Programación Declarativa, 2021-1
Nota de clase 13: Funtores, Aplicativos y Mónadas*

Manuel Soto Romero

20 de enero de 2021
Facultad de Ciencias UNAM

Hemos dicho hace algunas notas que HASKELL toma sus orígenes en el Cálculo λ y la Teoría de Categorías. En esencia, la teoría de categorías es el estudio de la composición. Una categoría es una colección de objetos y morfismos entre ellos de forma tal que la composición tenga sentido. Este tipo de estructuras resultan ser muy comunes en la mayoría de los campos de las matemáticas, tiene un fuerte vínculo con la Lógica y la Teoría de Tipos a través de las llamadas Categorías Cartesianamente Cerrada. En nuestro caso, dentro de la Programación Funcional, diseño como las mónadas son originarias de la teoría de Categorías [2].

Por supuesto, estos temas quedan fuera del alcance de estas notas, sin embargo, podemos entender de manera superficial cómo es que HASKELL hace uso de estos conceptos y las principales ventajas y beneficios que traen consigo.

13.1. Funtores

Comencemos revisando la definición de las siguientes funciones:

```
inc :: [Int] → [Int]
inc []     = []
inc (x:xs) = (x+1):inc xs

sqr :: [Int] → [Int]
sqr []     = []
sqr (x:xs) = (n^2):sqr xs
```

Ambas funciones son definidas de la misma forma:

- Con la lista vacía no hacemos nada.
- Con las listas que tienen cabeza y cola aplicamos una función a la cabeza y procesamos recursivamente la cola.
- La única diferencia es la función que se aplica.

*En su mayoría traducida literalmente de *Programming in Haskell* del autor Graham Hutton.

Abstrayendo este patrón, es fácil ver la obtención de la ya conocida función `map`:

```
map :: (a → b) → [a] → [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

De esta forma, podemos redefinir las funciones anteriores como:

```
inc = map (+1)
sqr = map (^2)
```

Podemos generalizar este patrón a otras estructuras de datos, pues en un principio podemos mapear también sus elementos. A la clase de tipos que soportan tal mapeo se les llama *funtores*. Su declaración es la siguiente:

```
class Funtor f where
   fmap :: (a → b) → f a → f b
```

Es decir, para que un tipo parametrizado `f` sea instancia de `Functor`, debe implementar la función `fmap`. Es fácil ver cómo nuestras listas parametrizadas pertenecen a esta clase de tipos a partir de `map`, veamos este y algunos otros ejemplos:

Ejemplo 13.1. El ejemplo más simple, como mencionamos antes, son las listas.

```
instance Funtor [] where
   fmap = map
```

Ejemplo 13.2. Nuestro segundo ejemplo puede apreciarse con nuestro ya conocido tipo `Maybe`.

```
instance Funtor Maybe where
   fmap _ Nothing  = Nothing
   fmap g (Just x) = Just (g x)
```

Es decir, aplicar un mapeo sobre el valor de falla (`Nothing`), mantiene la falla, mientras que ara un valor de éxito (`Just`) propagamos el mapeo sobre del valor que delimita.

```
Ejemplo> fmap (+1) Nothing
Nothing
Ejemplo> fmap (*2) (Just 3)
Just 6
Ejemplo> fmap not (Just False)
Just True
```

Ejemplo 13.3. Veamos ahora el caso de los árboles:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
            deriving(Show)
```

Podemos definir la función `fmap` como sigue:

```
instance Functor Tree where
   fmap g (Leaf x)   = Leaf (g x)
   fmap g (Node l r) = Node (fmap g l) (fmap g r)
```

```
Ejemplo> fmap length (Leaf ‘‘abc’’)
Leaf 3
Ejemplo> fmap even Node (Leaf 1) (Leaf 2)
Node (Leaf False) (Leaf True)
```

La mayoría de funtores definidos en Haskell se comportan de manera similar a los ejemplos anteriores, en el sentido de que el tipo `f` es una estructura que contiene elementos de tipo `a` a los cuales llamamos *contenedores de tipo*, de forma tal que `fmap` aplica una función dada a cada uno de sus elementos.

