Universidad de Buenos Aires
Facultad de Ingenieria

Teoria de Lenguaje

Haskell

1^{r} cuatrimestre 2015

Sebastian Gavrilov, 96252
Pablo Hazan, 96522
Esteban Bellegarde, 95381

14 de julio de 2015

Índice

1. Inferencia de tipos **2**

2. Recursión **4**

3. Pereza **4**

4. Garbage Collection **4**

5. Funciones de orden superior **5**
5.1. Funciones como parametro . . . 5
5.2. Funciones como resultado . . . 6
5.3. Plieges (folds) . . . 6
5.4. Funciones Anónimas (lambdas) . . . 7

6. Functores **7**
6.1. Leyes de los Functores . . . 8
6.2. Functores Aplicativos . . . 8

7. Mónadas **8**
7.1. Las Leyes de las Mónadas . . . 9
7.2. Mónadas comunes . . . 10
7.3. Manejo de errores . . . 10
7.3.1. Mónada Maybe . . . 10
7.3.2. Either . . . 10
7.3.3. Mónada Error . . . 11
7.4. Mónada List . . . 11
7.5. Mónada IO . . . 12
7.6. Mónada State . . . 12
7.7. Mónada Reader . . . 13
7.8. Mónada Writer . . . 13

8. Zippers **14**

9. Concurrencia **15**
9.1. Threads . . . 15
9.2. Comunicación entre hilos . . . 15
9.3. Comunicación por canales . . . 16
9.4. Aclaraciones y problemas . . . 16

10. Paralelismo **16**

11. Software Transactional Memory **17**

1. Inferencia de tipos

Haskell permite definir funciones sin decir de que tipos son estas, esto, muy lejos de querer decir que Haskell no usa un tipado fuerte, es debido a que realiza inferencia de tipos. Un puede escribir una función que sume dos números como:

```
1 sumar :: (Num a) => a -> a -> a
2 sumar x y = x + y
```

Donde la primer linea indica que las variables a son de la clase de tipos Num. Si en GHCI usamos el comando :t para pedir el tipo de la funcion sumar tendremos:

```
1 :t sumar
2 sumar :: (Num a) => a -> a -> a
```

Este resultado era evidente, ya que en la propia definición lo habíamos indicado. Pero Haskell permite definir la función sumar de la siguiente manera:

```
1 sumar x y = x + y
```

Al utilizar el comando :t en GHCI obtendremos la misma respuesta que antes, aunque esta vez no le hayamos indicado al lenguaje los tipos en la definición. Este resultado proviene de que Haskell realiza inferencia de tipos basandose en el algoritmo de Hindley Milner o Damas Milner. Este algoritmo sigue seis reglas lógicas:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad [\texttt{Var}]$$

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'} \quad [\texttt{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\ x\ .\ e : \tau \to \tau'} \quad [\texttt{Abs}]$$

$$\frac{\Gamma \vdash e_0 : \sigma \qquad \Gamma,\ x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau} \quad [\texttt{Let}]$$

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \quad [\texttt{Inst}]$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall\ \alpha\ .\ \sigma} \quad [\texttt{Gen}]$$

Para explicar que quieren decir estas seis reglas primer es necesario entender la sintaxis y el significado de alguno de los símbolos.

Lo primero es la barra horizontal, esta viene a cumplir el rol de una implicación lógica, o sea, si el antecedente, lo de arriba, es verdadero entonces, necesariamente, el consecuente, lo de abajo, también lo es. Cuando en la parte superior hay N hipótesis, para asegurar el consecuente es necesario que se cumplan las N hipótesis, es como si las separaciones por espacio estubiesen indicando una operación and

lógica. Lo siguiente son los dos puntos ":", estos indican de que tipo es una expresión, por ejemplo x : Int indica que x es del tipo int. El contexto se representa con γ. Para indicar que algo se encuentra en un determinado contexto usamos $\in$. $\vdash$ es básicamente que se puede demostrar algo, lo precede el contexto necesario para demostrar lo que quiere. La coma ",ës utilizada para ampliar el contexto, una unión, si entendemos al contexto como el conjunto de referencias a variables de determinados tipos, donde se pisa el elemento si este ya se encontraba en el contexto original. Por ultimo, el símbolo $\sqsubseteq$ indica una especie de inclusión, en realidad es mas bien una herencia, donde el tipo de la izquierda es un subtipo del de la derecha. Con todos la notación clara es sencillo entender las seis reglas. Por Ejemplo, la segunda indica que si en nuestro contexto tenemos una expresión e0 que toma un elemento del tipo τ y devuelve otro que es del tipo τ', y en ese mismo entorno tenemos una segunda expresión que es del tipo τ, entonces podemos decir que en ese mismo entorno que la expresión e0 e1 es del tipo τ'. La quinta nos dice que si en un entorno la expresión e es del tipo σ' y que σ' es un subtipo de σ entonces la expresión e es del tipo σ.

El algoritmo de inferencia de tipos usa estas reglas de forma recursiva. Realiza un ajuste de patrones, donde mira la expresión que tiene, y en cual de los consecuentes encaja mejor, una vez encontrado va a los antecedentes y mira que necesita para que estos se cumplen, si con la información actual es suficiente entonces ha terminado de inducir los tipos, en caso contrario toma estos antecedentes que aun no tiene determinados y vuelve a realizar el mismo ajuste de patrón, y así va infiriendo en cada sub-expresión los tipos necesarios. Cuando encuentra que la información alcanzada es suficiente para cumplir TODAS las hipótesis la inferencia esta completa.

