileW :: Bexp -> While () -> While ()
whileW bexp ws = bValW bexp >>= (\b -> if b then ws >> whileW bexp ws
    else skipW)
```

De manera similar a cómo implementamos la función *ifW*, en esta función evaluamos la condición. Si es *True* ejecutaremos las sentencias del bloque del while y volveremos a llamar a **whileW** como nos indica la semántica del while. Si es *False* devolvemos el mismo estado.

8 Referencias

References

[1] Learn You a Haskell for Great Good!
`http://learnyouahaskell.com/chapters`

[2] All About Monads.
`https://wiki.haskell.org/All_About_Monads`

[3] Tutorials of Monads.
`https://wiki.haskell.org/Tutorials#Using_monads`

[4] Ejercicios intermedios de Haskell
`http://blog.tmorris.net/posts/20-intermediate-haskell-exercises/`