Sin embargo, esto no siempre es así. Por ejemplo, el tipo `IO` recientemente estudiado, sabemos que funciona como un contenedor, por ejemplo `IO String` encapsula cadenas. Sin embargo, en este tipo en particular, no podemos acceder a su estructura interna, por lo que si queremos hacerlo parte de `Functor`, tendremos que auxiliarnos de la primitiva `do`.

```
instance Functor IO where
    fmap g mx = do x ← mx
                   return (g x)
```

```
Ejemplo> fmap show (return True)
‘‘True’’
```

13.1.1. Principales beneficios

- La función `fmap` puede usarse para procesar los elementos de cualquier estructura *funtorial*. Es decir, no tenemos que definir funciones con nombres distintos que en esencia hacen lo mismo.
- Podemos definir funciones *genéricas* que pueden usarse con cualquier funtor.

Ejemplo 13.4. La función `inc` puede redefinirse por medio de restricciones de clase de forma tal que pueda ser usada con cualquier funtor.

```
inc :: Functor f => f  Int → f Int
inc = fmap (+1)
```

```
Ejemplo> inc (Just 1)
Just 2
Ejemplo> inc [1,2,3,4,5]
[2,3,4,5,6]
Ejemplo> inc (Node (Leaf 1) (Lead 2))
Node (Leaf 2) (Lead 3)
```

13.1.2. Leyes de los funtores

Adicionalmente, los funtores deben cumplir con las siguientes leyes:

```
      fmap id  =  id
  fmap (g.h)  =  fmap g . fmap h
```

Estas os leyes nos indican: (1) que la función `fmap` debe preservar la identidad y (2) la composición de funciones. Estas leyes en conjunto se aseguran de que el mapeo sea correcto de forma que no se eliminen o agreguen elementos, se desordene la estructura, etc.

Ejemplo 13.5. Supongamos que tenemos la siguiente instancia para listas.

```
fmap g []     = []
fmap g (x:xs) = fmap g xs ++ [g x]
```

La definición es correcta en cuanto a los tipos. Sin embargo, al verificar las leyes, vemos que caemos en inconsistencias.

```
Ejemplo> fmap id [1,2]
[2,1]
Ejemplo> fmap (not.even) [1,2]
[False,True]
Ejemplo> (fmap not . fmap even) [1,2]
[True,False]
```

13.2. Aplicativos

De la sección anterior, podemos concluir que los funtores abstraen muy bien la idea de aplicar un mapeo sobre cada elemento de una estructura. Sin embargo, las funciones que recibe la función `fmap` tienen la característica de sólo recibir un argumento. ¿Pero qué pasa cuando los resultados finales terminan encapsulando funciones? Por ejemplo, supongamos la siguiente ejecución:

```
Ejemplo> fmap (+) [1,2,3]
???
```

En primera instancia, sabemos que lo que hará la función `fmap` será aplicar la función `(+)` a cada elemento de la lista, con lo cual obtendremos una lista con una forma similar a `[(1+),(2+),(3+)]`. Sin embargo, al ejecutarla en Haskell obtenemos un error:

```
Ejemplo> fmap (+) [1,2,3]
<interactive>:1:1: error:
* No instance for (Show (Integer -> Integer))
arising from a use of ‘print’ (maybe you haven’t applied a function to enough
arguments?)
* In a stmt of an interactive GHCi command: print it
```

Esto debido a que Haskell no tiene una forma de impresión definida sobre las funciones, sin embargo, podemos estar seguros de que obtuvimos la lista antes mencionada. Esto lo podemos verificar, revisando el tipo de la función.

```
Ejemplo> :t fmap (+) [1,2,3]
fmap (+) [1,2,3] :: Num a => [a -> a]
```

¿Y si pudiéramos usar cada una de esas funciones obtenidas con los elementos de otra lista? ¿Por qué nos limitamos a un argumento?