Si hacemos el seguimiento del como funciona este algoritmo en la función sumar que tenemos más arriba, y utilizando la currificación de Haskell tenemos que la función es de la forma

```
sumar x y = x + y
```

que es lo mismo que

```
sumar x y = (+) x y
```

para ajustar el patrón tomamos

```
e0 = (+) x
e1 = y
```

esto encaja en el segundo patrón, donde se realiza e0 e1 que es del tipo τ', para esto se induce que e0 es del tipo $\tau \to \tau$' y que e1 es tipo τ'. Hasta ahora solo sabemos que y es del tipo τ. Luego hay que ajustar el patrón e0. Donde volvemos a caer en el segundo caso. Entonces tenemos que :

```
e'0 = (+)
e'1 = x
```

Con e'0 e'1 es del tipo $\tau \to \tau$' Concluimos que, dado que e'0 es del tipo $\tau 1 \to$ t1' y que e'1 es del tipo $\tau 1$, como e'0 e'1 = e0 entonces e0 : τ'1 = $\tau \to \tau$' Hasta aquí tenemos que

```
y : τ
x : τ1
(+) :  τ1 → τ → τ'
```

Además, como (+) : (Num a) =>a $\to$ a $\to$ a Por la primera regla tenemos que $\tau 1 \to \tau \to \tau$' = a $\to$ a $\to$ a O sea

```
τ1 = a
τ = a
t' = a
```

Y, finalmente, usando la quinta regla tenemos que

```
τ1 ⊑ a
τ ⊑ a
t' ⊑ a
```

Finalmente solo resta recomponer todo, usando que sumar = e'0 e'1 e1 El tipo de sumar es (Num a) =>a $\to$ a $\to$ a.

2. Recursión

La recursión o recursividad es la forma de especificar una función basandose en su propia definición. Es una parte muy importante de Haskell. Una función es recursiva cuando una parte de su definición incluye a la propia función. Necesita por lo menos un *caso base* que no hace llamado recursivo para que exista una condición límite.

Este ejemplo muestra una función de factorial recursiva, separando claramente el caso base.

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Muchas de las funciones comunes en Haskell se pueden definir de forma recursiva, por ejemplo el *length*, la función que devuelve el número de elementos de una lista:

```
length []     = 0
length (x:xs) = 1 + length xs
```

Recursión es usada para definir casi todas las funciones de manejo de números y listas, sin embargo en la práctica no es de usarse tan seguido: la recursividad está abstraida por las funciones de las librerias de Haskell, permitiendo al programador razonar sus problemas en más alto nivel. Por ejemplo, la función de factorial de ejemplo escrita anteriormente se puede definir de la siguiente manera:

```
factorial n = product [1..n]
```

3. Pereza

Una característica destacable de Haskell es que es *perezoso* (o *no estricto*). Esto significa que nada se va a evaluar hasta que sea directamente necesario - la evaluación queda diferida hasta que el resultado es requerido por otra computación.

El siguiente es un ejemplo de la pereza de Haskell:

```
let (a, b) = (length [1..5], reverse "hola mundo") in 1 + 1
```

Como la expresión después del *in* no utiliza los valores de a y b, estos quedan sin evaluar, ya que no son necesarios. Tambien pueden no ser necesarios por completo, o ser *parcialmente* requeridos:

```
let (a, b) = (length [1..5], reverse "hola mundo")
    'o':ss = b
```

Como solo es necesaria la primera letra para concluir el matcheo de patrones, la evaluación es parcial.

Las funciones en Haskell pueden ser perezosas o estrictas. Podemos analizar si una función es perezosa o estricta pasandole **undefined** y viendo si su ejecución falla. Esto se debe a que en Haskell la evaluación forzada del **undefined** siempre termina en un error.

```
let (x, y) = undefined in x -- Error!
length [undefined, undefined, undefined]
-- No hay error, length es perezoso
```

La evaluación perezosa tiene muchas ventajas, como la reutilización de código, posibilidad de generar estructuras de datos infinitas y definiciones circulares, pero su mayor inconveniente es que el uso de memoria se vuelve muy dificil de predecir, por ejemplo las expresiones `3+2 :: Int` y `5 :: Int` denotan el mismo valor pero pueden tener diferentes tamaños en memoria.

4. Garbage Collection

Muchos lenguajes implementan un Garbage Collection (GC) para ocuparse de los objetos no alcanzables y liberar dicha memoria.

Los lenguajes imperativos suelen generar una cantidad de basura a lo largo de su ejecución, y si estos constan de un GC simplifican la vida del programador, al desligarlo de la necesidad de liberar el espacio no utilizado, y de no deber perder nunca la referencia. De lo contrario, si un lenguaje posee un GC, el programador debe olvidarse de lo que no necesite, simplemente haciendolo inalcanzable, de este modo las referencias almacenadas son solos las que en ese momento resultan útiles, el GC pasará en algún momento y liberará la memoria no alcanzable.

Los lenguajes funcionales, como lo es Haskell, suelen generar muchísima mas basura que los lenguajes imperativos, esto es debido a la inmutabilidad de sus variables, y para guardar un nuevo valor es necesario crear una nueva variable. Cada iteración en un llamado recursivo genera un nuevo valor. En Haskell no es insólito producir 1Gb de datos en un segundo. Para esto el compilador de Haskell GHC tiene un potente GC que se ocupa de gestionar la basura en forma eficiente.

La inmutabilidad de las variables en Haskell no solo obliga a generar gran cantidad de basura, sino que esta característica es aprovechada para realizar la recolección. Para esto utiliza el criterio de generación: datos jóvenes y viejos. Las variables mas viejas, al ser inmutables, no apuntan nunca a un dato joven, pues los datos jóvenes no existen en el momento en que las variables viejas son creadas. Esta es la idea que utiliza el recolector de basura de Haskell para aumentar la eficiencia. El recolector de basura no mirará toda la memoria en la que estuvo trabajando nuestro programa, sino que sólo revisará entre los valores más jóvenes, y liberará los que no estén señalados, que suelen ser la gran mayoría, dado el comportamiento recursivo. Esto también es una ventaja, porque en realidad, mientras mas basura joven aparezca menor es el trabajo que realiza el recolector de basura; esto, que resulta tan poco intuitivo se explica con la forma que se utiliza para almacenar los datos jóvenes y los viejos. Los datos jóvenes son almacenados en un bloque de memoria especial, una “guardería”, cuando esta guardería se llena el GC mira sólo en esta memoria quienes son alcanzables, por lo tanto útiles y los copia en la memoria de las variables mas viejas, luego nos habilita a reutilizar la “guardería”, la cual se encuentra “vacía”, ya que cualquier dato que pisemos no será útil, o lo tendremos copiado con los valores mas viejos. Este es el motivo por el cual, con mayor cantidad de basura joven, la recolección es más rápida, lo que sucede es que hay menos datos que copiar.

5. Funciones de orden superior

Una función de orden superior es aquella que puede tomar funciones como parámetro, o devolver una función como resultado, o ambas cosas.

Haskell no solo soporta las funciones de orden superior, sino que hace un uso permanente de esta cualidad, y de forma muy natural.

5.1. Funciones como parametro

Funciones como parámetro

Una función que puede tomar como parámetro a otra función es considerada de orden superior. En Haskell esto se utiliza todo el tiempo para filtrar datos mediante algún criterio, realizar una acción sobre un conjunto de datos, etc.

Un ejemplo de una función que viene por defecto en Haskell que toma funciones como parámetro es la función filter, a la cual se le pasa una función y una lista, filter llama a la función con cada elemento de la lista, si la función devuelve true el elemento es añadido a la lista que da como resultado.

Si se implementa una función fQuickSort a la que le paso una función f que dado un elemento de la lista me devuelve un valor comparable, y una lista que quiero ordenar de la siguiente forma:

```
1 fQuickSort :: (Ord b) => (a -> b) -> [a] -> [a]
2 fQuickSort _ [] = []
3 fQuickSort f (x:xs) =
4   let
5       menores = fQuickSort f [a | a <- xs, (f a) <= (f x)]
6       mayores = fQuickSort f [a | a <- xs, (f a) > (f x)]
```

```
  in menores ++ [x] ++ mayores
```

En esta implementación es realmente versátil, ya que dependiendo que función f se utilice para medir el "tamaño" de los elementos de la lista obtendremos un resultado distinto, y no restringimos los elementos de la lista a elementos ordenables, dado que el criterio de orden los obtenemos en función de la "medida" de los elementos.

5.2. Funciones como resultado

Dijimos que una función también es de orden superior si puede dar como resultado otra función. Esto es útil, pues se puede crear constructores de funciones, estos constructores recibirán un valor, y devolverán una función para cada valor que tomen. Por ejemplo:

```
mutiplicarPor :: (Num a =>) a -> (a -> a)
multiplicarPor x = (*) x
```

Para cada valor de x, multiplicarPor da como resultado una función que recibe un numero y devuelve el el producto del numero pasado con x.

En realidad, Haskell utiliza las funciones de orden superior todo el tiempo, ya que las funciones de Haskell solo pueden tomar una única variable, esto es lo mismo que decir que las funciones en Haskell están Currificadas. No hay ninguna contradicción, cuando tenemos una función que aparenta recibir más de una variable lo que en realidad tenemos es una función que recibe un dato y nos devuelve una función, la cual toma el siguiente dato, y nos devuelve otra función y así sucesivamente. Esta es la explicación del porque cuando anotamos en la definición de tipos de una función no diferenciamos entre los parámetros de entrada y el valor de retorno.

La Currificación hace evidente la necesidad de que el lenguaje soporte funciones de orden superior. A medida que vamos aplicando parcialmente la función vamos obteniendo nuevas funciones como resultado, y este es el concepto de que una función sea de orden superior por devolver una función. Usando el ejemplo anterior, el fQuickSort es un constructor de sorts, al que se le pasa una función f y nos da un sort que ordena con un determinado criterio, lo mismo pasa con filter y map.

En Haskell este tipo de funciones son tan comunes que tiene montones de aplicaciones en las librerías standar utilizando funciones de orden superior, ya mencionamos las funciones filter y map, pero se incluyen muchos más.

5.3. Plieges (folds)

Unas funciones particularmente útiles y cómodas son los pliegues (folds). Como en Haskell las variables son inmutables no existen los iteradores clásicos de los lenguajes imperativos, sino que se utiliza la recursividad de las funciones, esto es tan común que existen algunas funciones útiles tienen incorporado estos patrones para realizar iteraciones. Si querríamos implementar la función elem utilizando pliegues podríamos hacerlo como

```
elem' :: (Eq a)=> a -> [a] -> Bool
elem' y ys = foldl f False ys
  where f acc x = if x==y then True else acc
```

Lo que estamos haciendo es pasarle a nuestro pliegue una función que devuelve True si el elemento que recibe es el mismo que el que nosotros buscamos, y sino devuelve el acc, además la función recibe como "segundo" parámetro el valor inicial del acumulador que va a pasarle a la función, y una lista sobre la que deseamos que itere. Lo que hace foldl es agarrar la cabeza de la lista, pasarselo a la función f, y tomar el valor obtenido como nuevo acumulador, luego se llama a si mismo con la cola de la lista, la misma función y este nuevo acumulador. El resultado que devuelve foldl es el ultimo valor que devuelve el acumulador.