Si de hecho queremos usar los elementos obtenidos después de aplicar una función, mismos que quedan dentro de una estructura (como una lista o un valor de tipo `Maybe`) con otra estructura, necesitamos distintas versiones de `fmap`: una por cada argumento/estructura añadidos, es decir:

```
fmap0 :: a → f a

fmap1 :: (a → b) → f a → f b

fmap2 :: (a → b → c) → f a → f b → f c

fmap3 :: (a → b → c → d) → f a → f b → f c → f d
```

...

Sin embargo, si queremos que esto se aplique a distintos tipos de datos, tal como la función `fmap`, necesitamos definir distintas clases de tipos, una para cada una de las funciones antes mostradas. Por supuesto, esto es tedioso y prácticamente imposible, dado que podemos tener un número infinito de versiones de `fmap`. En realidad, ya nos enfrentamos a este problema con anterioridad y es justo el caso de la currificación donde las funciones en realidad no reciben mas que un argumento, usaremos este conceptos para abstraer la idea.

Otro concepto que es importante notar, es que para el caso de la función `fmap` que revisamos en la sección anterior, seguíamos prácticamente tres pasos:

1. Sacar cada elemento de la estructura.
2. Aplicarle la función.
3. Volverlo a meter en la misma estructura.

Sin embargo, en este caso, dependiendo del número de argumentos tendríamos que sacar más de una vez los valores de la estructura, lo cual no es muy óptimo. ¿Por qué no metemos entonces la función inicial en una estructura? De esta forma podemos realizar las operaciones entre estructuras, de hecho, en lugar de estructuras, es usual que se les llame contextos y así lo haremos a partir de este punto.

Nuestro objetivo es entonces mantener todo dentro del mismo contexto y apoyarnos de la currificación para definir nuestras distintas variantes de `fmap`. Definimos entonces las siguientes dos funciones:

```
pure :: a -> f a

(<*>) :: f (a -> b) -> f a -> f b
```

El primer operador `pure` es el que nos va a permitir construir contextos a partir de valores. En este caso, nuestro interés es meter las funciones en dichos contextos, para lo cual nos servirá mucho esta función. Por otro lado el operador `<*>`[1] es una generalización de la aplicación de funciones que convenientemente mantiene los resultados dentro de contextos. Este operador, al igual que la aplicación de funciones normal, asocia a la izquierda, es decir:

```
g <*> x <*> y <*> z
```

[1]Hay toda una discusión sobre el nombre de este operador, sin embargo adoptaremos la convención de llamarlo simplemente `apply`. Para más información, consultar la siguiente página: `https://stackoverflow.com/questions/3242361/what-is-called-and-what-does-it-do`

se entiende como:

```
(((g <*> x) <*> y) <*> z)
```

De esta forma, si tenemos n argumentos, tendremos:

```
pure g <*> x1 <*> x2 <*> ... <*> ... xn
```

Nótese que g es una función de aridad n y que debemos introducirla en un contexto. Una vez hecho esto, podemos operar con el resto de contextos. De esta forma, podemos definir nuestra colección de funciones de la siguiente manera:

```
fmap0 :: a → f a
fmap0 = pure

fmap1 :: (a → b) → f a → f b
fmap1 g x = puro g <*> x

fmap2 :: (a → b → c) → f a → f b → f c
fmap2 g x y = pure g <*> x <*> y
```

...

De esta forma, vemos que si un funtor soporta tanto a `pure` como a `<*>`, podemos omitir la definición de las versiones de `fmap`. A los funtores que soportan `pure` y `<*>` son llamados *funtores aplicativos* o simplemente *aplicativos*.