Si quisiéramos implementar foldl podríamos hacerlo así:

```
foldl' :: (a->b->a)-> a->[b]->a
foldl' _ acc [] = acc
foldl' f acc (x:xs) = foldl' f (f acc x) xs
```

5.4. Funciones Anónimas (lambdas)

Al usar este tipo de funciones solemos tener que crear funciones con el único objetivo de pasarlas a nuestras funciones de orden superior, lo cual no tiene mucho sentido, para esto aparecen las denominadas funciones anónimas, o funciones lambdas. Las funciones lambda son expresiones (devuelven un valor), por eso podemos pasarlas como parámetros a funciones de orden superior. Su sintaxis es:

```
(\a b -> 2a/b)
```

donde a y b son los parámetros que recibe, lo que sucede a ->indica que el comportamiento de dicha función. Suelen estar encerradas entre paréntesis para delimitarlas, de lo contrario tomarán todo el renglón. Utilizando este tipo de funciones, podríamos haber escrito nuestro pliegue como

```
elem' :: (Eq a)=> a -> [a] -> Bool
elem' y ys = foldl (\acc x-> if x==y then True else acc) False ys
```

6. Functores

Los functores son una clase de tipos que tiene un método llamado fmap, donde

```
fmap :: (a -> b) -> f a -> f b
```

La idea de fmap es que toma una funcion de a en b, un functor que contiene un a, y devuelve un functor que tiene un b. Una idea intuitiva de un functor podría ser como una caja, fmap abre esta caja, aplica f al contenido y devulve una caja con el resultado de aplicar la función.

Un término más correcto para definir lo que es un funtor sería contexto computacional. El contexto sería que la computación podría tener un valor, o podría fallar. Si queremos un constructor de tipos que sea una instancia de functor este debe pertenecer a la familia $* -> *$.

Algunos functores básicos son las listas [], Maybe, Either a (en este ultimo caso, como necesitamos que nuestro tipo tome un único tipo concreto y Either toma dos debe estar parcialmente evaluda), si se definiese una clase Tree a también podría ser un funtor. Otros no tan claros son IO y (->) r.

Las acciones IO son como cajas que encierran datos que provienen del mundo real, o que saldrán al mismo, podemos ligar el contenido de una acción IO a una variable usando <-, trabajar con estos datos, realizar algunos calculos, pero cuando querramos sacarlos al mundo exterior es necesario transformarlos en otra acción IO usando return. Si queremos mapear un valor IO lo que en realidad buscamos es obtener una nueva acción IO que contenga el resultado de aplicar una determinada función, esto será:

```
instance Functor IO where
  fmap f action = do
    result <- action
    return (f result)
```

El tipo de una función r ->a se puede reescribir como (->) r a esto indica que (->) r es una caja de a. Como ->es de la familia $* -> * -> *$ debe estar parcialmente aplicado, igual que Either.

```
instance Functor ((->) r) where
  fmap f g = (\x -> f (g x))
```

Aplicar fmap a una función nos da una nueva función, si pensamos que aplicamos esta nueva funcion a un valor esto daría por resultado la función original aplicada a nuestro valor, y a este resultado le aplicaría la función que le pasamos a fmap, esto no es otra cosa que realizar la composición de funciones.

```
instance Functor ((->) r) where
  fmap = (.)
```

6.1. Leyes de los Functores

Ahora vamos a ver las leyes de los functores. Para que algo sea una functor, debe satisfacer una serie de leyes. Se espera que todos los functores exhiban una serie de propiedades y comportamientos. Deben comportarse fielmente como cosas que se puedan mapear. Al llamar fmap sobre un functor solo debe mapear una función sobre ese funtor, nada más. Este comportamiento se describe en las leyes de los functores. Hay dos de ellas que todas las instancias deFunctor deben cumplir, pero Haskell no hace esta comprobación, debe ser verificada por el usuario que la implementa.

La primera ley de funtores establece que si mapeamos la función identidad sobre un functor, el functor que obtenemos debe ser igual que el original (fmap id = id).

La segunda ley dice que si mapeamos el resultado de una composición de dos funciones sobre un funtor debe devolver lo mismo que si mapeamos una de estas funciones sobre el funtor inicial y luego mapeamos la otra función ((f . g) = fmap f)

6.2. Functores Aplicativos

Dentro de un functor podemos tener almacenado cualquier tipo de valor, y como en Haskell las funciones son ciudadanos de primer orden estas también son un valor, por ende podemos tener funciones encerradas dentro de un functor, por ejemplo, si hacemos

```
fmap (*) Just 7
```

obtenemos

```
Just (* 7)
```

Si queremos aplicar la función almacenada a el valor almacenado por Just 11 podriamos tratar de hacerlo, pero si queremos un comportamiento que funcione en todos los functores debemos crear algo más general, este es el motivo por el cual aparecen los functores aplicativos.

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

La definición de esta clase de tipos nos dice que si alguien es un Applicativa también es un Functor. Las funciones no están implementadas por defecto, pero pure toma un valor y lo encapsula dentro de un functor aplicativo. La segunda funcion es una ampliacion de fmap, toma un functor aplicativo, "extrae" su función, y mapea dicha función sobre otro functor. La clase de tipos maybe los implementa de la sigueinte forma:

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

7. Mónadas

Las **mónadas** en Haskell se pueden pensar como descripciones *componibles* de computaciones. Presentan la posibilidad de separar la combinación de computaciones de su ejecución y permiten acarrear datos extra implícitamente en adición al resultado de la computación, que *se producirá* cuando la mónada sea corrida. De esta manera permiten suplementar las funcionalidades *puras* con I/O, estado, indeterminismo, etc.

En terminos del lenguaje una mónada es un tipo parametrizado que es instancia de la clase *Monad*. Su definición es la siguiente:

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

Podemos ver la mónada como un contenedor para un valor **a**. La función **return** se ocupa de poner ese valor adentro de la mónada. Entonces la función (>>=), tambien conocida como *bind*, aplica la función que se le pasa por parámetro al contenido de la mónada obteniendo como resultado otra mónada (obviamente la función pasada tiene que tener el tipo adecuado). Se puede ver como funciona en el siguiente ejemplo:

```
putStrLn "Como te llamas?"
>>= (\_ -> getLine)
>>= (\name -> putStrLn ("Hola, " ++ name ++ "!"))
```

El operador (>>=) se ocupa de tomar el valor del lado izquierdo y combinarlo con la función del lado derecho para producir un valor nuevo. El ejemplo de arriba se puede reescribir con la notación **do**, que es un azucar sintáctico alrededor del operador *bind*:

```
do
   putStrLn "Como te llamas?"
   name <- getLine
   putStrLn ("Hola, " ++ name ++ "!")
```

Ese código puede parecer de un lenguaje imperativo, y de hecho lo es: otra forma de ver las mónadas es pensar que son la abstracción necesaria para suplementar las funcionalidades que no cuadran adentro del paradigma funcional.