13.2.1. Ejemplos de Aplicativos

Al igual que los funtores, los aplicativos tienen su propia clase de tipos a la cual le llamamos `Applicative`. Por supuesto para que un tipo forme parte de esta clase debe ser parte primero de `Functor` tales como las listas, `Maybe` o `IO`. Veamos las implementaciones para estos tipos de datos.

El tipo `Maybe`

```
instance Applicative Maybe where
   pure = Just

   Nothing <*> _ = Nothing
   (Just g) <*> mx = fmap g mx
```

Con este tipo, la función `pure` mete un elemento usando el constructor `Just` que es nuestro contexto. Por otro lado `apply` de un `Nothing` es un `Nothing` debido a que usamos este tipo de dato para abstraer la idea de que ha ocurrido un error, de forma tal que debemos propagarlo sin importar los resultados restantes. En caso contrario, es decir, que tengamos un elemento dentro de un contexto `Just`, debemos aplicar la función que encapsula en contexto con el contexto siguiente. Veamos algunos ejemplos:

```
Ejemplo> pure (+1) <*> Just 1
Just 2
Ejemplo> pure (+) <*> Just 1 <*> Just 2
Just 3
Ejemplo> pure (+) <*> Nothing <*> Just 2
Nothing
```

El tipo de las Listas

```
instance Applicative [] where
   pure x = [x]

   gs <*> xs = [g x | g ← gs, x ← xs]
```

Con este tipo, la función `pure` mete un elemento simplemente encerrándolo entre corchetes, es decir una lista de un elemento, que es nuestro contexto. Por otro lado `apply` aplica cada una de las funciones contenidas en la primera lista con cada uno de los elementos de la segunda lista. Usamos listas por comprensión para que realice todas las posibles combinaciones.

```
Ejemplo> pure (+1) <*> [1,2,3]
[2,3,4]
Ejemplo> pure (+) <*> [1] <*> [2]
[3]
Ejemplo> pure (*) <*> [1,2] <*> [3,4]
[3,4,6,8]
```

El último ejemplo ilustra la generación de las posibles combinaciones. Un ejemplo de ejecución es:

```
> pure (*) <*> [1,2] <*> [3,4]
> [*] <*> [1,2] <*> [3,4]
> [g x | g ← [x], x ← [1,2]] <*> [3,4]
> [(1*),(2,*)] <*> [3,4]
> [g x | g ← [(1*),(2*)], x ← [3,4]]
> [(1*3),(1*4),(2*3),(2*4)]
> [3,4,6,8]
```

El tipo IO

```
instance Applicative IO where
   pure = return

   mg <*> mx = do{g ← mg; x ← mx; returm (g x)}
```

Observación 13.1. El uso de llaves y puntos y comas con la primitiva `do` es válido para denotar donde inicia y termina cada sentencia así como el uso de llaves para delimitar el bloque de sentencias. Recordemos que es una manera de simular un comportamiento imperativo.

Con este tipo, la función `pure` mete un elemento el contexto por medio de `return`. Por otro lado `apply` ejecuta un bloque de sentencias de forma que la primera sentencia abre el primer contexto (`g`), la segunda el segundo (`x`), aplica `g x` y lo encierra en un contexto mediante `return`.

Por ejemplo, podemos definir la función `getChars` que obtiene los caracteres contenidos en una cadena.

```
getChars :: Int → IO String
getChars 0 = return []
getChars n = pure (:) <*> getChar <*> getChars (n-1)
```

13.2.2. Leyes de los aplicativos

Al igual que los funtores, los aplicativos deben cumplir con ciertas leyes. En este caso tenemos las siguientes cuatro leyes:

```
pure id <*> x     =  x
pure (g x)        =  pure g <*> pure x
x <*> pure y      =  pure (\g -> g y) <*> x
x <*> (y <*> z)   =  (pure (.) <*> x <*> y <*> z
```

De forma general, estas leyes nos indican que:

- `pure` debe preservar la identidad.
- `pure` debe preservar la aplicación de funciones.
- El orden de evaluación de los componentes no importa.
- El operador `<*>` es asociativo.

Finalmente, Haskell provee una primitiva que permite simplificar el uso de `pure` dado que siempre lo primero que haces es aplicar la función que encapsula esta función a la estructura que se encuentra inmediatamente a la derecha. Es decir, `<$>` se define como.

```
g <$> x = fmap g x
```

Con lo cual podemos modificar nuestros códigos anteriores como sigue:

```
g <$> x1 <*> x2 <*> ... <*> xn
```

13.3. Mónadas

Hemos introducido el concepto de *mónada* en la nota pasada. Sin embargo, ahora que hemos estudiado los conceptos de *funtores* y *aplicativos* podemos estudiar algunas propiedades más interesantes. Comencemos con un pequeño ejemplo para la definición de un pequeño evaluador para expresiones aritméticas.

Supongamos el siguiente tipo de dato.

```
data Expr = Val Int
          | Div Expr Expr
```

El primer constructor representa constantes enteras y el segundo la división de dos expresiones aritméticas. Un a primera versión de nuestro evaluador se muestra a continuación:

```
eval :: Expr → Int
eval (Val n) = n
eval (Div x y) = div (eval x) (eval y)
```

Esta versión es funcional, sin embargo, cuando el segundo argumento de la división se evalúa a cero, se genera una excepción.

```
> eval (Div (Val 1) (Val 0))
*** Exception: divide by zero
```

Para solucionar esto, tenemos dos posibles opciones, mismas que verificamos en la nota pasada. Optaremos entonces por usar nuestro tipo de dato `Maybe` para definir una definición segura.

```
divisionSegura :: Int → Int → Maybe Int
divisionSegura _ 0 = Nothing
divisionSegura n m = Just (div n m)
```

Por supuesto, para usar esta definición nos obliga a cambiar el tipo de la función `eval` así como su implementación, como se muestra a continuación:

```
eval :: Expr → Maybe Int
eval (Val n) = Just n
eval (Div x y) =
   case eval x of
      Nothing → Nothing
      Just n → case eval y of
                  Nothing → Nothing
                  Just m → divisionSegura n m
```

Con lo cual nuestro evaluador soluciona el problema de la división por cero.

```
> eval (Div (Val 1) (Val 0))
Nothing
```

Aunque esta versión funciona correctamente, se aprecia que es poco legible y por otro lado es bastante tedioso tener que verificar una por una las evaluaciones de los operandos, por ejemplo, su en lugar de un división binaria tuviéramos un división n-aria, tendíamos n expresiones `case`, lo cual es poco práctico.

Sin embargo, dado que lo que estamos haciendo es aplicar la función división segura a los elementos guardaos en el contexto `just`, es posible usar el patrón que estudiamos en la sección pasada, debido a que `Maybe` es un aplicativo. Veamos una nueva versión de nuestro evaluador usando `pure` y `apply`.

```
eval :: Expr → Maybe Int
eval (Val n) = pure n
eval (Div x y) = pure divisionSegura <*> eval x <*> eval y
```

La intuición nos dice que esta función tiene sentido, sin embargo, tenemos un problema con los tipos y es que la función `divisionSegura` tiene como valor de regreso `Maybe Int`, cuando debería ser simplemente `Int`, lo cual es imposible puesto que perderíamos los casos que se manejan mediante `Nothing`.