La implementación mas sencilla del (>>=). toma el valor del lado izquierdo, le aplica la función y devuelve el resultado, sin embargo se vuelve realmente útil cuando esa implementación hace algo extra.

7.1. Las Leyes de las Mónadas

Las mónadas por convención deben cumplir las siguientes leyes:

```
-- Identidad por la izquierda
return x >>= f = f x

-- Identidad por la derecha
m >>= return = m

-- Asociatividad
(m >>= f) >>= g = m >>= (x -> f x >>= g)
```

Para poder entenderlo mejor y usarlo de una forma más sencilla, el módulo Control.Monad define el operador de composición de mónadas:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
(m >=> n) x = do
                y <- m x
                n y
```

Con el uso de ese operador las tres reglas pueden escribirse de la siguiente forma:

```
-- Identidad por la izquierda
return >=> f = f

-- Identidad por la derecha
```

```
f >=> return = f

-- Asociatividad
(f >=> g) >=> h = f >=> (g >=> h)
```

7.2. Mónadas comunes

La siguiente tabla lista las mónadas más comunes usadas en Haskell, denotando el problema que tratan de solucionar en términos imperativos.

Mónada	Semántica imperativa
Maybe	Excepción anónima
Error	Excepción con descripción
State	Estado global
IO	Entrada y Salida
[] (list)	Indeterminismo
Reader	Entorno
Writer	Logger

En las siguientes secciones se explicarán algunas de las mónadas que aparecen en esta tabla.

7.3. Manejo de errores

7.3.1. Mónada Maybe

Es una de las mónadas más utilizadas y es muy sencilla a la vez. La definición de esta Mónada es la siguiente:

```
data Maybe a = Nothing | Just a

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

La mónada Maybe incorpora la posibilidad de encadenar computaciones que pueden devolver Nothing como resultado, en cuyo caso la cadena terminaría antes. Por ejemplo:

```
maybeHalf :: Int -> Maybe Int
maybeHalf a
        | even a = Just (div a 2)
        | otherwise = Nothing
```

La función que acabamos de definir devolverá la mitad de un número, si es par, y **Nothing** si es impar. La podemos usar de la siguiente manera:

```
ghc> Just 10 >>= maybeHalf
Just 5
ghc> Just 10 >>= maybeHalf >>= maybeHalf
Nothing
ghc> Just 10 >>= maybeHalf >>= maybeHalf >>= maybeHalf
Nothing
```

Sin el uso de la mónada Maybe nos veriamos obligados a anidar if y else para las llamadas consecutivas.

7.3.2. Either

Otra mónada muy usada es *Either*. Su tipo está definido de la siguiente manera:

```
data Either a b = Left a | Right b
```

Su propósito es muy parecido al de Myabe, Left es considerado un error mientras que Right - un valor normal. La diferencia es que Left permite guardar un valor (a diferencia del Nothing en el caso de Maybe).

Podemos escribir una función que divida un número por una lista de enteros uno por uno segura utilizando **Either**:

```
divBy :: Integral a => a -> [a] -> Either String [a]
divBy _ [] = Right []
divBy _ (0:_) = Left "divBy: division by 0"
divBy numerator (denom:xs) =
    case divBy numerator xs of
      Left x -> Left x
      Right results -> Right ((numerator `div` denom) : results)
```

Y la podemos utilizar de la siguiente forma:

```
ghci> divBy 50 [1,2,5,8,10]
Right [50,25,10,6,5]
ghci> divBy 50 [1,2,0,8,10]
Left "divBy: division by 0"
```

7.3.3. Mónada Error

La mónada Error es la forma que tiene Haskell de simular las *excepciones*. Se logra de la siguiente forma: se encadenan computaciones que pueden lanzar excepción derivando la ejecución a la instancia que puede manejarla. Esta es la definición de **MonadError**:

```
class Monad m => MonadError e m | m -> e where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a
```

Un ejemplo de uso es:

```
example :: (Error e, MonadError e m) => m String
example = throwError (strMsg "Esto es un error")
    `catchError` const (return "Atrapo el error")

example >>= putStrLn    -- Imprime "Atrapo el error"
```

Note que el tipo MonadError lleva 2 parametros, **e**, el tipo de error, y **m**, constructor que representa una mónada. Mas adelante en la definición se ve que el **throwError** y **catchError** son métodos de la clase MonadError, siendo los equivalentes a las estructuras familiares de otros lenguajes. Eso tiene la ventaja de poder definir su significado para cada mónada particular.

7.4. Mónada List

Resulta que la lista es una mónada tambien! Las listas se utilizan para modelar las computaciones no determinísticas que pueden devolver un número de resultados arbitrario. Se define de la siguiente manera:

```
instance Monad [] where
    m >>= f = concat (map f m)
    return x = [x]
```

El return es fácil de entender: simplemente devuelve una lista de un elemento. El operador *bind* tambien se entiende si consideramos que tiene que cumplir con el tipo `[a] -> (a -> [b]) -> [b]` (por su defenición más general). La función **f** aplica a cada elemento y, dado su tipo, retorna una lista - por eso es necesario el **concat** al final, para a la salida obtener una lista.

¿Por qué las listas son mónadas? Se explica con el siguiente ejemplo de notación monádica de una lista:

```
foo = do
  x <- [1 .. 10]
  y <- [2, 3, 5, 7]
  return (x * y)
```

foo es un múltiplo de x e y, con x siendo un número no determinístico entre 1 y 10, e y siendo 2, 3, 5 o 7. Este ejemplo es la notación larga y monádica de una comprensión de lista, podría ser escrita así:

```
foo = [x * y | x <- [1 .. 10], y <- [2, 3, 5, 7]]
```

7.5. Mónada IO

Dado que Haskell es un lenguaje funcional y perezoso, no podemos expresar las efectos reales de las operaciones de entrada y salida con funciones puras. De hecho, estas operaciones no se pueden ejecutar perezosamente, ya que esto haria que los efectos reales sean impredecibles. La mónada **IO** es el medio para representar las acciones de entrada/salida como valores de Haskell, para que el programador pueda manipularlos con funciones puras.