Sin embargo, no todo está perdido, puesto que, como vimos en la nota pasada, `Maybe` es una mónada, con lo cual tenemos a nuestra disposición el operador de secuencia `>>=` y en particular la versión endulzada `do`. De forma tal que podemos reescribir nuestro evaluador como:

```
eval :: Expr → Maybe Int
eval (Val n) = Just n
eval (Div x y) = do n ← eval x
                    m ← eval y
                    divisionSegura x y
```

Como podemos apreciar, `Maybe` es tanto una mónada como un aplicativo y en realidad a los aplicativos que soportan el uso de el operador de secuencia `>>=` y `return` se les consideran mónadas. Por ejemplo, la clase de tipos `Maybe` se define como sigue:

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

    return = pure
```

Notemos que `return` es simplemente otro nombre para `pure`.

13.3.1. La mónada `State`

Hasta esto punto hemos usado a las mónadas para simular un comportamiento *imperativo*. Principalmente mediante el uso del operador de secuencia `»=` y su versión endulzada `do`. Este comportamiento se logra tomando los contextos, sacando su valor (posiblemente procesarlo) y pasando el resultado al siguiente contexto y así sucesivamente hasta finalizar la secuencia con un valor `return` que genere un contexto con el resultado.

Ahora modelaremos otra característica de la programación imperativa: el concepto de estado. Para hacer esto, haremos uso de la mónada `State`. Para fines prácticos consideraremos como "estado" a un valor entero, sin embargo, esto puede modificarse. De esta forma, definimos un tipo de dato `State` como sigue:

```
type State = Int
```

Queremos entonces poder pasar de un estado a otro, de esta forma, podemos modelar un cambio de estado como una función que toma un estado y regresa un nuevo estado (posiblemente modificado). Llamamos a las funciones de este tipo *Transformadores de Estado* o en inglés *State Transformer*. Modelamos estas funciones mediante el tipo `ST` cuya definición se muestra a continuación.

```
type ST = State → State
```

Sin embargo, lo ideal sería que partiendo de un estado, además del estado modificado, obtuviéramos el valor resultante. De nada nos sirve tener únicamente la "memoria" modificada únicamente. Podemos modificar esto, añadiendo un parámetro de tipo a nuestra definición anterior.

```
type ST  a = State → (a,State)
```

Es natural preguntar por qué no tenemos un valor de entrada. Sin embargo, esto no es del todo necesario, pues podemos definir funciones con dicho tipo de dato y simplemente regresar un transformador. Por ejemplo, si tenemos un transformador que recibe un caracter y devuelve un entero, definiríamos una función con el siguiente tipo:

```
Char → ST Int
```

Este tipo abrevia la siguiente función, que por cierto, está currificada:

```
Char → State → (Int,State)
```

El hecho de que nuestro transformador esté parametrizado, da pie a que podamos usar la primitiva `do` para escribir programas con estado, es decir, necesitamos que sea instancia de la clase de tipos `Monad`. Debido a esto, debemos cambiar la definición del tipo pues la primitiva `type` no permite hacer a los tipos parte de una clase. En su lugar usaremos la primitiva `newtype`, para lo cual debemos proveer un constructor que no hace otra cosa mas que encapsular la función antes discutida

```
newtype ST a = S (State → (a,State))
```

Si queremos hacer uso de la función envuelta por el constructor `S`, necesitamos *desencapsularla* para lo cual, definimos la función `app` que dado un estado, lo aplica con el transformador correspondiente y devuelve dicho resultado:

```
app :: ST a → State → (a,State)
app (S st) x = st x
```

Volviendo `ST` parte de `Monad`

Con esta función defina, podemos entonces crear la instancia correspondiente de `Mondad` que a su vez necesita crear las instancias correspondientes de `Applicative` y de `Functor`.

Funtor

Comencemos entonces con `Functor` para lo cual necesitamos definir la función `fmap`.

```
instance Functor ST where
   fmap g st = S (\s let → (x,s') = app st s in (g x, s'))
```

La definición de la función `fmap` realiza los siguientes pasos, por medio de la expresión `let`:

1. Debemos construir una nueva función que espera un estado.
2. Dicho estado, será pasado al transformador que recibimos como parámetro, recordando que nos regresa una tupla, en este caso `(x,s')` donde `x` es el valor obtenido y `s'` es el nuevo estado.
3. El mapeo se da, aplicando la función `g` al valor resultante `x`.
4. La función construida, debe encapsularse usando `S` que es nuestro contexto.

Aplicativo

Definamos las funciones `pure` y `(<*>)` para que forme parte de `Applicative`.

```
instance Applicative ST where
   pure x = S (\s → (x,s))

   stf <*> stx = S (\s →
      let (f,s') = app stf s
          (x,s'') = app stx s' in
         (f x, s''))
```

Recordemos que `pure` captura elementos en un contexto, en este caso, para nuestro transformador, debemos añadirlo en la tupla correspondiente. Por otro lado, la definición de `<*>` sigue los siguientes pasos:

1. Usamos una expresión `let` para evaluar nuestros dos transformadores. Al igual que hicimos con `fmap` debemos construir una función, encapsulada en el constructor `S`.
2. Primero aplicamos el transformador `stf` con el estado pasado como parámetro, esto nos devuelve la tupla `(f,s')`.
3. Con el estado obtenido `s'` aplicamos el transformador `stx` lo cual nos devuelve la tupla `(x,s'')`.
4. Finalmente realizamos la aplicación de `f` y `x` y lo encapsulamos en un resultado, con el último estado modificado `s''`. Es decir, hacemos una aplicación encadenada.

Mónada

Necesitamos definir el comportamiento del operador de secuencia »=:

```
instance Monad ST where
   st >>= f = S (\s -> let (x,s') = app st s in
      app (f x) s')
```

Recordemos que `pure` captura elementos en un contexto, en este caso, para nuestro transformador, debemos añadirlo en la tupla correspondiente. Por otro lado, la definición de `<*>` sigue los siguientes pasos:

1. Usamos una expresión `let` para evaluar nuestros dos transformadores. Al igual que hicimos con `fmap` debemos construir una función, encapsulada en el constructor `S`.
2. Primero aplicamos el transformador `stf` con el estado pasado como parámetro, esto nos devuelve la tupla `(f,s')`.
3. Con el estado obtenido `s'` aplicamos el transformador `stx` lo cual nos devuelve la tupla `(x,s'')`.
4. Finalmente realizamos la aplicación de `f` y `x` y lo encapsulamos en un resultado, con el último estado modificado `s''`.

En resumen

- El mapeo se da aplicando la función correspondiente al valor encapsulado en el transformador.
- El aplicativo se evalúa y pasa los resultados de forma encadanada, pasando el resultado de cada transformador al siguiente de forma encadenada.
- Finalmente, el operador de secuencia, aplica el primer transformador y pasa su resultado al siguiente, tal y como cualquier otro contexto.

Reetiquetando árboles

Ejemplo 13.6. Para ejemplificar el uso estos transformadores, partamos de un árbol de tipo parametrizado.

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
            deriving(Show)
```

Supongamos que tenemos un árbol de caracteres, definido como sigue:

```
tree :: Tree Char
tree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')
```

Una aplicación del uso de transformadores se da reetiquetando dicho árbol con valores enteros, sin embargo, queremos que dichas etiquetas se generen dependiendo de los valores generado por otras ramas, usando un contador, por ejemplo. No podemos simplemente definir una función recursiva, pues necesitamos conocer el estado de otros subárboles. Veamos una primera implementación.

```
rlabel :: Tree a -> Int -> (Tree Int, Int)
rlabel (Leaf _) n = (Leaf n, n+1)
rlabel (Node l r) n = (Node l' r', n'')
   where (l',n') = rlabel l n
          (r',n'') = rlabel l n'
```

```
Ejemplo> fst(rlabel tree 0)
Node (Node (Leaf 0) (Leaf 1)) (Leaf 2)
```