7.6. Mónada State

La mónada *State* permite acarrear estado a lo largo de una ejecución. Con ella se puede hacer lo siguiente: dado un valor de estado, se produce un resultado y un nuevo valor de estado. Esta es su definición:

```
newtype State s a = State { runState :: (s -> (a,s)) }

instance Monad (State s) where
  return x = State $ \s -> (x,s)
  (State h) >>= f = State $ \s -> let (a, newState) = h s
                                      (State g) = f a
                                  in  g newState
```

Como se ve en la declaración de tipo, State es solo una abstracción de una función que toma un estado, devuelve un valor intermedio y un nuevo estado.

Un ejemplo de uso de State es una pila. La función **push** agrega un elemento al tope de la pila y **pop** saca uno. Sin State tendriamos que arrastrar la pila como argumento a estas funciones, lo cual no es lo más cómodo de usar. Con State la podemos definir de la siguiente forma:

```
type Stack = [Int]

pop :: State Stack Int
pop = State $ \(x:xs) -> (x,xs)

push :: Int -> State Stack ()
push a = State $ \xs -> ((),a:xs)
```

Un ejemplo de uso de la pila sería:

```
stackManip = do
    push 3
    a <- pop
    pop
```

La implementación de State en Control.Monad también es instancia de la clase MonadState que define 2 funciones muy útiles:

```
put newState = State $ \_ -> ((), newState)
get = State $ \st -> (st, st)
```

Las cuales nos permiten manejar el estado de una manera más sencilla.

7.7. Mónada Reader

Algunos problemas de programación requieren computaciones en un cierto entorno (como un set de variables), pero no requieren la generalidad de la mónada **State**. La mónada **Writer** permite aportarle a una computación un entorno. Esta es su definición de tipo y su instancia de mónada:

```
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad (Reader e) where
    return a          = Reader $ \e -> a
    (Reader r) >>= f = Reader $ \e -> runReader (f (r e)) e
```

En el siguiente ejemplo se puede ver facilmente su utilidad:

```
import Control.Monad.Reader

greeter :: Reader String String
greeter = do
    name <- ask
    return ("Hola, " ++ name ++ "!")
```

```
ghci> runReader greeter "Jorge"
"Hola, Jorge!"
```

Usamos la función **ask** que esta definida en la clase **MonadReader**. Esta es su definición y el como Reader implementa esa clase:

```
class MonadReader e m | m -> e where
    ask    :: m e
    local  :: (e -> e) -> m a -> m a

instance MonadReader (Reader e) where
  ask         = Reader id
  local f c = Reader $ \e -> runReader c (f e)
```

Esta clase provee de funciones para que el uso de **Reader** sea más cómodo. **ask** recupera el entorno y **local** ejecuta la computación en el entorno modificado.

7.8. Mónada Writer

La mónada **Writer** sirve para producir un stream de datos adicional a los valores computados. Es útil para generar un log sobre una ejecución. Esta es la definición del tipo **Writer**. Como con los ejemplos anteriores, nos permite redefinir su comportamiento:

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

A continuación se puede ver un ejemplo sencillo de uso del Writer:

```
import Control.Monad.Writer

doubleWithLog :: Int -> Writer String Int
doubleWithLog x = do
          tell ("Acabo de duplicar " ++ (show x) ++ "!")
          return (x * 2)
```

Si ejecutamos esta función, obtendremos lo siguiente:

```
ghci> runWriter $ doubleWithLog 8
(16, "Acabo de duplicar 8!")
```

8. Zippers

Los zippers tienen como objetivo lograr un movimiento de forma eficiente en una estructura, conocida, que almacena datos. En un zipper se almacena el dato al que se ha llegado por un movimiento y, además, la información necesaria para reconstruir toda la estructura original.

Un problema muy sencillo para ver la utilidad de los zippers es el de querer aplicar una función en una hoja de un árbol, y luego aplicar la misma función en el hermano de dicha hoja. Si no se contase con una estructura del tipo zipper, habría que realizar todo el recorrido hasta la hoja, luego aplicar la función, y luego volver, desde la raíz a realizar el mismo recorrido, salvo en el último movimiento, y nuevamente aplicar la función. Usando los zippers podemos movernos hasta la hoja deseada, y como podemos reconstruir la estructura, podemos pedir el padre, y además saber si el nodo actual era el hijo izquierdo o el derecho, para así saltar al nodo hermano y aplicar nuevamente la función. Con este ejemplo se observa que al querer realizar determinadas acciones con bloques de información de una estructura, y estos datos se encuentran "próximos" en dicha estructura, al construir un zipper para esta estructura nos ahorramos recorrer reiteradas veces la estructura original, mejorando la velocidad de nuestro programa.

Volviendo al ejemplo del árbol binario podemos definir las estructuras de la siguiente forma:

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
data Crumb a = LeftCrumb a (Tree a) | RightCrumb a (Tree a) deriving (Show)
type Breadcrumbs a = [Crumb a]

type Zipper a = (Tree a, Breadcrumbs a)
```

Aquí se crea el tipo árbol, el cual puede estar vacío o tener un dato y otros dos árboles. El tipo miga, el cual dice si nos movimos a la derecha o a la izquierda, contiene la información que contenía el nodo padre, y además tiene el subárbol no visitado. Además se crea el sinónimo de tipo migas de pan, que simplemente es una lista de migas. Por ultimo se crea el tipo zipper, el cual tiene el árbol en el cual estaremos parados, y además las migas de pan del camino recorrido. Con todos estos tipos de datos definidos podemos crear funciones que nos permitan movernos por de forma adecuada por los árboles.