Ejemplo 13.7. La definición anterior funciona, sin embargo, en cada paso necesitamos conocer el estado exacto del cómputo. Podemos simplificar la función por medio de transformadores. Definiremos una función `fresh` que genere los números de manera consecutiva, de forma que la función anterior no lidie con estos valores, llamamos a esta función `fresh`.

```
fresh :: ST Int
fresh =  S (\n → (n,n+1))
```

En pocas palabras, dado un estado `n`, `fresh` genera un nuevo número, sumando simplemente 1. Con esto, redefinimos nuestra función `rlabel` como `alabel`, recordando que `ST` es un aplicativo:

```
alabel ::  Tree a -> ST (Tree Int)
alabel (Leaf _) = Leaf <$> fresh
alabel (Node l r) = Node <$> alabel l <*> alabel r
```

```
Ejemplo> fst (app (alabel tree) 0)
Node (Node (Leaf 0) (Leaf 1)) (Leaf 2)
```

Ejemplo 13.8. Dado que `ST` también es una mónada, podemos definir también la función usando la primitiva `do`. La diferencia con nuestra versión que usa aplicativos, será que debemos dar un nombre a cada uno de los pasos intermedios.

```
mlabel :: Tree a → ST (Tree Int)
```

```
mlabel (Leaf _) = do n <- fresh
                     return (Leaf n)
mlabel (Node l r) = do l' <- mlabel l
                       r' <- mlabel r
                       return (Node l' r')
```

13.3.2. Funciones genéricas

Otro de los beneficios que obtenemos al abstraer el concepto de mónada, es la posibilidad de definir funciones genéricas para cualquier tipo de mónada. A continuación, se presentan algunos ejemplos de estas funciones.

`mapM` Una versión monádica de `map` en el cual, los resultados devueltos por la función son mónadas y devolvemos los mismos encapsulados en una mónada con dicha lista. Se añade un ejemplo de uso con el apoyo de `Maybe`.

```
mapM :: Monad m ⇒ (a → m b) → [a] → m [b]
mapM f [] = return []
mapM f (x:xs) = do y ← f x
                   ys ← mapM f xs
                   return (y:ys)

conv :: Char → Maybe Int
conv c
   | isDigit c = Just (digitToInt c)
   | otherwise = Nothing
```

```
Ejemplo> mapM conv "1234"
Just [1,2,3,4]
Ejemplo> mapM conv "123a"
Nothing
```

`filterM` Una versión monádica de `filter`. Podemos usar esta función para calcular el conjunto potencia de una lista.

```
filterM :: Monad m ⇒ (a → Bool) → [a] → m [a]
filterM p [] = return []
filterM p (x:xs) = do b ← p x
                      ys ← filterM p xs
                      return (if b then x:ys else ys)
```

```
Ejemplo> filterM (\x -> [True,False]) [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

`join` Ahora definimos una versión generalizada de `concat` que funciona como un aplanador de listas.

```
join :: Monad m ⇒ m (m a) → m a
join mmx = do mx ← mmx
              x ← mx
              return x
```

```
Ejemplo> join [[1,2],[3,4],[5,6]]
[1,2,3,4,5,6]
Ejemplo> join (Just (Just 1))
Just 1
Ejemplo> join (Just Nothing)
Nothing
Ejemplo> join Nothing
Nothing
```

13.3.3. Leyes de la Mónadas

Finalmente, tenemos las leyes de las mónadas:

```
return x >> f       =  f x
mx >>= return       =  mx
(mx >>= f) >>= g    =  mx >>= (\x -> (f x >>= g))
```

De forma general, estas leyes nos indican que:

- `return` es la identidad del operador de secuencia.
- El operador de secuencia es asociativo.

Todas las mónadas que hemos revisado en esta nota, cumplen estas leyes.

Referencias

[1] Diego Pedraza, *Teoría de Categorías y Programación Funcional,* Universidad de Sevilla, Proyecto de Titulación, 2018.

[2] Graham Hutton, *Programming in Haskell*, Tercera Edición, Cambridge University Press, 2008.