```
goUp :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)

goRight :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goRight (Node x l r, bs) = (r, RightCrumb x l:bs)

goLeft :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a )
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

Con estas tres funciones podemos movernos libremente, y eficientemente, sobre el árbol. Si una vez alcanzada una determinada posición queremos realizar un cambio sobre ese árbol podemos usar la siguiente función

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Node x l r, bs) = (Node (f x) l r, bs)
modify f (Empty, bs) = (Empty, bs)
```

Esta función recibirá una función y un zipper, y nos devolverá un nuevo zipper, el cual es el resultado de aplicarle la función al nodo actual. Además podrían crearse otras funciones útiles, por ejemplo, una función que reciba un árbol y devuelva un zipper de dicho árbol, utilizando el árbol pasado como la raíz, una función que dado un zipper indique si se puede subir, o estamos en la raíz del árbol, otra que agregue/reemplace un subárbol por otro que le pasemos, otra que se ocupe de llegar a la raíz, etc.

Otra estructura, más sencilla, en la que son útiles los zippers son las listas, donde los movimientos posibles serían avanzar o retroceder:

```
type ListZipper a = ([a],[a])

goForward :: ListZipper a -> ListZipper a
goForward (x:xs, bs) = (xs, x:bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b:bs) = (b:xs, bs)
```

La primera lista del zipper es la cola de la lista original que resulta del movimiento, la segunda es una lista invertida de los elementos que están por delante del elemento actual. En este caso los zippers son mas simples que en los árboles, pero no por esto dejan de ser útiles. En Haskell los Strings son un sinónimo de tipo de una lista de caracteres, o sea que podemos usar un zipper para movernos dentro de un texto, y modificar y o agregar caracteres en el mismo, esto es realmente útil si estamos creando un editor de texto.

9. Concurrencia

Un programa concurrente necesita realizar varias tareas al mismo tiempo. Estas tareas no necesariamente tienen que estar relacionadas entre si. El correcto funcionamiento de un programa concurrente no necesita varios núcleos.

En contraste, un programa paralelo soluciona un solo problema con el mejor rendimiento posible, empleando para eso más de un núcleo.

9.1. Threads

Un hilo es una acción *IO* que se ejecuta independientemente de los otros hilos. Los hilos en Haskell no son determinísticos. Para crear un thread, usamos la función *forkIO* del módulo *Control.Concurrent*.

Un ejemplo de uso podría ser la compresión de un archivo:

```
import Control.Concurrent (forkIO)
import qualified Data.ByteString.Lazy as L
import Codec.Compression.GZip (compress)

compressFile = L.writeFile "files/foo.gz" . compress

do
    content <- L.readFile "files/foo.txt"
    forkIO (compressFile content)
    putStrLn "Gracias por comprimir!"
    return ()
```

9.2. Comunicación entre hilos

La forma más simple que tiene un hilo para comunicarse con otro es compartiendo una variable. En Haskell podemos lograrlo utilizando una variable sincronizable **MVar**. Una **MVar** actúa como una caja con un solo elemento, que puede estar vacia o llena. Si tratamos de poner un valor adentro de una **MVar** que ya está llena, la ejecución del hilo en cuestión se suspenderá hasta que otro hilo la vacie. De la misma manera, si intentamos sacar el valor de una **MVar** vacia, el hilo se suspenderá hasta que algún otro thread le agregue algo. En el siguiente ejemplo se puede ver el uso de una **MVar**:

```
import Control.Concurrent

```

```
do
  m <- newEmptyMVar
  forkIO $ do
    v <- takeMVar m
    putStrLn ("Recibido " ++ show v)
  putStrLn "Enviando"
  putMVar m "Hola!"
```

Utilizamos la función *newEmptyMVar* para crear una MVar, *putMVar* para ponerle un valor adentro y *takeMVar* para sacarlo. Notese que solo podemos utilizar una MVar adentro de un bloque *do*.

9.3. Comunicación por canales

Utilizando el tipo *Chan*, podemos crear una comunicación mediante un canal. El siguiente código ejemplifica su uso:

```
import Control.Concurrent
import Control.Concurrent.Chan

do
  ch <- newChan
  forkIO $ writeChan ch "Hola!"
  forkIO $ writeChan ch "Hola desde otro hilo!"
  readChan ch >>= print
  readChan ch >>= print
```

Si el *Chan* está vacio, el bloque *readChan* espera hasta que haya un valor para leer. La función *writeChan* nunca se bloquea, escribe al canal inmediatamente.

9.4. Aclaraciones y problemas

Para tener en cuenta: **MVar** y **Chan** no son *estrictos*, su contenido no es evaluado sin explicitarlo. Además, el **writeChan** siempre acepta lo que se le escribe. Si un hilo escribe más frecuentemente que otro lee los mensajes, el canal crecerá desatendidamente.

Ambos métodos de comunicación entre hilos sufren de una serie de problemas típicos de concurrencia de estado compartido (en el caso de *Chan* se debe a que internamente están implementados con Mvars). Los problemas más conocidos son *deadlock* y *starvation*. En una situación de *Deadlock* dos o más hilos quedan suspendidos para siempre esperando el acceso a un recurso compartido. *Starvation* ocurre cuando un recurso es tomado por mucho tiempo por un hilo, mientras otro hilo no puede progresar, esperando el acceso (y probablemente la ejecución de ese último hilo es mucho más rápida).

Esos problemas típicos se tratan de solucionar con formas más recientes de hacer concurrencia, como por ejemplo *Software Transactional Memory*, que analizaremos más adelante.

10. Paralelismo

Para muchos problemas costosos de resolver, podríamos obtener el resultado más rápido separando la solución y evaluandola en varios núcleos. Por defecto ghc utiliza un solo núcleo. Para poder tomar provecho de múltiples núcleos necesitamos pasarle la opción **-threaded** al compilador a la hora de linkear.

Un ejemplo de uso de evaluación paralela sería la siguiente función, inspirado en el famoso algoritmo Quicksort. El siguiente código es el sort secuencial:

```
sort :: (Ord a) => [a] -> [a]
sort (x:xs) = lesser ++ x:greater
  where lesser = sort [y | y <- xs, y < x]
        greater = sort [y | y <- xs, y >= x]
sort _ = []
```

De la siguiente manera lo convertimos en un código que se ejecuta en paralelo.

```
import Control.Parallel (par, pseq)

parSort :: (Ord a) => [a] -> [a]
parSort (x:xs) = force greater `par` (force lesser `pseq`
                                      (lesser ++ x:greater))
                 where lesser = parSort [y | y <- xs, y < x]
                       greater = parSort [y | y <- xs, y >= x]
parSort _ = []
```

Es solo un poco más complicado que la versión inicial: agregamos las funciones **par**, **pseq** y **force**. La función **par** evalúa el argumento de la derecha a *Weak Head Normal Form* y retorna el de la izquierda. Para programas paralelos *Weak Head Normal Form* significa que la expresión está evaluada hasta el constructor de más afuera. Por ejemplo, para la expresión [**1, 2, 3**], *WHNF* es evaluar el constructor de lista y no el contenido de la misma. Lo interesante del **par**, es que *puede* evaluar el argumento a la izquierda en paralelo.

La función **pseq** es similar, solo que garantiza que el elemento a la izquierda será evaluado a *WHNF* antes del argumento de la derecha. **force** es necesario para forzar la evaluación de la lista. Podemos definir **force** de la siguiente manera:

```
force :: [a] -> ()
force xs = go xs `pseq` ()
  where go (_:xs) = go xs
        go [] = 1
```

La función **par** no promete que una expresión será evaluada en paralelo con otra, pero si promente hacerlo si ẗiene sentido.̈ Esta promesa le da la posibilidad de actuar inteligentemente a la hora de ejecutar un **par**. En tiempo de ejecución, el programa puede decidir que la expreción es muy liviana, como para que la ejecución paralela tenga sentido. O puede notar que todos los núcleos están ocupados en el momento.

11. Software Transactional Memory

Software Transactional Memory, **STM**, es un módulo que nos da herramientas simples, pero poderosas, que buscan solucionar los problemas de concurrencia comunes (los mencionamos en el capítulo 9).

La idea es muy sencilla: ejecutamos un bloque de acciones como una transacción, utilizando la función **atomically**. Una vez que entramos a ese bloque, otros hilos no pueden ver ninguna modificación hasta que terminemos la ejecución, y tampoco podemos ver modificaciones realizados por otros hilos desde ese bloque. Por esas propiedades decimos que nuestra ejecución está *isolada*.

Cuando salimos de la transacción, puede ocurrir una de las dos posibilidades:

1. Si ningún otro hilo modifica los mismos datos, todas las modificaciones ocurridas durante la transacción serán visibles para los otros threads.

2. En el otro caso, todas las modificaciones hechas se descartan y nuestro bloque de acciones se vuelve a ejecutar automáticamente.

En el siguiente ejemplo sencillo de una cuenta bancaria podemos ver su funcionamiento:

```
import Control.Concurrent.STM

type Account = TVar Int

withdraw :: Account -> Int -> STM ()
withdraw acc amount = do
    bal <- readTVar acc
    writeTVar acc (bal - amount)

```

```
deposit :: Account -> Int -> STM ()
deposit acc amount = withdraw acc (- amount)

atomically $ do
    acc <- newTVar (20 :: Int)
    deposit acc 20
    withdraw acc 10
    readTVar acc
```

STM es una mónada parecido a la **IO**. **TVar** funciona muy parecido a **MVar**, solo que solo se puede usar adentro de un bloque *atomically*.

Entre otras funciones útiles de **STM** se encuentra **retry**, que permite reiniciar una transacción desde 0 (sin aplicar las modificaciones). Por ejemplo, si quisiéramos solo sacar una cantidad limitada de dinero de nuestra cuenta bancaria, lo podriamos escribir de la siguiente manera:

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount = do
    bal <- readTVar acc
    if amount > 0 && amount > bal
    then retry
    else writeTVar acc (bal - amount)
```

orElse permite hacer una acción en el caso de que otra falle (sea reiniciada):

```
limitedWithdraw2 :: Account -> Account -> Int -> STM ()
limitedWithdraw2 acc1 acc2 amt
    = orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)
```

Referencias

[1] ¡Aprende Haskell por el bien de todos!, http://aprendehaskell.es

[2] Real World Haskell, http://book.realworldhaskell.org

[3] "What part of Milner-Hindley do you not understand?", http://stackoverflow.com/questions/12532552/what-part-of-milner-hindley-do-you-not-understand

[4] Recursion, http://en.wikibooks.org/wiki/Haskell/Recursion

[5] Recursion Patterns, https://www.fpcomplete.com/school/starting-with-haskell/introduction-to-haskell/3-recursion-patterns-polymorphism-and-the-prelude

[6] GHC Memory Management, https://wiki.haskell.org/GHC/Memory_Management

[7] Laziness, http://en.wikibooks.org/wiki/Haskell/Laziness

[8] Understanding Monads, http://en.wikibooks.org/wiki/Haskell/Understanding_monads

[9] Monad, https://wiki.haskell.org/Monad

[10] What is a monad?, http://stackoverflow.com/questions/44965/what-is-a-monad

[11] Monad laws, https://wiki.haskell.org/Monad_laws

[12] All about monads, https://wiki.haskell.org/All_About_Monads

[13] The State Monad: A Tutorial for the Confused?, http://brandon.si/code/the-state-monad-a-tutorial-for-the-confused/

[14] Yet Another Monad Tutorial (part 6: more on error-handling monads), http://mvanier.livejournal.com/5343.html/